Fanorona

Final Design Document (Milestone 4)

Group 1

Darren Chay Loong
Jessen Dasilva
Martin Guevara
Nadeem Howlader
Carson Mifsud
Min Park
Will Pringle (Representative) wpringle@uoguelph.ca

Executive Summary

This document goes into the details of the design of the game Fanarona, a strategic 2-player board game that originates from Madagascar, dating back to the 1680s, where 2 players try to capture all the stones of their opponents in a similar style to checkers. The board, which can have different sizes (such as 9x5) is composed of intersections and players can move their stones to those intersections by doing a capturing move such as an approach of withdrawal, or a non-capturing move.

In order to better understand the design of the application (game), we first look at a couple of use cases which will serve as guides to understand the flow of how some of the events that could occur within the game and how the different entities interact with each other. To summarize those interactions, a high-level entity diagram was created to represent the basis of our design. The application design is further elaborated by analyzing the sequence diagrams, application class diagram, and application class list, where we see that the game logic is mainly found within the game subsystem, which includes a board region and there is an external Player class.

Within the game subsystem, we see the Match class, which will be used to manage a match between 2 players, and therefore interacts with the Player, MoveValidator, and Board classes. As their name suggests, the external Player class is used to simulate a player, the Board class, found within the board region, is used to retrieve and send information to the board (such as moving stones on the board, or getting the stone information in one location), and the MoveValidator which is used to check the validity of a move. It is also important to note that the Match class will be the main point of interaction between the model and the controller (in the MVC model).

We then proceed to take a more detailed look at the client-server interactions. This can be seen by analyzing the controller class diagram, the controller sequence diagram, and the controller class details. Here we see that the HomeController is the main entry point for our system and they will be able to log in or sign up which reroutes them to the SessionsController and UsersController respectively. Once logged in, they would be able to start and join a game, thus calling the MatchesController, which communicates with the Match class mainly via the Action Cable classes. A more detailed look at the data used throughout the program can be seen via the data model. Finally, URL calls, rails sequence diagrams, and rails web page sketches all provide a visual context of the client-server interactions.

Important limitations to take note of are that, while Fanarona can be played on different board sizes, the program currently only supports a single board size (i.e 9x5). Also, since a tie in the game involves extremely complex conditions, those conditions were mostly omitted.

When implementing the design, it is crucial that the rules are understood well before implementing the MoveValidator class, since the algorithms for that class are one of the most complex stones in the design and are closely related to the rules of the game. Finally, the Action Cable classes are a crucial part of the design which links the game logic to everything else and should be well understood before implementing.

Table of Contents

Game Rules

- High level description of the game:
    - Fanorona is a turn-based game consisting of moving and capturing pieces across the board's edges that can go vertically, horizontally, and diagonally
- Goal of game
    - Capture all your opponent's pieces
- Board and pieces
    - Single board consisting of a web of 45 edges organized specifically and connected horizontally, vertically, and diagonally, with no interception
    - Pieces are simple round objects contrasted by their colors (either black, or white)
- Valid moves and captures
    - A single turn consists of:
        - Capture sequences
            - A capture is signaled either by moving your piece into the adjacent space of an opponent's piece or starting in the adjacent space and moving directly away from an opponent's piece.
            - This move will capture every opponent's piece in the line of that movement
            - A single capture can be made, or you can choose to make other sequential captures (if available). Restrictions for additional captures include:
                - You must move the same piece that made the capture
                - You cannot return to any space twice
                - You cannot move in the same direction twice

          Or

        - A single relocation move
            - A piece can be moved one space vertically, horizontally, or diagonally
    - In order of priority:
        - If there exists a possible capture sequence; you must execute the capture sequence
        - Otherwise, you may execute a relocation move
- Rules of the game are taken from: https://www.boardspace.net/fanorona/english/rules.html

Use Cases

UC1: **Starting a match**

- ➢ Primary Actor: Player
- ➢ Goal: Player should be able to start a match for Fanorona.
- ➢ Stakeholders List:
    - Player: The person with white stones.
    - Opponent: The person with black stones.
    - Board Displayer: Displays the game board and the stones to the Player and the Opponent.
    - Match Manager: Responsible for assigning the colour to the Player and Opponent and also creates the board.
- ➢ Initiating Event: None
- ➢ Main success Scenario:
    1. The Player provides their name to the Match Manager.
    2. The Match Manager assigns the Player with white colour stones.
    3. The Match Manager assigns the Opponent with black colour stones.
    4. Board Displayer sets up the board according to the official Fanorona rules and player selection.
    5. Board Displayer displays the initial game board and the starting positions of the stones.
- ➢ Pre Conditions:
    - None
- ➢ Post Conditions:
    - The Player and Opponent are able to start playing the game and the Player may begin their turn.
- ➢ Alternate Flows or Exceptions: None
- ➢ Use cases Utilized:
    - None
- ➢ Scenario Notes:
    - None

UC2: **Making a move**

- ➢ Primary Actor: Player
- ➢ Goal: Player wants to move a stone.
- ➢ Stakeholders List:
    - Player: The person making a move.
    - Move Validator: Validates that the move the Player is making is valid.
    - Match Manager: Responsible for storing the stone and location information from the Player.
    - Opponent: The opposition in the Fanorona match.
- ➢ Initiating Event: Player’s turn.
- ➢ Main success Scenario:
    1. The Player informs the Match Manager the stone they wish to move.
    2. The Move Validator validates that the stone can move.

3. The Player informs the Match Manager which location they want the stone to move to.
4. The Move Validator validates that the stone can move to the location.
5. The Move Validator moves the stone to the location for a paika.

➢ Pre Conditions:
    - The Opponent finishes making their turn.
    - It is the first turn of the game.
➢ Post Conditions:
    - The Player moves a stone.
➢ Alternate Flows or Exceptions:
    5a. Capturing an Opponent's stone(s):
        5a. 1. Perform UC3: **Capturing a stone**
    4a. Winning a Game:
        4a. 1. There is only one move left for the Player to win the game.
        4a. 2. Perform UC4: **Winning a match**
➢ Use cases Utilized:
    - UC3: **Capturing a stone**
    - UC4: **Winning a match**
➢ Scenario Notes:
    - None

UC3: **Capturing a stone**

➢ Primary Actor: Player
➢ Goal: Player wants to capture Opponents stones.
➢ Stakeholders List:
    - Player: Capturing an Opponent's stone(s) on the game board.
    - Opponent: Player captures their stones.
    - Move Validator: Verifies a stones movement was valid according to the game rules.
➢ Initiating Event: The Player has made a move.
➢ Main success Scenario:
    1. Move Validator verifies whether the stone moves to a valid location.
    2. Move Validator checks if capturing at least one of the Opponent's is possible.
    3. The Player captures the Opponent's stone(s) and removes them from the game board.
    4. Perform UC3: **Capturing a stone**
➢ Pre Conditions: A player has at least one stone with the ability to capture at least one of their Opponents stones.
➢ Post Conditions: The Player removes at least one of the Opponent's stones from the game board.
➢ Alternate Flows or Exceptions:
    1a. Finish turn:
        1a. 1. The Player may choose to finish their turn, after capturing at least one of the Opponent's stones.
        1a. 2. The Players turn ends.
        1a. 3. The control will turn over to the Opponent.
    1b. Out of moves:

1b. 1. The Player does not have any more stones they can capture.
1b. 2. The Players turn ends.
1b. 3. The control will turn over to the Opponent.

- ➢ Use cases Utilized:
  - UC3: **Capturing a stone**
- ➢ Scenario Notes:
  - None

UC4: **Winning a match**

- ➢ Primary Actor: Player
- ➢ Goal: A winner will be chosen.
- ➢ Stakeholders List:
  - Player: The winner of the game.
  - Opponent: Does not win the game.
  - Match Manager: Facilitating the match.
- ➢ Initiating Event: It is the Players turn.
- ➢ Main success Scenario:
  1. The Opponent does not have any black stones remaining on the game board.
  2. The Player has at least one white piece remaining on the board.
  3. The Match Manager declares the Player as the winner of the match.
  4. Match Manager asks the Player and Opponent if they want to rematch or quit the game.
  5. Player and Opponent agree to rematch.
  6. Perform UC1: **Starting a match**
- ➢ Pre Conditions: Player captures the Opponent's remaining stone(s).
- ➢ Post Conditions: The Player and Opponent are in a new match.
- ➢ Alternate Flows or Exceptions:

4a. Player or Opponent decides to quit the match:
4a. 1. The Player and Opponent exit the game.

- ➢ Use cases Utilized:
  - UC1: **Starting a match**
- ➢ Scenario Notes:
  - None

UC5: **Forfeiting a match**

- ➢ Primary Actor: Player
- ➢ Goal: Player wants to forfeit while playing a match.
- ➢ Stakeholders List:
  - Player: Want to end the game whenever they want.
  - Opponent: The opposition.
  - Match Manager: Facilitate the match.
- ➢ Initiating Event: It is the Player's turn and they want to forfeit the match.
- ➢ Main success Scenario:
  1. Player informs the Match Manager they want to forfeit the match.

2. The Match Manager confirms with the Player if they would like to forfeit the match.
3. Player confirms to end the match.
4. The Match Manager removes all of the Players stones from the game board.
5. Perform UC4: **Winning a match**

➢ Pre Conditions: Player is participating in a match of Fanorona.

➢ Post Conditions: Player exits from the match and the Match Manager declares the Opposition as the winner.

➢ Alternate Flows or Exceptions:

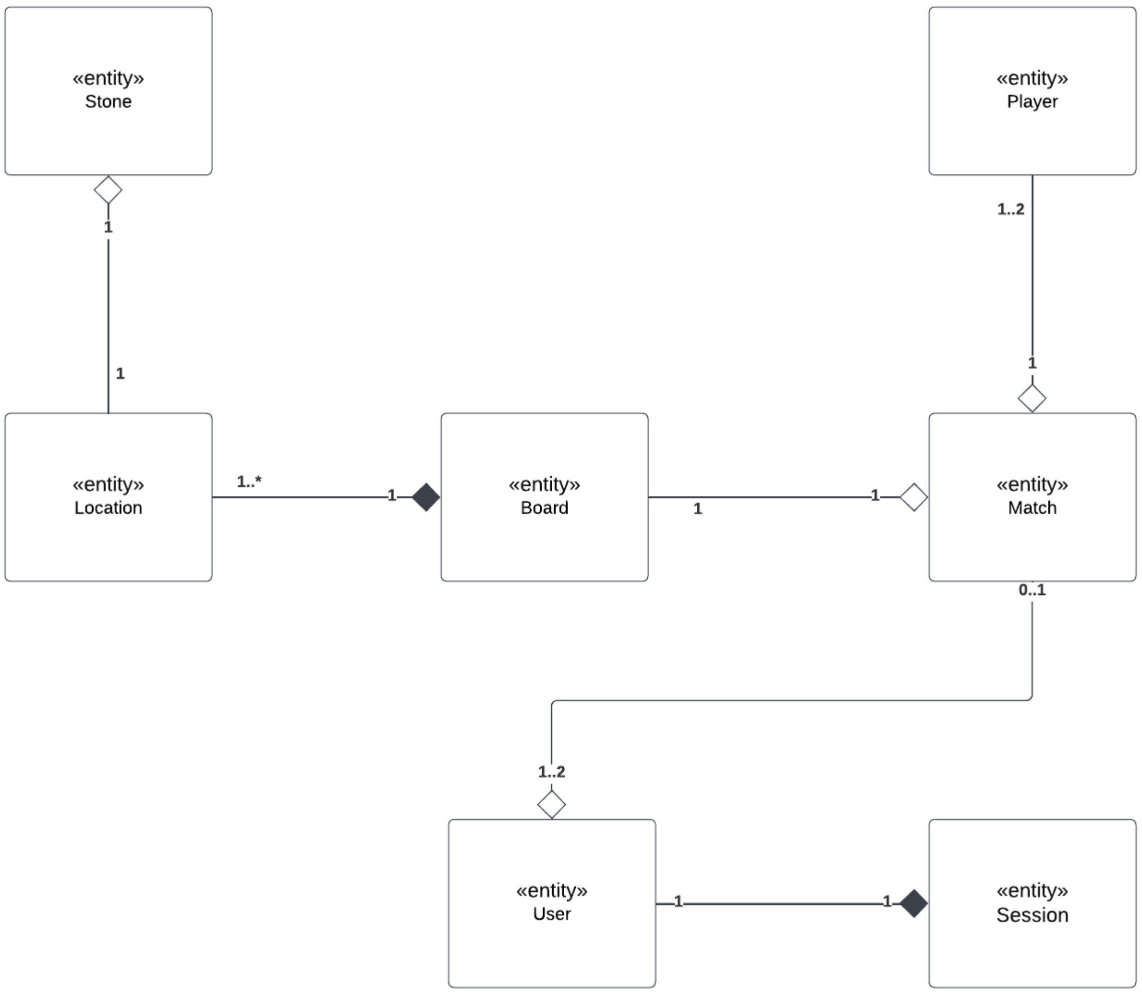
    3a. Player does not confirm to end the match:

        3a. 1. Match Manager returns control to the Player.

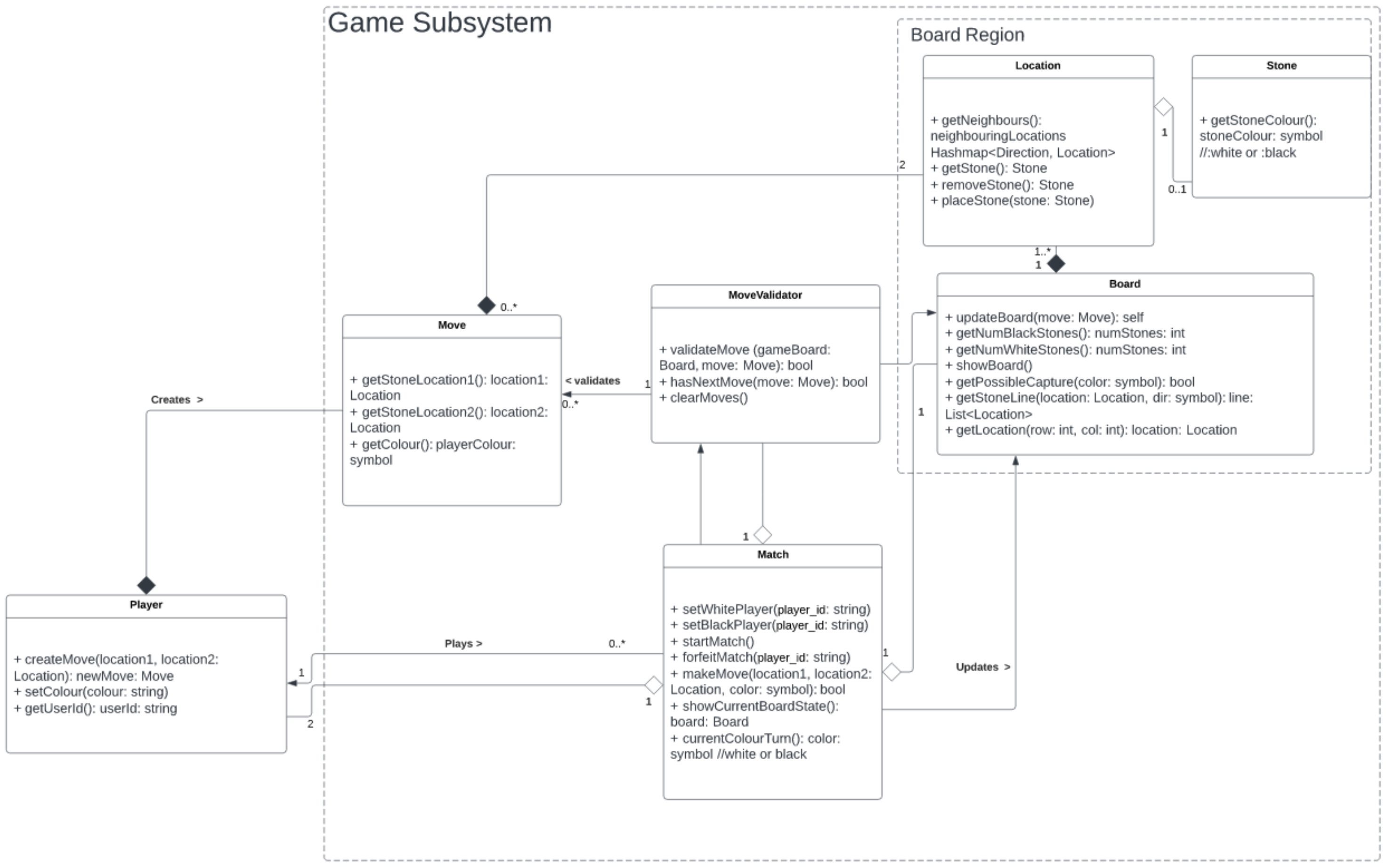
➢ Use cases Utilized:
- UC4: **Winning a match**

➢ Scenario Notes:
- None

Entity Diagram



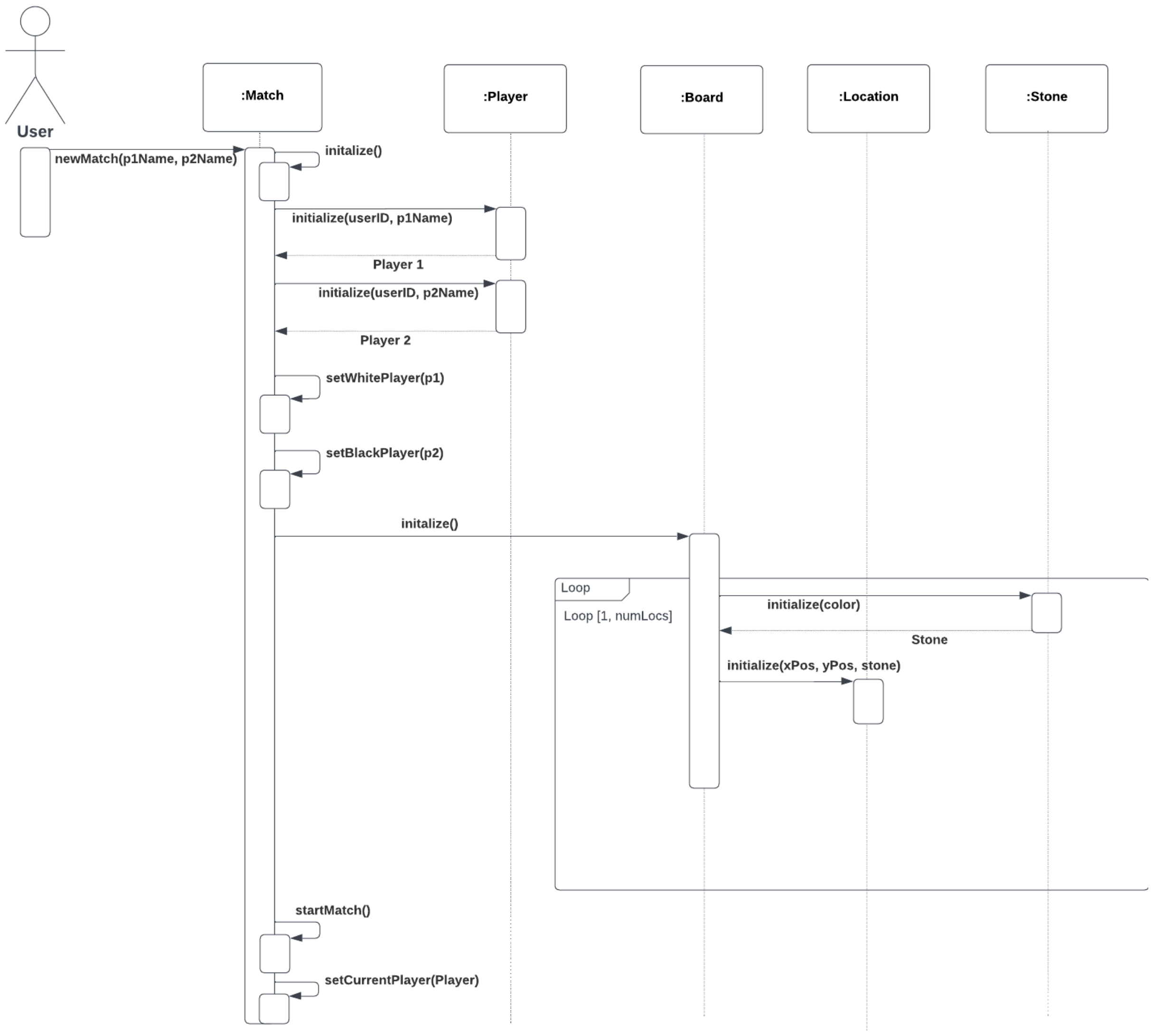
Application Class Diagram

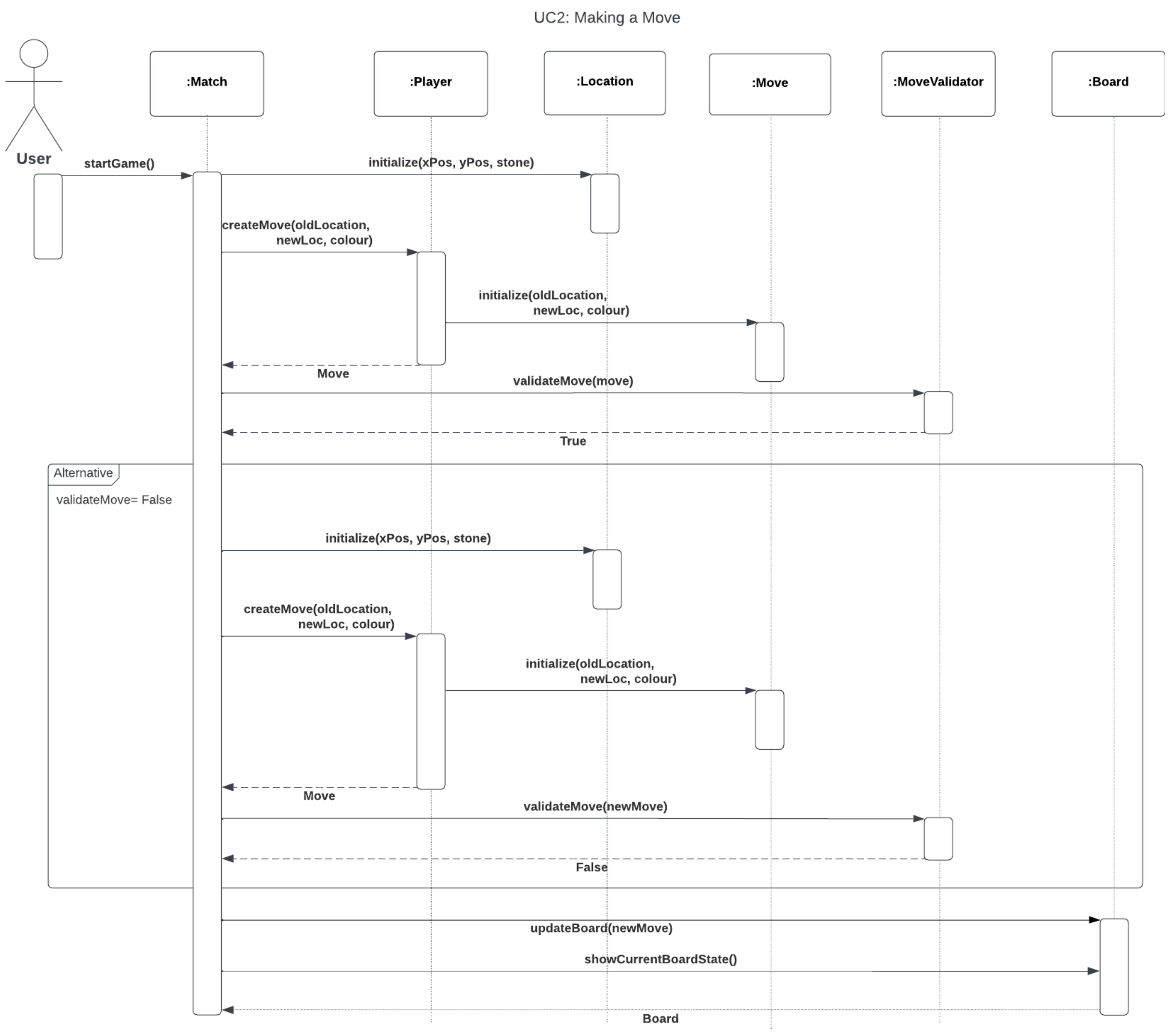


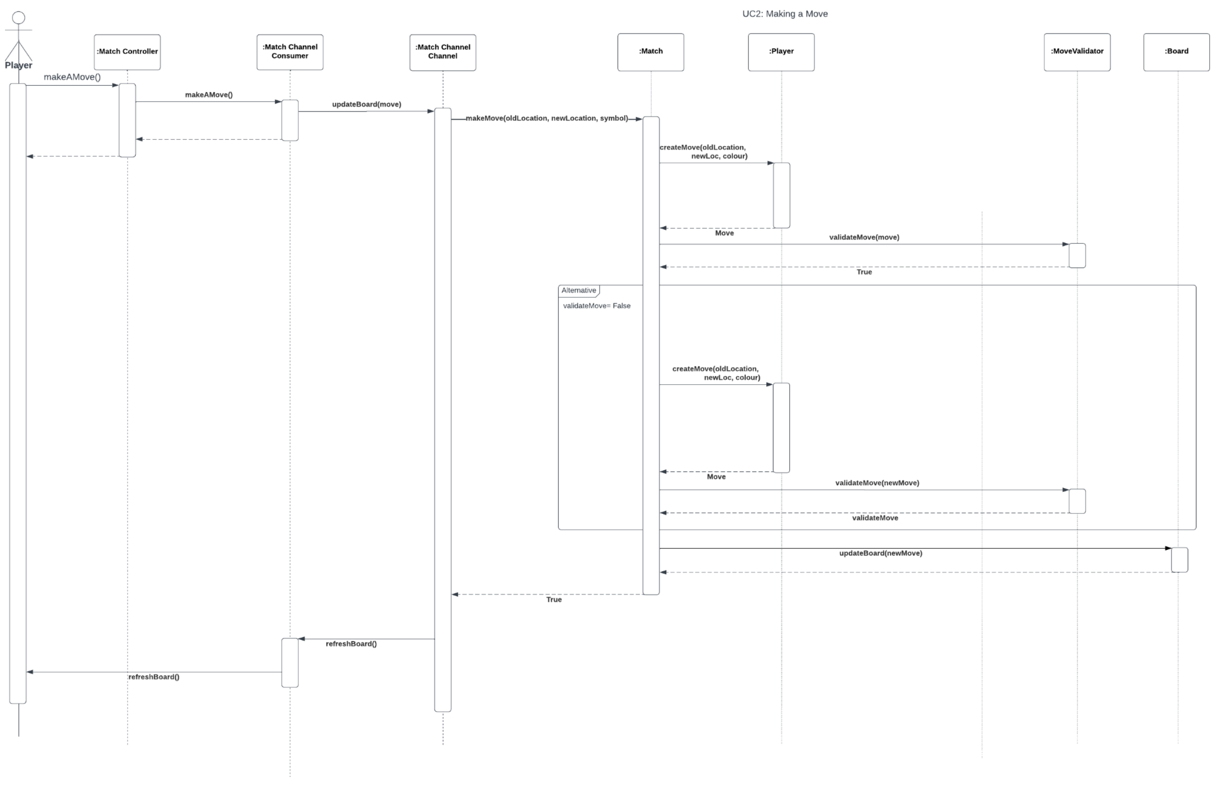
Game Subsystem
Board Region
Location
+ getNeighbours(): neighbouringLocations Hashmap<Direction, Location>
+ getStone(): Stone
+ removeStone(): Stone
+ placeStone(stone: Stone)
Stone
+ getStoneColour(): stoneColour: symbol //:white or :black
1
0..1
2
1..*
1
Board
+ updateBoard(move: Move): self
+ getNumBlackStones(): numStones: int
+ getNumWhiteStones(): numStones: int
+ showBoard()
+ getPossibleCapture(color: symbol): bool
+ getStoneLine(location: Location, dir: symbol): line: List<Location>
+ getLocation(row: int, col: int): location: Location
MoveValidator
+ validateMove (gameBoard: Board, move: Move): bool
+ hasNextMove(move: Move): bool
+ clearMoves()
Move
+ getStoneLocation1(): location1: Location
+ getStoneLocation2(): location2: Location
+ getColour(): playerColour: symbol
0..*
< validates
1
0..*
Creates >
1
Match
+ setWhitePlayer(player_id: string)
+ setBlackPlayer(player_id: string)
+ startMatch()
+ forfeitMatch(player_id: string)
+ makeMove(location1, location2: Location, color: symbol): bool
+ showCurrentBoardState(): board: Board
+ currentColourTurn(): color: symbol //white or black
1
1
Updates >
Player
+ createMove(location1, location2: Location): newMove: Move
+ setColour(colour: string)
+ getUserId(): userId: string
Plays >
0..*
1
1
2

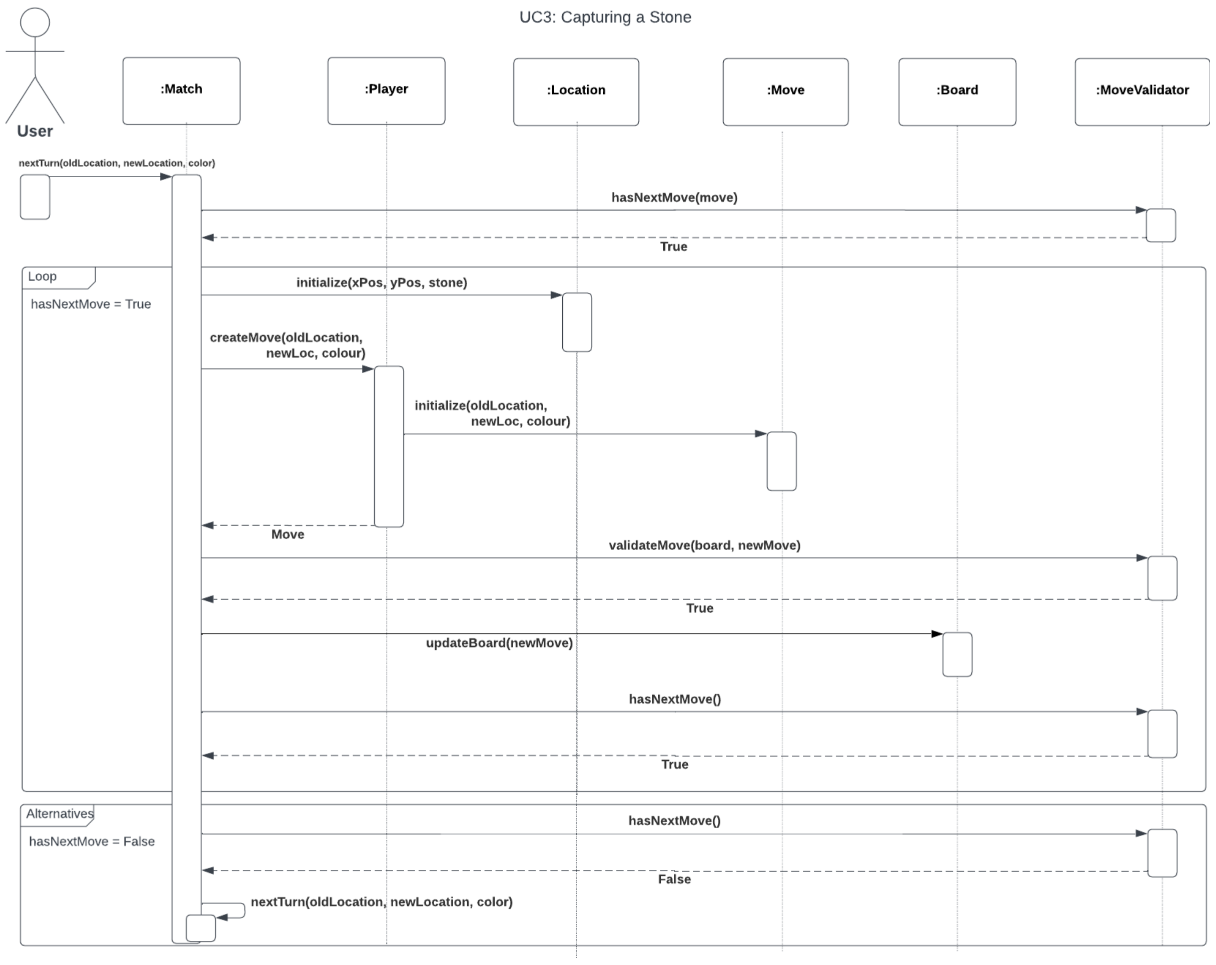
Application Sequence Diagrams

UC1: Starting a Match

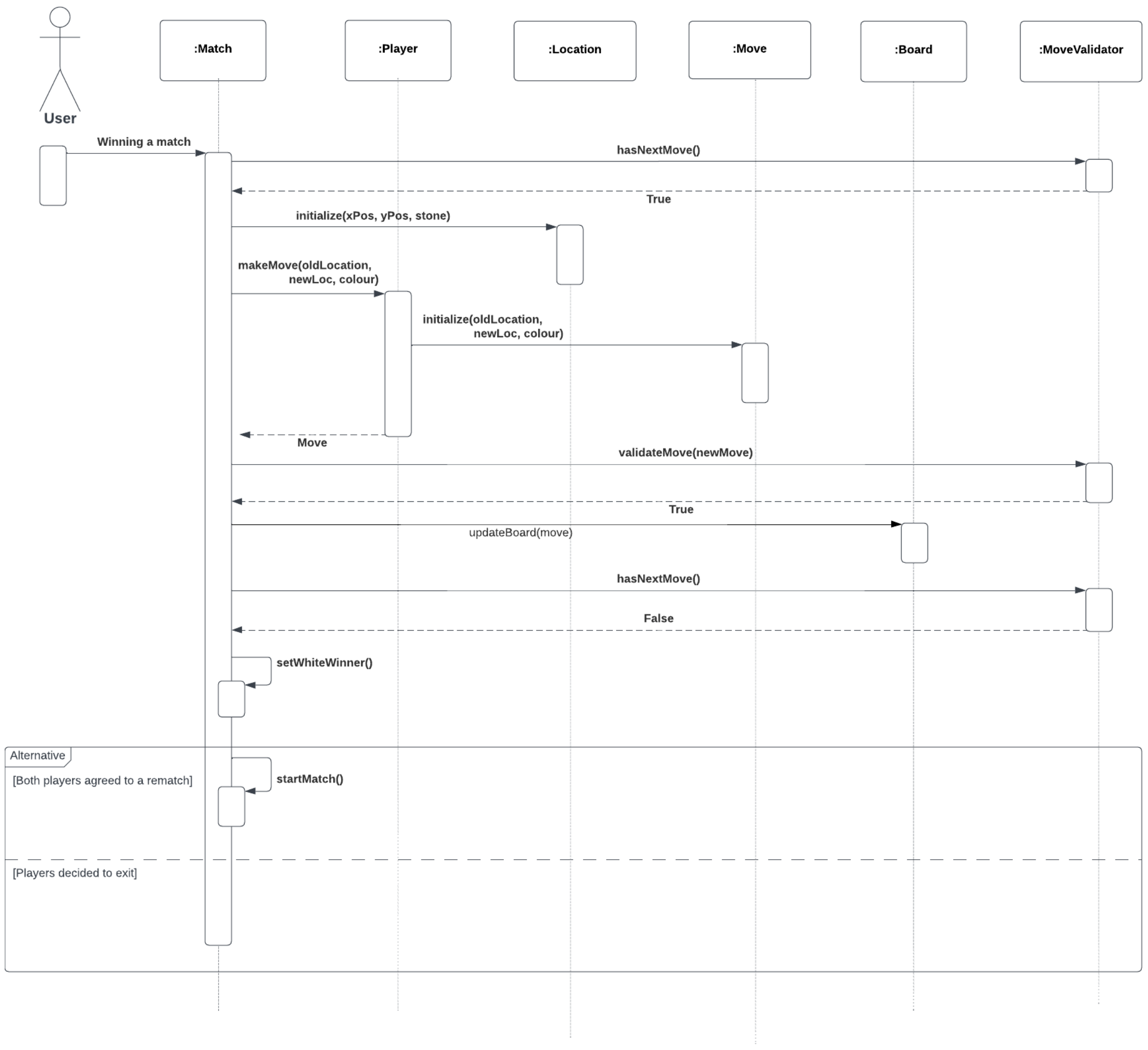



UC2: Making a Move
User
:Match
:Player
:Location
:Move
:MoveValidator
:Board
startGame()
initialize(xPos, yPos, stone)
createMove(oldLocation, newLoc, colour)
initialize(oldLocation, newLoc, colour)
Move
validateMove(move)
True
Alternative
validateMove= False
initialize(xPos, yPos, stone)
createMove(oldLocation, newLoc, colour)
initialize(oldLocation, newLoc, colour)
Move
validateMove(newMove)
False
updateBoard(newMove)
showCurrentBoardState()
Board

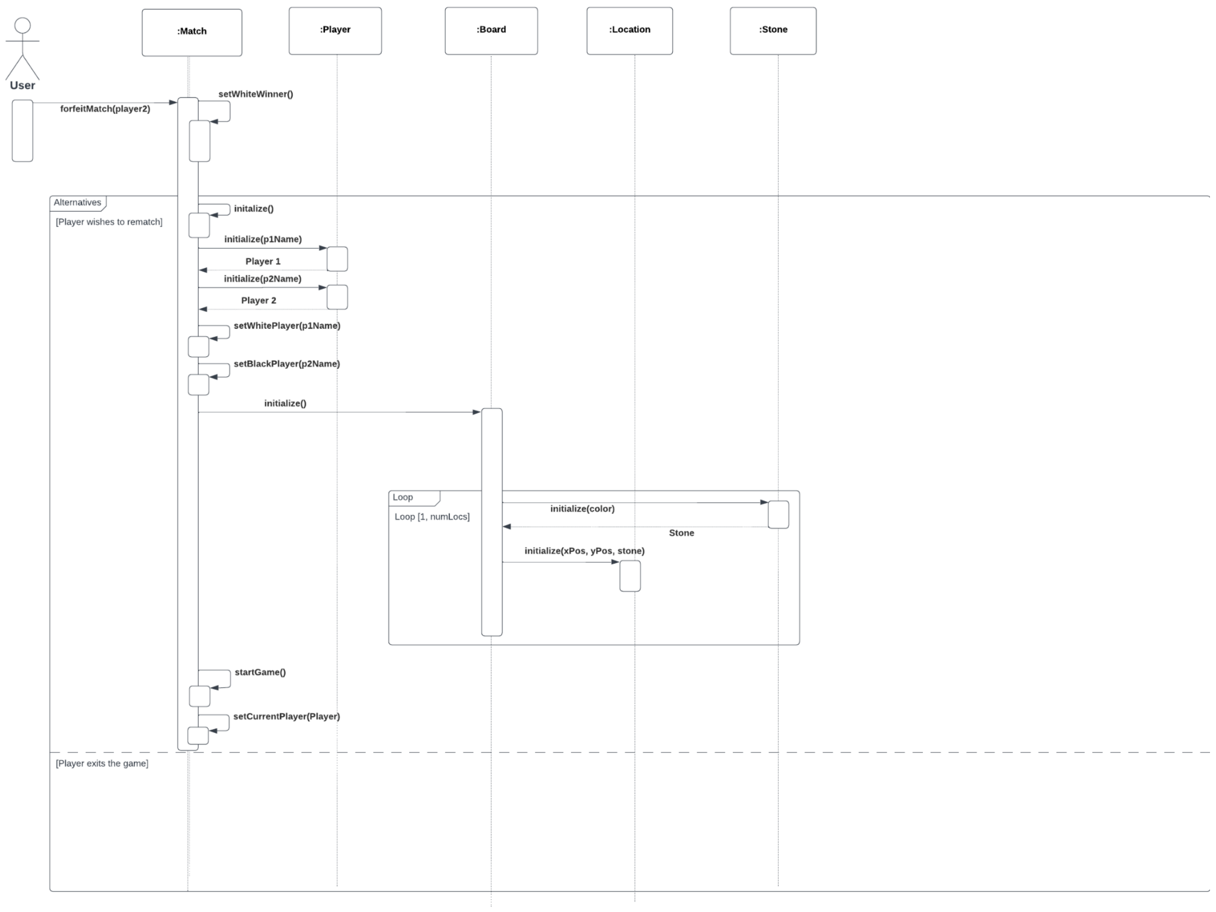

UC2: Making a Move
Player
:Match Controller
:Match Channel Consumer
:Match Channel Channel
:Match
:Player
:MoveValidator
:Board
makeAMove()
makeAMove()
updateBoard(move)
makeMove(oldLocation, newLocation, symbol)
createMove(oldLocation, newLoc, colour)
Move
validateMove(move)
True
Alternative
validateMove= False
createMove(oldLocation, newLoc, colour)
Move
validateMove(newMove)
validateMove
updateBoard(newMove)
True
refreshBoard()
refreshBoard()


UC3: Capturing a Stone
User
:Match
:Player
:Location
:Move
:Board
:MoveValidator
nextTurn(oldLocation, newLocation, color)
hasNextMove(move)
True
Loop
hasNextMove = True
initialize(xPos, yPos, stone)
createMove(oldLocation, newLoc, colour)
initialize(oldLocation, newLoc, colour)
Move
validateMove(board, newMove)
True
updateBoard(newMove)
hasNextMove()
True
Alternatives
hasNextMove = False
hasNextMove()
False
nextTurn(oldLocation, newLocation, color)

UC4: Winning a match

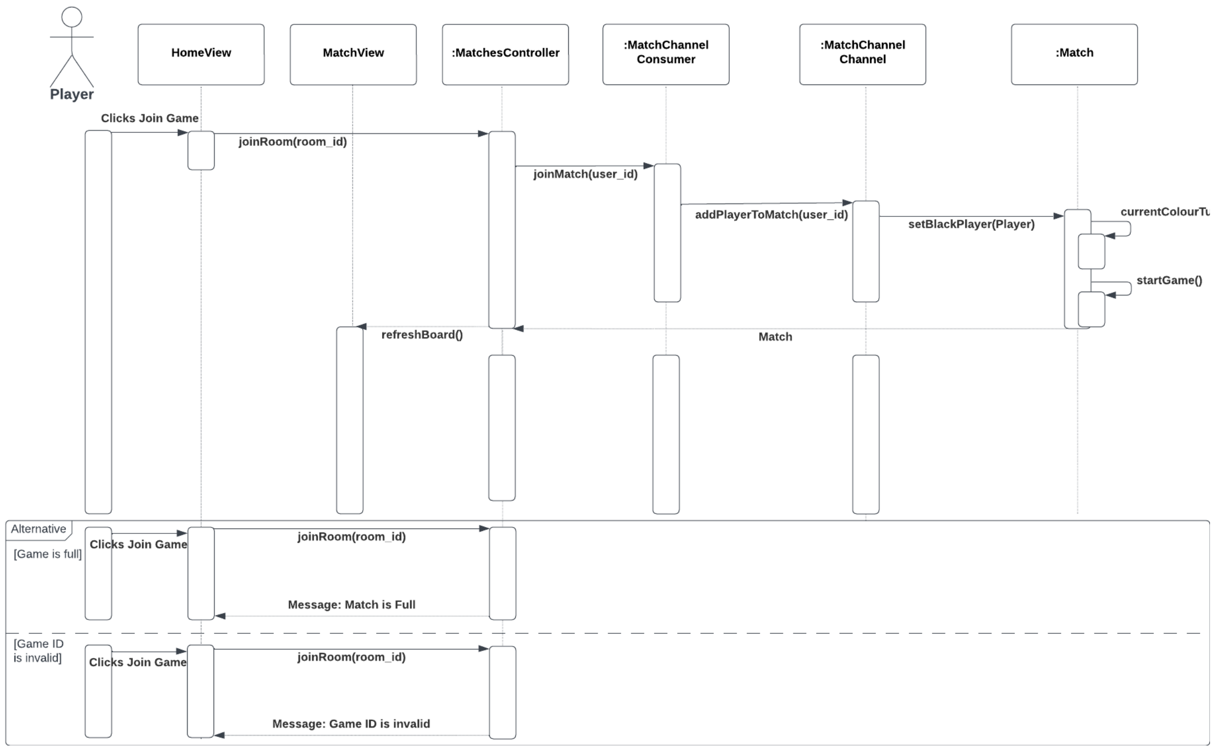


UC5: Forfeting a match

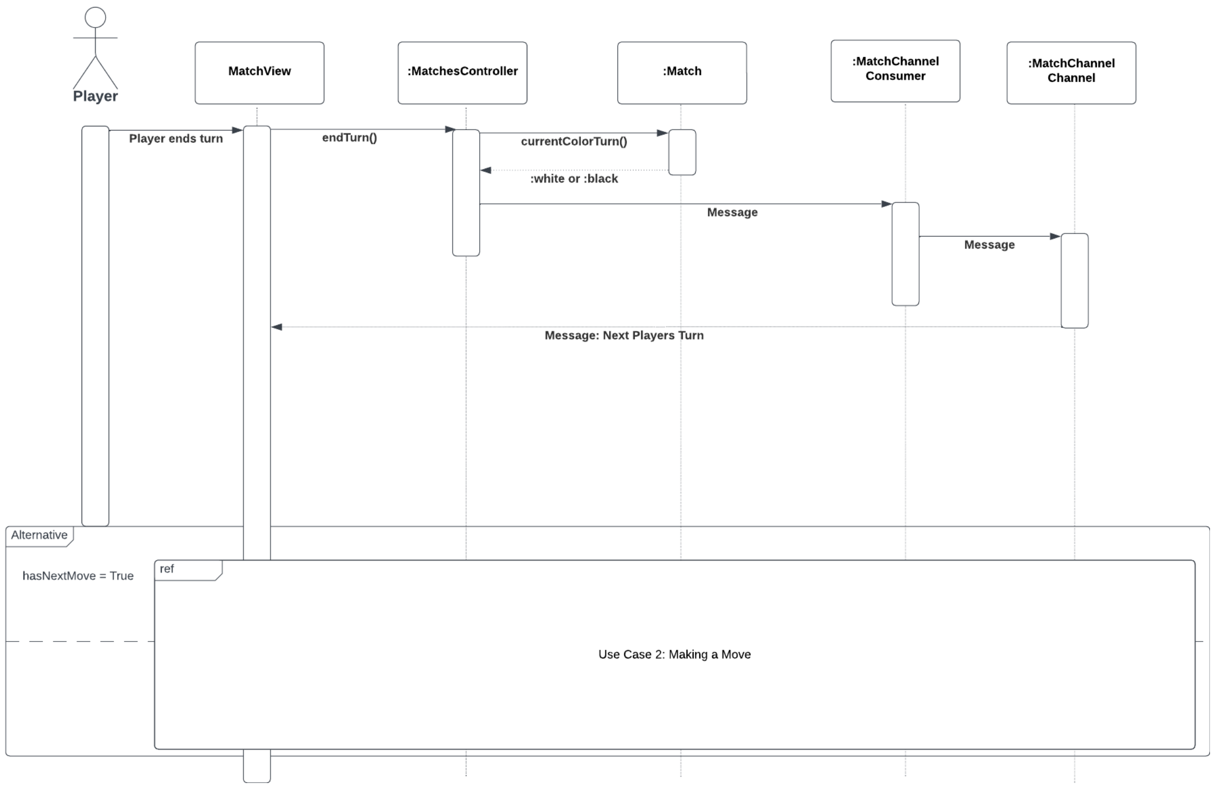


Controller Class Diagrams

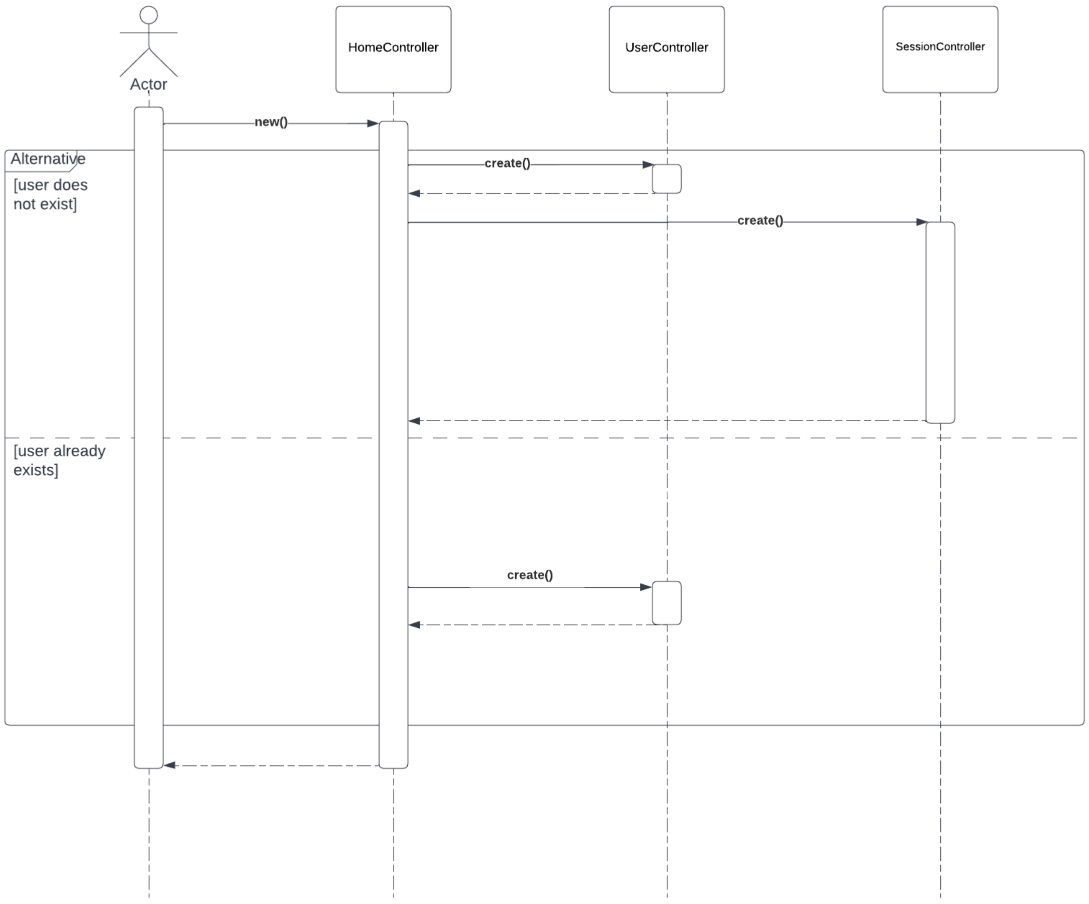
Use Case 6: Joining a Match



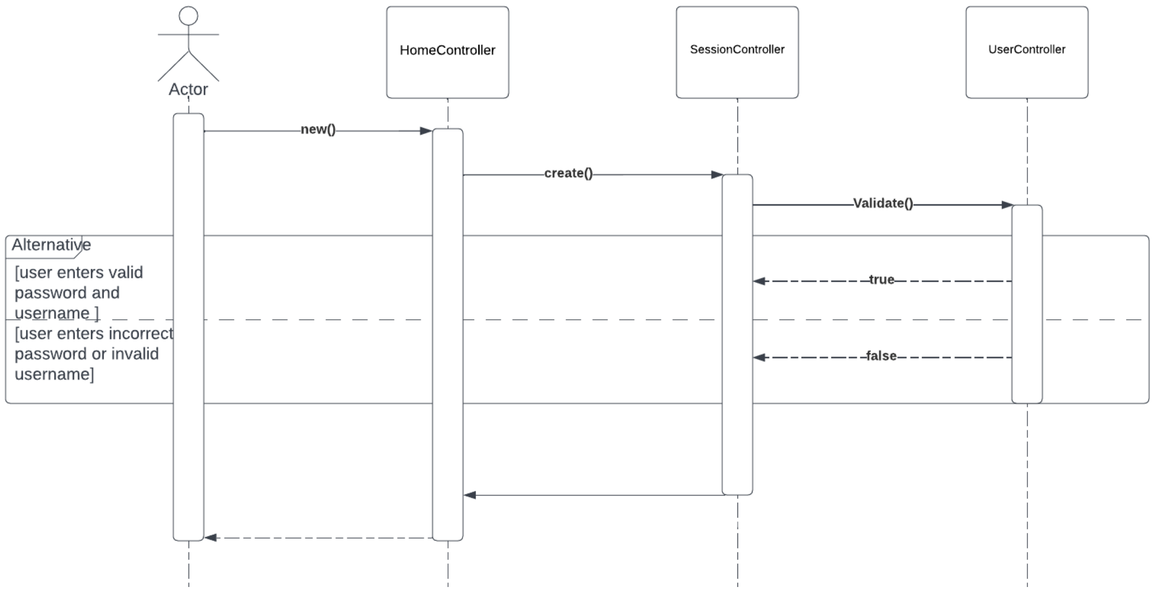
Use Case 7: Changing Players Turn

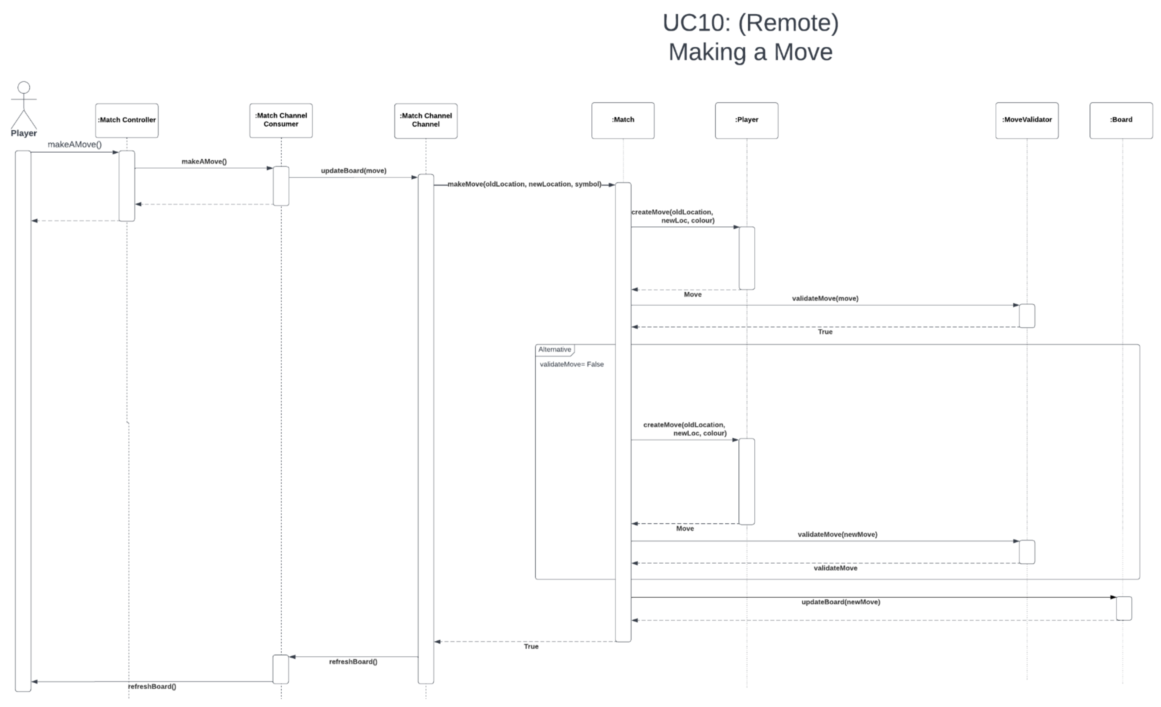


UC8: Signing Up for an Account



UC9: Signing into an Account


Actor
HomeController
SessionController
UserController
new()
create()
Validate()
Alternative
[user enters valid password and username]
true
[user enters incorrect password or invalid username]
false



Link to our sequence diagrams:
https://lucid.app/lucidchart/5d43ef9d-db11-4106-8bb0-b6e05de6316e/edit?invitationId=inv_74a82eac-a724-47a0-8af2-6e9d426af340&page=tpF_bvhJTFg7#

Application Class List

Stone

Variables

- colour: enum :white, :black

Public Methods

- initialize(colour: symbol)
    - Creates a new Stone object
    - Takes in a colour parameter and sets the instance variable colour.
    - Returns the Stone object
- getStoneColour(): colour: symbol
    - Returns the stones colour (:white or :black)

Location

Variables

- neighbouringLocations: Hashmap<Direction, Location>
- coordinates: Set(X,Y)
- stone: Stone

Public Methods

- initialize(xPos, yPos: integer, stone: Stone)
    - Create a Location object and set its coordinates with the xPos and yPos variables.
    - If a stone is provided, the stone variable is set
    - It then generates the list of neighbouring edges
        - **Note:** if for e.g, the position is 1,1 it will not have a top, top-left, and left edge, and the location for those in the neighbouringEdges hashmap would be nil.
- getNeighbours(): neighbouringLocations: Hashmap<Direction, Location>
    - Returns all of the surrounding locations of the location
- getStone(): stone: Stone
    - Return the stone to the location. Returns nil if there is no stone in this location
- removeStone(): stone: Stone
    - Removes the stone from that location and returns it.
- placeStone(stone: Stone)
    - Sets the stone variable for the location

Board

Variables

- playerOneStones: int
- playerTwoStones: int
- Locations: List<List<Location>>
- columns: int
- rows: int

Public Methods

- initialize()
  - Creates all 44 Stone objects (22 white and 22 black stones), and stores them in the locations hashmap on the game board:
    - Board has 5 rows with 9 columns in each
    - First 2 rows are black
    - Middle row is alternating black and white with the middle (5th) column blank.
    - Last 2 rows are white
    - For each stone, it calls createLocation(xPos, yPos) which returns a Location object
  - Then it sets the player<Number>Stones to 22 each
  - Can pass the locations of all Player1 and Player2 stones back to the Match.
- updateBoard(move: Move): self: Board
  - It takes the move object and replaces the contents of location1 to location2's contents on the board
  - If there was a stone in newLocation, then it decrements the appropriate player's stones (player<Number>Stones)
  - Returns self
- getNumBlackStones(): numStones: int
  - returns the number of black stones on the Board
  - Returns an integer
- getNumWhiteStones(): numStones: int
  - returns the number of white stones on the Board
  - Returns an integer
- showBoard(): board: List<List<Location>>
  - Returns the raw board data
- getPossibleCapture(colour: symbol): bool
  - Checks if the given player (by colour) has an available capture move or not
  - Checks the board 2D array for all players' stones (by colour)
    - For each stone, check neighbouring locations.
      - If there is an opponent stone in the neighbouring locations, and the space in the opposite direction is empty, return True
      - If there is an empty location, check an additional location by one space in the same direction (X or/and Y). If there is an opponent stone in that space, return True
    - Else, return False

    - Returns a bool
- getStoneLine(location: Location, dir: symbol): stoneLine: List<Location>
    - Gets the line of stones in the direction of the given location
    - Checks the board 2D array for original location, then finds the corresponding directional line of locations in the 2D array
        - If direction is up / down, finds the vertical line of locations (Y coordinates) based on the location's given X coordinate
        - If direction is left / right, finds the horizontal line of locations (X coordinates) based on the locations given Y coordinate
        - If direction is up+right or down+right, finds the diagonal line of locations (X + Y coordinates) based on the direction:
            - As Y coordinates increment by 1, X coordinates increment by 1
            - As Y coordinates decrement by 1, X coordinates decrement by 1
        - If direction is up+left or down+left, finds the diagonal line of locations (X + Y coordinates) based on the direction:
            - As Y coordinates increment by 1, X coordinates decrement by 1
            - As Y coordinates decrement by 1, X coordinates increment by 1
    - Returns a list of locations
- getLocation(row: int, col: int): location: Location
    - Returns the location object in the board 2D array with given row as X coordinate, and col as Y coordinate

Player

Variables

- name: string
- colour: string
- userId: string

Public Methods

- initialize(userId, name: string)
    - Takes in a name parameter and sets the Players object name variable and the user_id mapping.
- setColour(colour: string)
    - Sets the colour instance variable with the input string
- createMove(location1, location2: Location): newMove Move
    - creates a move object based on the parameters.
    - Returns the created move object
- getUserId(): user_id: string
    - Returns the user_id

Match

Variables

- board: Board
- whitePlayer: Player
- blackPlayer: Player
- isWhiteWinner: Bool
- isBlackWinner: Bool
- currentPlayer: Player

Public Methods

- initialize()
  - Creates a new Board object, and sets both isBlackWinner and isWhiteWinner to false.
- setWhitePlayer(userId: string)
  - Creates a player using the user id given, sets that player to white, and then assigns that player to the whitePlayer variable
- setBlackPlayer(userId: string)
  - Creates a player using the user id given, sets that player to black, and then assigns that player to the blackPlayer variable
- startMatch()
  - Ensures that all players, variables, and board is set, starts the game, and sets the current player to white using *setCurrentPlayer(whitePlayer)*
- forfeitMatch(userId: string)
  - Calls *set<colour>Winner*, based on which player is given, which will end the game
- makeMove(location1, location2: Location, color: symbol): bool
  1. Checks that the move is valid using moveValidator's *validateMove* method
  2. If the move is valid, it then updates the board using *updateBoard* from the Board class
  3. Then it checks if either the number of black or white stones are 0, if one of them are, set the respective one as the winner (and we stop processing here) using the *getNum<colour>Stones* from board
  4. Then it checks if the player can another move using moveValidator's *hasNextMove*
  5. If player does not have another move:
     - Clear all the moves in moveValidator using *clearMoves*
     - Change the current player to the other player using *setCurrentPlayer*
  6. Returns true if the method was successfully completed. False if an error occurred.
- showCurrentBoardState(): board: Board
  - Returns the board's current state by calling board's *showBoard* method
- currentColourTurn(): colour: symbol
  - Returns which colour's turn it is

Private Methods

- setWhiteWinner()
    - Change isWhiteWinner variable to True and ends the game by calling *MatchChannel.endMatch(:white)*
- setBlackWinner()
    - Change isBlackWinner variable to True and ends the game by calling *MatchChannel.endMatch(:black)*
- setCurrentPlayer(player Player)
    - Sets the variable currentPlayer to the player whose turn it currently is

Move

Variables

- oldLocation: Location
- newLocation: Location
- whichPlayer: Player

Public Methods

- initialize(location1, location2: Location, playerColour: string)
    - This is created on each player's turn and is passed the current location of their desired stone they want to move and the new location they are trying to move the stone to.
    - Sets the location1 and location2 and playerColour instance variables
- getColour(): playerColour: string
    - Returns the player’s colour assigned to that move
- getStoneLocation1(): location1: Location
    - Returns the location1 instance variable
- getStoneLocation2(): location2: Location
    - Returns the location2 instance variable

MoveValidator

Variables

- *moves*: move[]

Public Methods

- validateMove(Board *gameBoard*, Move *move*): Bool
    - Takes the board as a parameter, checks all the valid moves around that old location. If the new location is a paika, but one of the valid moves is a capture, then return false
    - Returns a Boolean if the desired move is allowed.
    - Algorithm description:
        - Calculate the *location change*; which is the difference with the *move*’s newLocation’s X and Y attributes, against oldLocation’s X and Y

attributes (newLocation X - oldLocation X, newLocation Y - oldLocation Y).

- Algorithm runs these specific checks on move and gameBoard objects:
    1. Check if stone is player's stone (colour matches)
        - Check *move*'s oldLocation X and Y attributes on the *gameBoard*, and access its colour variable. Compare the retrieved colour variable, with the Move object's colour. If they are the same, continue. Else, return false.
    2. Check if capture moves are available;
        - Calls getPossibleCapture() from Board object.
        - If getPossibleCapture() returns true, and *move* is a paika (determined by *location change* difference of 1 on either X or Y), return false. Else, continue.
    3. Check if new location is valid (# of difference in space is <= 2 spaces, 0> spaces)
        - If the *location change* difference is either (0,2), (2,0), (1,0), (0,1), (2,2), (1,1), then continue. Else, return false.
    4. Check if new location is populated by another stone
        - Check Move object's newLocation X and Y attributes on the Board object. If Stone is not retrieved, continue. Else, return false.
    5. Check if stone-to-capture is capturable
        - Calls getStoneLine() from Board object given the newLocation and the direction.
        - If there are opponent stones adjacent to the new location, return true
        - If there are opponent stones one space away from the new location, return true
        - Else return false
        - Old content
    6. Check *moves* list for subsequent moves
        - Use moves list in MoveValidator to check for:
            1. stone has to be the same
                - Get last appended Move in *moves* list. Compare the newLocation of the last appended Move, and the oldLocation of *move*. If they are the same, continue. Else, return false.
            2. Cannot return to same space twice
                - Create a hash of all Location objects from moves list (using the X and Y coordinates as the hash). Check newLocation of *move* in the hash. If it doesn't exist, continue. Else, return false.

                3. Can’t move in same direction twice in a row
                    - Get last appended Move in *moves* list. Compare the difference between the last appended Move’s oldLocation X and Y subtracted to its newLocation X and Y, and *move*’s oldLocation X and Y subtracted to its newLocation X and Y. If they are not the same, continue. Else, return false.
        - Copy *move* into *moves* variable in case of subsequent moves
- hasNextMove(Move *move*)
    - Determines if this is the first move of the game which means they do not have a next move to make.
    - If it's not the first move of the game, it determines if the player can make a subsequent Approach or Withdraw attack on the same turn.
    - Algorithm description
        - Calculate the *location change*; which is the difference with the *move*’s newLocation’s X and Y attributes, against oldLocation’s X and Y attributes (newLocation X - oldLocation X, newLocation Y - oldLocation Y).
        - Check *move* object radius for other potential captures. If *move* object can complete another subsequent capture that does not follow the same direction as the move that was just conducted (indicated by the *location change*), then return true. Else, return false
- clearMoves()
    - Clears the Moves list stored by MoveValidator to check previous subsequent moves

Remote Use Cases

UC6: **Joining a Match**

- Primary Actor: Player
- Goal: Both the Player and Opponent are in the same remote Fanorona match synchronously.
- Stakeholders List:
    - Match Manager: Checks if the Match ID the Player provides is valid, responsible for assigning the colour to the Player and Opponent, and also creates the board.
    - Player: The person with white stones.
    - Opponent: The person with black stones.
- Initiating Event: The Player has a Match ID.
- Main success Scenario:
    1. The Player provides a Match ID to the Match Manager.
    2. Match Manager checks to see if the Match ID is valid.

3. Match Manager places the Player into the match and assigns the Player with white colour stones.
4. The Player provides a Match ID to the Match Manager.
5. Match Manager places the Opponent into the match and assigns the Opponent with black colour stones.
6. Match Manager creates and displays the board to both players.

- ➢ Pre Conditions: The Match Manager creates the room for the match.
- ➢ Post Conditions: The Player and Opponent are in the same remote Fanorona game together.
- ➢ Alternate Flows or Exceptions:
  - 1a. Match ID is invalid:
    - 1a. 1. The Match Manager informs the Player that the Match ID is invalid.
    - 1a. 2. Player gives the Match Manager a new Match ID.
  - 2a. The game is full:
    - 2a. 1. The Match Manager does not place the Player into a match.
    - 2a. 2. The Match Manager informs the Player that the game for the Match ID is already full.
- ➢ Use cases Utilized:
  - None
- ➢ Scenario Notes:
  - None

UC7: **Changing a Turn**

- ➢ Primary Actor: Player
- ➢ Goal: The Match Manager informs the Player about their turn.
- ➢ Stakeholders List:
  - Match Manager: Notifies the Player and Opponent when their turn is over and transfers control of the board to the opposition.
  - Player: The person with white stones.
  - Opponent: The person with black stones.
- ➢ Initiating Event: Opponent finishes their move.
- ➢ Main Success Scenario:
  1. Match Manager indicates to the Opponent that their turn ends.
  2. Match Manager removes the Opponent’s ability to move a stone.
  3. The Match Manager informs the Player it is their turn.
  4. Player begins their turn.
- ➢ Pre Conditions: Opponent finishes their turn.
- ➢ Post Conditions: None
- ➢ Alternate Flows or Exceptions:
  - 1a. Making another move:
    - 1a. 1. Match Manager indicates the Opponent can make another move.
    - 1a. 2. Perform UC2: **Making a move**
- ➢ Use cases Utilized:
  - UC2: **Making a move**
- ➢ Scenario Notes:

- None

UC8: **Signing Up for an Account**

- ➢ Primary Actor: Player
- ➢ Goal: Player successfully creates an account.
- ➢ Stakeholders List:
    - Player: Player creates an account.
    - User Manager: Checks for storage and uniqueness in a Player’s data.
    - Account Authenticator: Account Authenticator verifies the user’s account data.
- ➢ Initiating Event: None
- ➢ Main Success Scenario:
    1. Player provides their account information to the Account Authenticator.
    2. The User Manager checks to see if the data is unique.
    3. The Account Authenticator stores the Players data.
    4. User Manager assigns 0 points to the Players account.
- ➢ Pre Conditions: None
- ➢ Post Conditions: Player successfully creates an account.
- ➢ Alternate Flows or Exceptions:

  2a. Duplicate data:

    2a. 1. The User Manager informs the Player the data is already in use.

    2a. 2. The account creation fails.
- ➢ Use cases Utilized:
    - ○ None
- ➢ Scenario Notes:
    - ○ None

UC9: **Signing into an Account**

- ➢ Primary Actor: Player
- ➢ Goal: The Player successfully signs into their account.
- ➢ Stakeholders List:
    - Player: The person who is signing into their account.
    - Account Authenticator: Authenticates the Players sign-in data.
    - User Manager: Checks for storage and uniqueness in a Player’s data.
- ➢ Initiating Event: The Player is online on the game website.
- ➢ Main Success Scenario:
    1. The Player asks the User Manager to gain access into their account.
    2. The Account Authenticator asks the Player for their account data.
    3. The Player provides their account data to the Account Authenticator..
    4. The Account Authenticator validates the information from the Player.
- ➢ Pre Conditions: The Player signs up for an account.
- ➢ Post Conditions: The Player successfully signs into their account.
- ➢ Alternate Flows or Exceptions:
    - ○ 4a. Account not found.

4a. 1. The Account Authenticator informs the player that they must sign up for an account.
4a. 2. Perform UC8: **Signing Up for an Account.**

- 4a. Incorrect data:

4b. 1. The Account Authenticator informs the Player their data is incorrect.
4b. 3. Perform UC9: **Signing into an Account**

➢ Use Cases Utilised:
- UC8: **Signing Up for an Account**

➢ Scenario Notes:
- None

UC10: **Making a move (Remote)**

➢ Primary Actor: Player

➢ Goal: Player wants to move a stone.

➢ Stakeholders List:
- Player: The person making a move.
- Move Validator: Validates that the move the Player is making is valid.
- Match Controller: Responsible for manipulating the current Match as the player interacts with the board.
- Match Channel: Used to send information about the Player and the Opponent's interactions with the board, between the Match Controller and the Match.
- Opponent: The opposition in the Fanorona match.
- Match: The current game of Fanarona that is being played between the player and he opponent.

➢ Initiating Event: Player's turn.

➢ Main success Scenario:
1. The Player informs the Match Controller the stone they wish to move.
2. The Player informs the Match Controller which location they want the stone to move to.
3. The Match Controller sends the information about the stone being moved and the location for the stone to be relocated to the Match Channel.
4. The Match Channel sends the stone being moved and the location for the stone to be relocated to the Match.
5. The Move Validator validates that the selected stone can move to the Player's selected location.
6. The Move Validator moves the stone to the location for a paika.
7. The Match Channel sends the updated board to the Player.

➢ Pre Conditions:
- The Opponent finishes making their turn,
- OR: The Player is making another move in the same turn,
- OR: It is the first turn of the game.

➢ Post Conditions:
- The Player moves a stone to a selected location.

➢ Alternate Flows or Exceptions:

6a. Capturing an Opponent's stone(s):

5a. 1. Perform UC3: **Capturing a stone**

5a. Winning a Game:

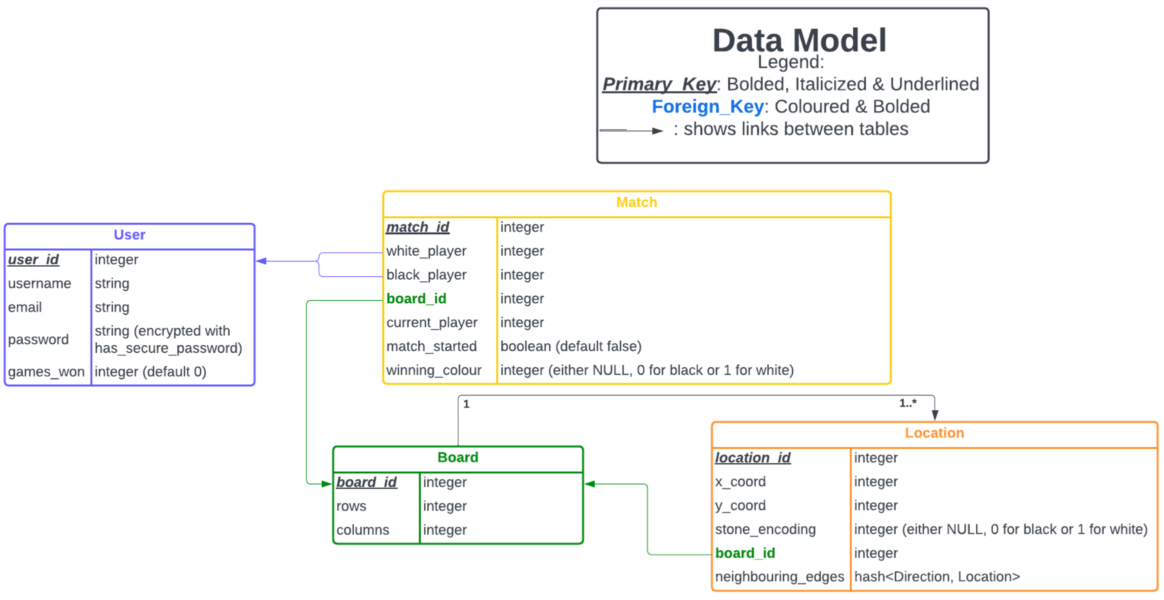
4a. 1. There is only one move left for the Player to win the game.

4a. 2. Perform UC4: **Winning a match**

➢ Use cases Utilized:
- UC3: **Capturing a stone**
- UC4: **Winning a match**

➢ Scenario Notes:
- None

Data Model



Data Model
Legend:
Primary_Key: Bolded, Italicized & Underlined
Foreign_Key: Coloured & Bolded
: shows links between tables
User
user_id integer
username string
email string
password string (encrypted with has_secure_password)
games_won integer (default 0)
Match
match_id integer
white_player integer
black_player integer
board_id integer
current_player integer
match_started boolean (default false)
winning_colour integer (either NULL, 0 for black or 1 for white)
1
1..*
Board
board_id integer
rows integer
columns integer
Location
location_id integer
x_coord integer
y_coord integer
stone_encoding integer (either NULL, 0 for black or 1 for white)
board_id integer
neighbouring_edges hash<Direction, Location>

List of URL Calls

Basic authentication URL calls (Users + Sessions + Home)

root home#new

get ‘home’, to: ‘home#show’, as: ‘show’

resources :users

resources :sessions, only [:create, :destroy]

get 'sign_up', to: 'users#new', as: 'sign_up'

post 'login', to: 'sessions#create', as: 'login'

get 'log_out', to: 'sessions#destroy', as: 'log_out'

Match URL calls

resources :match

post ‘create’, to ‘match#new’, as: ‘create’

get ‘find’, to: match#find, as: ‘find’

get ‘waiting_for_player’, to: ‘match#waiting_for_player’, as: ‘match’

post ‘start_match’, to: match#start, as ‘start’

post ‘make_move’, to: match#make_move, as ‘make_move’

get ‘refresh’, to: match#refresh_board, as ‘refresh’

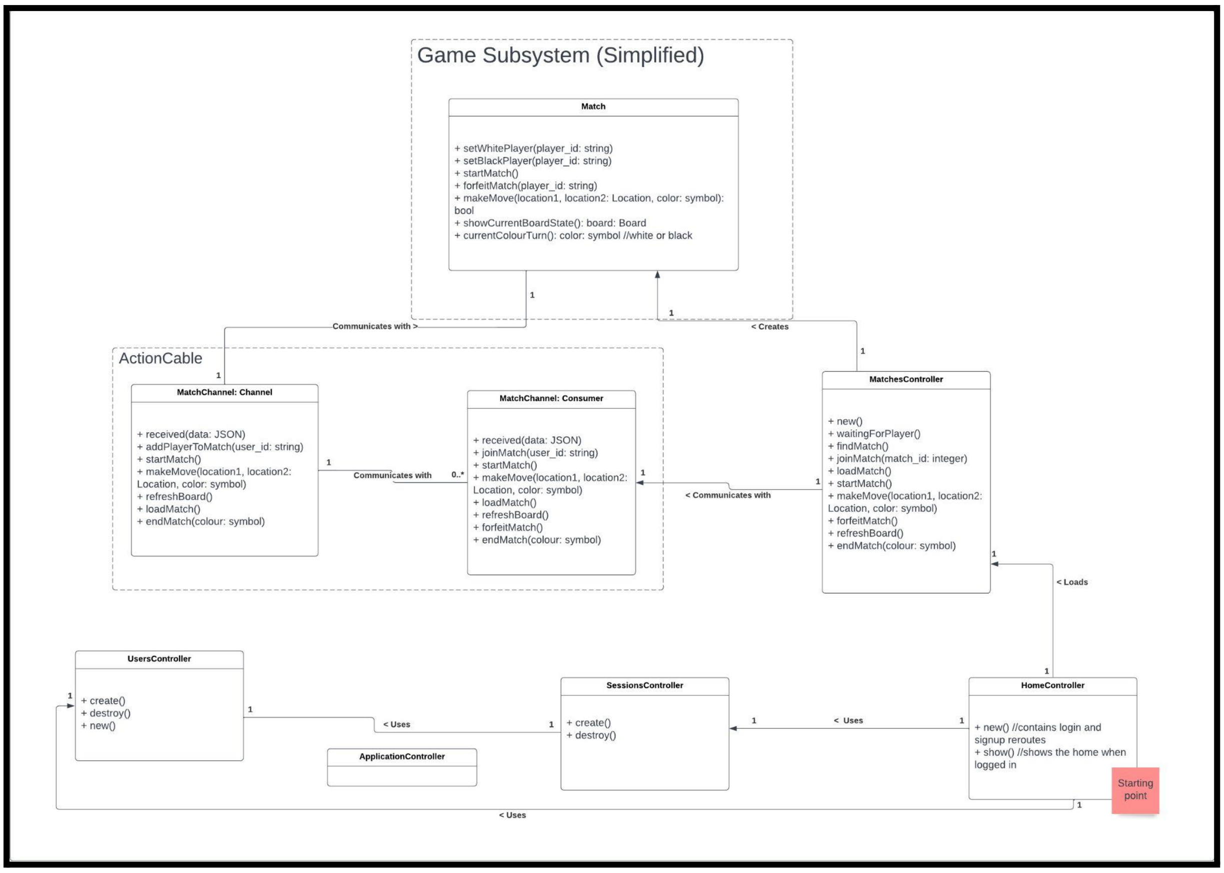
post ‘end_match’, to ‘match#end’, as: ‘end’

post ‘join’, to: match#join, as: ‘join’

post ‘forfeit_match’, to ‘match#forfeit’, as: ‘forfeit’

get ‘load_match’, to ‘match#load_match’, as ‘load_match’

Controller Class Diagram


Game Subsystem (Simplified)
Match
+ setWhitePlayer(player_id: string)
+ setBlackPlayer(player_id: string)
+ startMatch()
+ forfeitMatch(player_id: string)
+ makeMove(location1, location2: Location, color: symbol): bool
+ showCurrentBoardState(): board: Board
+ currentColourTurn(): color: symbol //white or black
1
Communicates with >
1
< Creates
1
ActionCable
1
MatchChannel: Channel
+ received(data: JSON)
+ addPlayerToMatch(user_id: string)
+ startMatch()
+ makeMove(location1, location2: Location, color: symbol)
+ refreshBoard()
+ loadMatch()
+ endMatch(colour: symbol)
1
Communicates with
0..*
MatchChannel: Consumer
+ received(data: JSON)
+ joinMatch(user_id: string)
+ startMatch()
+ makeMove(location1, location2: Location, color: symbol)
+ loadMatch()
+ refreshBoard()
+ forfeitMatch()
+ endMatch(colour: symbol)
1
< Communicates with
1
MatchesController
+ new()
+ waitingForPlayer()
+ findMatch()
+ joinMatch(match_id: integer)
+ loadMatch()
+ startMatch()
+ makeMove(location1, location2: Location, color: symbol)
+ forfeitMatch()
+ refreshBoard()
+ endMatch(colour: symbol)
1
< Loads
1
UsersController
+ create()
+ destroy()
+ new()
1
1
< Uses
1
ApplicationController
SessionsController
+ create()
+ destroy()
1
< Uses
1
HomeController
+ new() //contains login and signup reroutes
+ show() //shows the home when logged in
Starting point
1
< Uses

Controller Classes Details

UsersController

Name	**Request Type**	**Description**	**Returns/Redirects**
create()	POST /create *user#create*	Initializes a new User with the data provided by the user	After saving the user, returns to the *home#show* view if successful (account_page)
new()	GET /new *user#new*	Displays the *user#new* page and allows the User to create a new account.	Retrieves the *user#new* view (sign_up_page)
destroy()	DELETE /delete	Allows the User to delete	Redirect to *home#new*

	user#destroy	their own account from the Users table. Gets their user_id from the Sessions object used in Rails.	(sign_in_page) upon successful delete.

SessionsController

Name	**Request Type**	**Description**	**Returns/Redirects**
create()	POST /create *session#create*	Called by HomeController and creates a session when a user clicks 'LogIn' by finding the user through User.find_by_email(params[:email]), saving the user_id in the session object. If method returns false, flash the error to the user	After session is created, redirect back to *home#show* view (account_page)
destroy()	DELETE /destroy *session#destroy*	Destroys a Session when a User logs out by removing the user_id in the session object.	After session is destroyed, redirect back to *home#new* view (signin_page)

Note: Session information is managed by rails and not explicitly in this design. Therefore there is no session present in our data model. The Session controller is used for creating and destroying sessions. Please refer to the Action Controller documentation in Rails for more information.
https://guides.rubyonrails.org/action_controller_overview.html#session

HomeController

Name	**Request Type**	**Description**	**Returns/Redirects**
new()	GET /new *home#new*	Displays the page to allow a User to sign up or login for an account	Returns the *home#new* View (sign_in_page)
show()	GET /show *home#show*	After User has created an account or logged in, show their account details, the join game option and create new game	Returns the *home#show* view (account_page)

MatchesController

Name	**Request Type**	**Description**	**Returns/Redirects**
new()	POST /create *match#new*	Creates a new Matches object for a new game of Fanorona and stores the match_id in the action	Calls MatchChannel(Consumer).joinMatch to add the player to the match with the

		cable classes.	appropriate parameters. Returns the *created_match_Page.*
waitingForPlayer (room_id: integer)	GET /waiting_for_player *match#waiting_for_player*	Is invoked when MatchesController.new() is done processing and displays the match_id for another player to join	Returns the *match#waitingForPlayer* view (created_match_page)
findMatch()	GET /find *match#find*	Gets the page that allows a user to enter the match id. Is called from the home page when logged in	Returns the *match#find* view (join_match_page)
joinMatch(match_id)	POST /join *match#join*	Checks if the match_id is a valid ID and black_player is not set yet. Place the User into that game otherwise the code is invalid or the match_id doesn't exist yet.	If match_id is valid, call MatchChannel(Consumer).joinMatch to add the player to the match with the appropriate parameters. Else, display appropriate error message (match full or match doesn't exist) Returns *join_match_page*
loadMatch()	GET /load *match#load_match*	Is invoked once both Players have joined the same match_id and both colors in the Match object are set.This enables the first Player who joined the matc_id to click the start match button. The second Player receives a "waiting for the other player to start the game" message.	Returns the *match#load_match* view.
startMatch()	GET /start *match#start*	Once 2 players have joined the same match_id, the first player who joined the game is able to start the match. This invokes the startMatch() in the MatchChannel: Consumer.	Returns the *board_page* view once the player clicks on Start Game.
makeMove(oldLocation, newLocation, color)	POST /make_move *match#make_move*	After the User has selected the location of the piece they want to move, and the location they want to move to the	Returns True and calls the refreshBoard() method which returns *match#refresh_board*. Else Returns False and Player is

		piece to (from Page *board_page*), pass parameters to makeMove(oldLocation, newLocation, color) in the MatchChannel: Consumer class.	notified of an invalid move.
forfeitMatch()	POST /forfeit *match#forfeit*	Calls setBlackWinner() or setWhiteWinner() on the Match object which then calls forfeitMatch(player_id) on the Match object which invokes the MatchChannel: Channel's forfeit method.	Returns *end_game_Page* view to all Players subscribed to the MatchChannel: Channel.
refreshBoard()	GET /refresh_board *match#refresh_board*	Updates the front-end view after a Player has made a move. After the game board table is updated, refresh the view based on the changes that are made from pushing the changes from the model to the front-end view to be displayed to the user by calling the Match.showCurrentBoardState method.	Returns the updated *match#refresh_board* view (board_page)
endMatch()	GET /end *match#end*	Once a match has ended (from calling forfeit or from someone making a winning move), we call this action to display the end match view and show which color won	Returns the *match#end* view (end_game_page)

ActionCable Classes

MatchChannel: Consumer (Client Side Component)

Name	**Description**	**Calls**
received(data)	Receives the data from the MatchChannel Channel and	This method will call the appropriate function in the

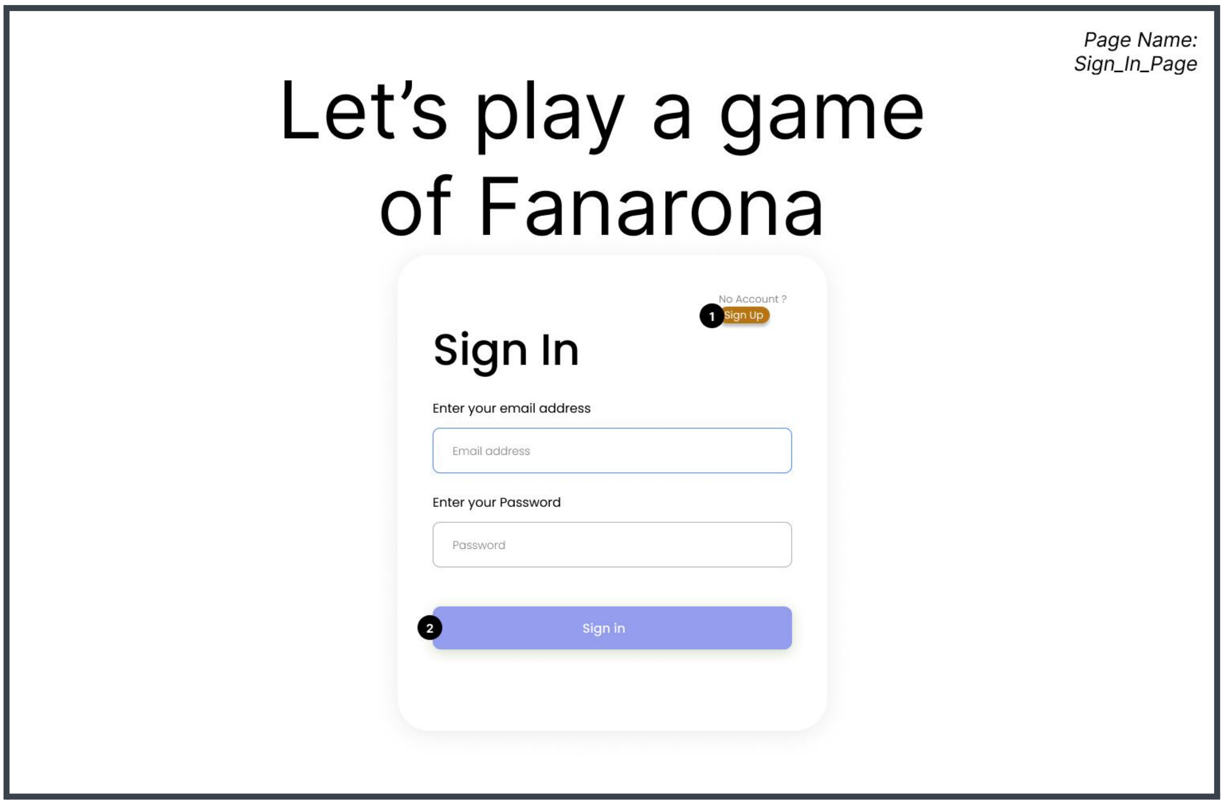
	parses the data parameters to know which method to call (i.e “data = loadMatch(), parameters=[“123”]”).	consumer class based on the parsed data it receives.
joinMatch(user_id: String)	Called when a user is joining a match (either when creating a match through Match.new or Match.joinMatch) and packages the user_id into a JSON which will be sent to the server side function	Sends a message to the MatchChannel.addPlayerToMatch (Channel) method (which will be processed through the received method first). If this user is the one who created a new match, redirect to the *match#waiting_for_player* view (created_match_page)
startMatch()	Alerts both Players that the Match has begun. Invokes the startMatch() method in the MatchChannel: Channel.	Sends a message to the MatchChannel.startMatch (Channel) method (which will be processed through the received method first)
makeMove(oldLocation, newLocation, colour)	Sends a message to the MatchChannel.makeMove (Channel) method (which will be processed through the received method first) that contains the oldLocation, newLocation and colour.	Returns True or False to the MatchController makeMove().
loadMatch()	After a Player enters a valid game_id the MatchChannel: Consumer loads the specific match.	Calls the *match#load* view from MatchesController
refreshBoard()	Will refresh the display by calling the MatchesController.refreshBoard action	Invokes the *match#refresh_board* view (board_page)
forfeitMatch()	Called by the MatchController.forfeitMatch method, and sends a message to the MatchChannel Channel to call endMatch	Sends a message to the MatchChannel.endMatch (Channel) method (which will be processed through the received method first).
endMatch(colour: Symbol)	Invoked by MatchChannel: Channel endMatch(). The Match end view is pushed to both Players (the subscribers of the channel) showing who won the game	Calls the *match#end* view (end_game_page)

MatchChannel: Channel (Server Side Component)

Name	**Description**	**Calls**
received(data)	Receives the data from the MatchChannel consumer and parses the data parameters to know which method to call and the parameters to use for that method (e.g “data = joinMatch(), parameters=[“123”]”).	This method will call the appropriate function in the channel class based on the parsed data it receives
addPlayerToMatch(user_id: String)	This consumes the received parameters (data = “joinMatch(user_id)”) and places the user into the corresponding match_id. If there is no user yet for that match (so white_player and black_player is nil), set the white_player to user_id, else set it to black_player. Once this is set, it calls the respective methods in the consumer	If black_player was set, it calls the MatchChannel.loadMatch for all the subscribers from the consumer. Else, don’t call anything (since the white player should have the waiting for player view already) Returns *match#load_match* to Player 1 if both Players are in the game and Returns *match#load_match* to Player 2 with a message saying “waiting for Player 1 to start the game”). Returns *created_match_Page* if only Player 1 is in the game.
startMatch()	If both Players are registered in the same game, startMatch() in the Match object is called. Both Players subscribed to this channel are alerted.	After calling Match.startMatch, it calls the refreshBoard method from this class.
makeMove(location1, location2: Location, colour: Symbol)	Calls makeMove() from the Matches object which determines if the move was a valid move or not.	Returns True or False. Call the refreshBoard() method from this class if True. If False, returns *board_Page* with an error message saying “invalid move”.
refreshBoard()	Message to be broadcast to all subscribers of the match to refresh the page	Calls the refreshBoard method from the client side (Consumer) for all subscribers

loadMatch()	Message to be broadcast to all subscribers of the match to now display the *match#load* view	Calls the loadMatch method from the client side (Consumer) for all subscribers
endMatch(colour :Symbol)	Message to be broadcast to all subscribers of the match to now display the *match#end* view	Calls the endMatch method from the client side (Consumer) for all subscribers with the colour parameter

Rails – Web Pages



Page Name: Sign_In_Page

Let's play a game of Fanarona

Sign In

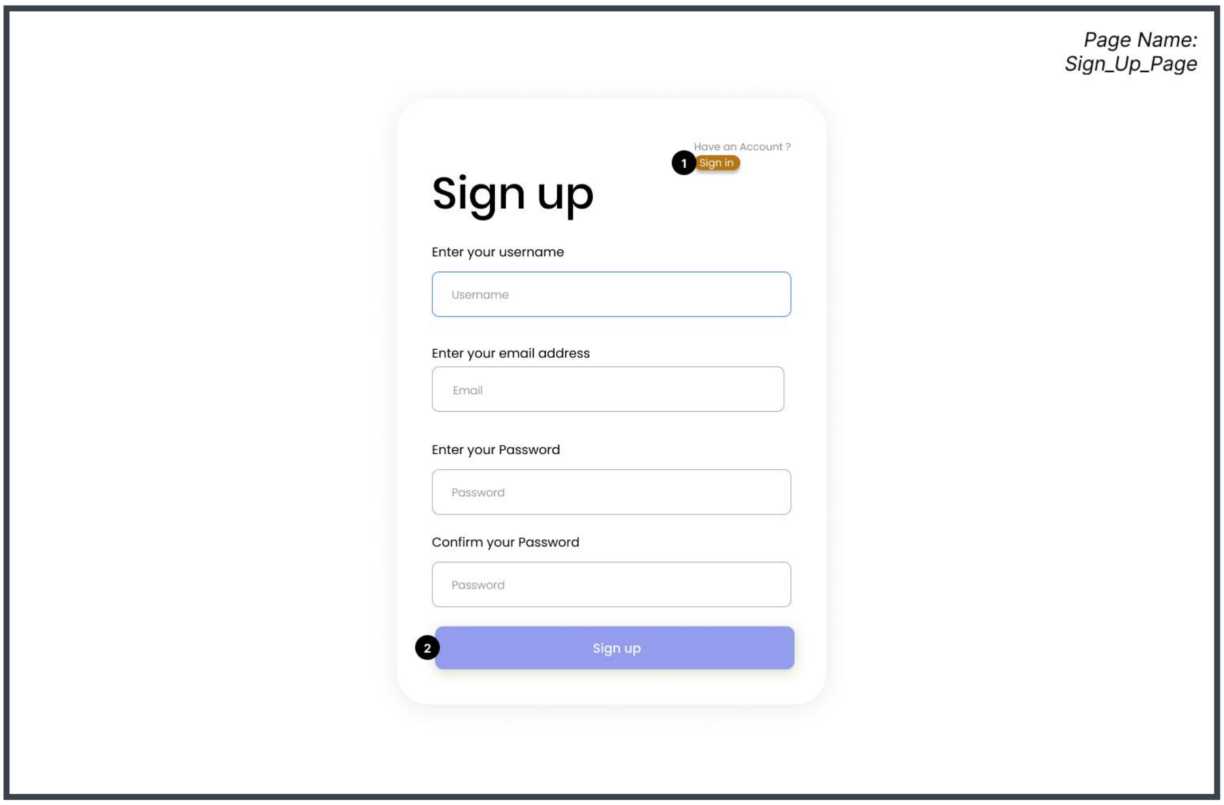
Enter your email address

Email address

Enter your Password

Password

Sign in



Page Name: Sign_Up_Page

Sign up

Enter your username

Username

Enter your email address

Email

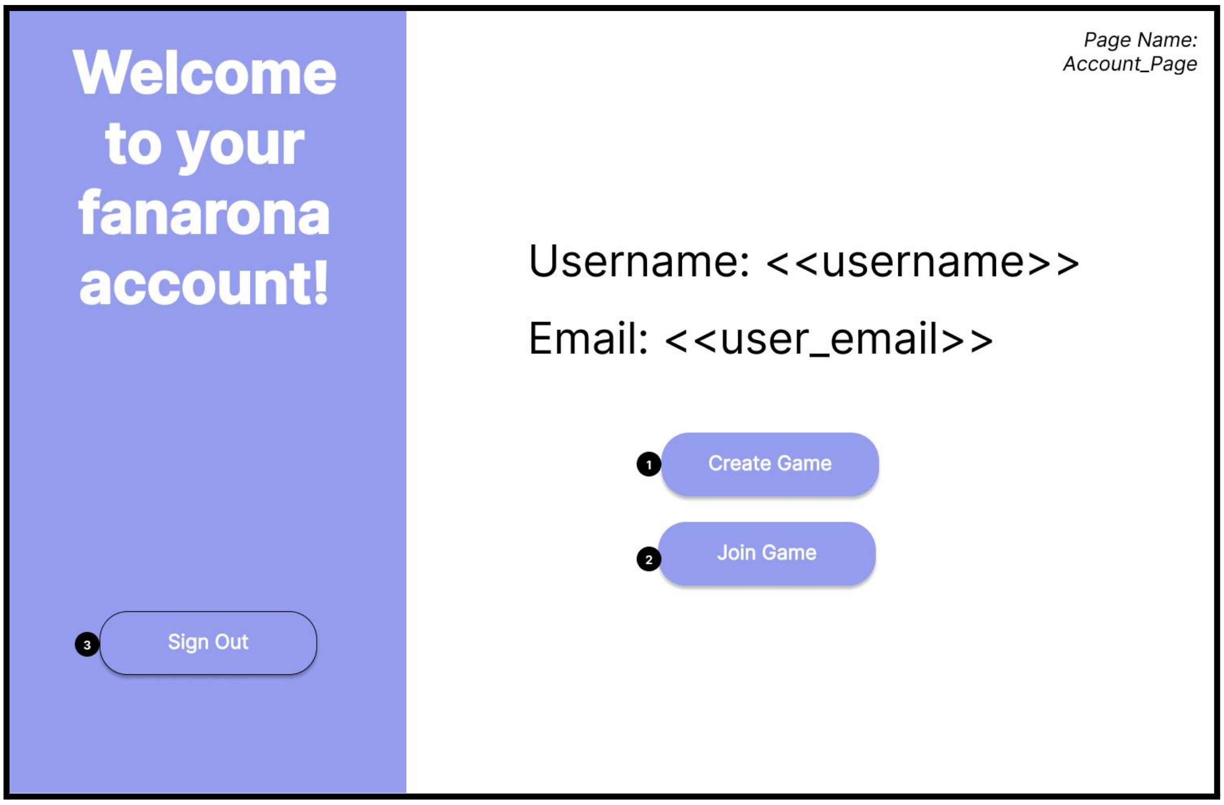
Enter your Password

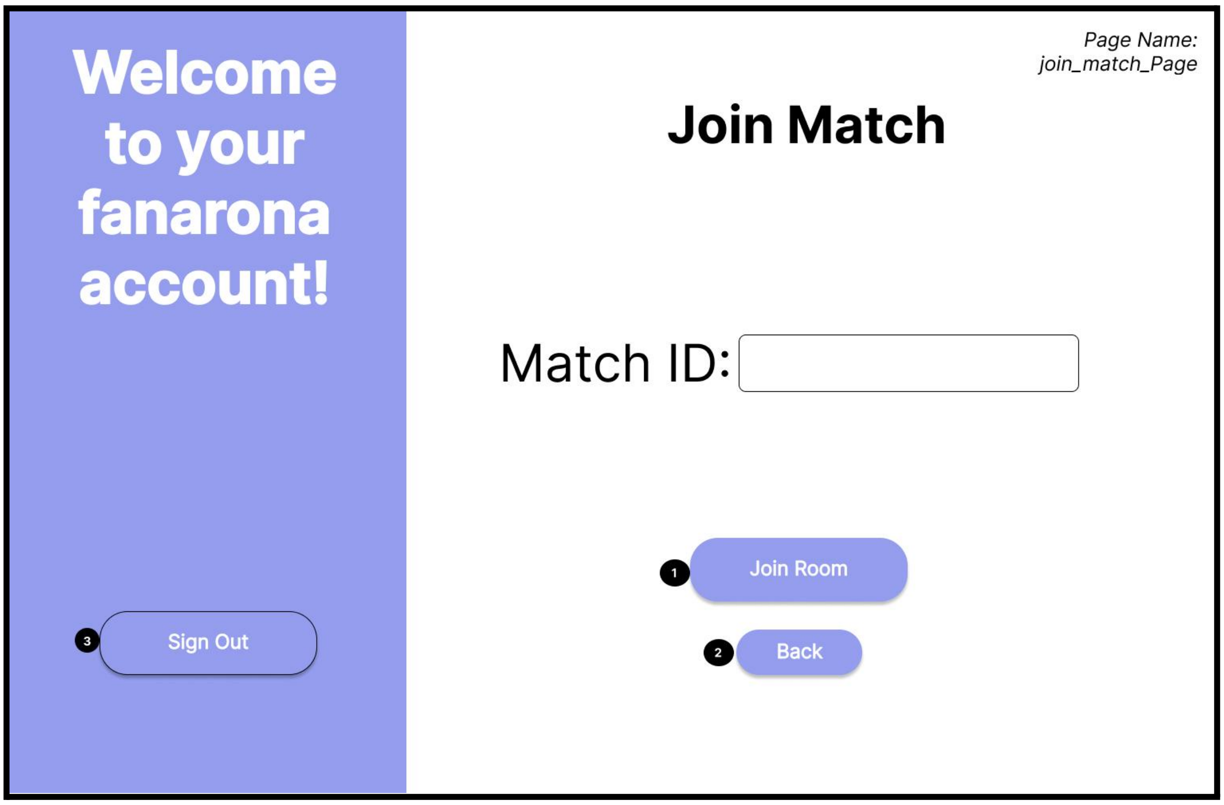
Password

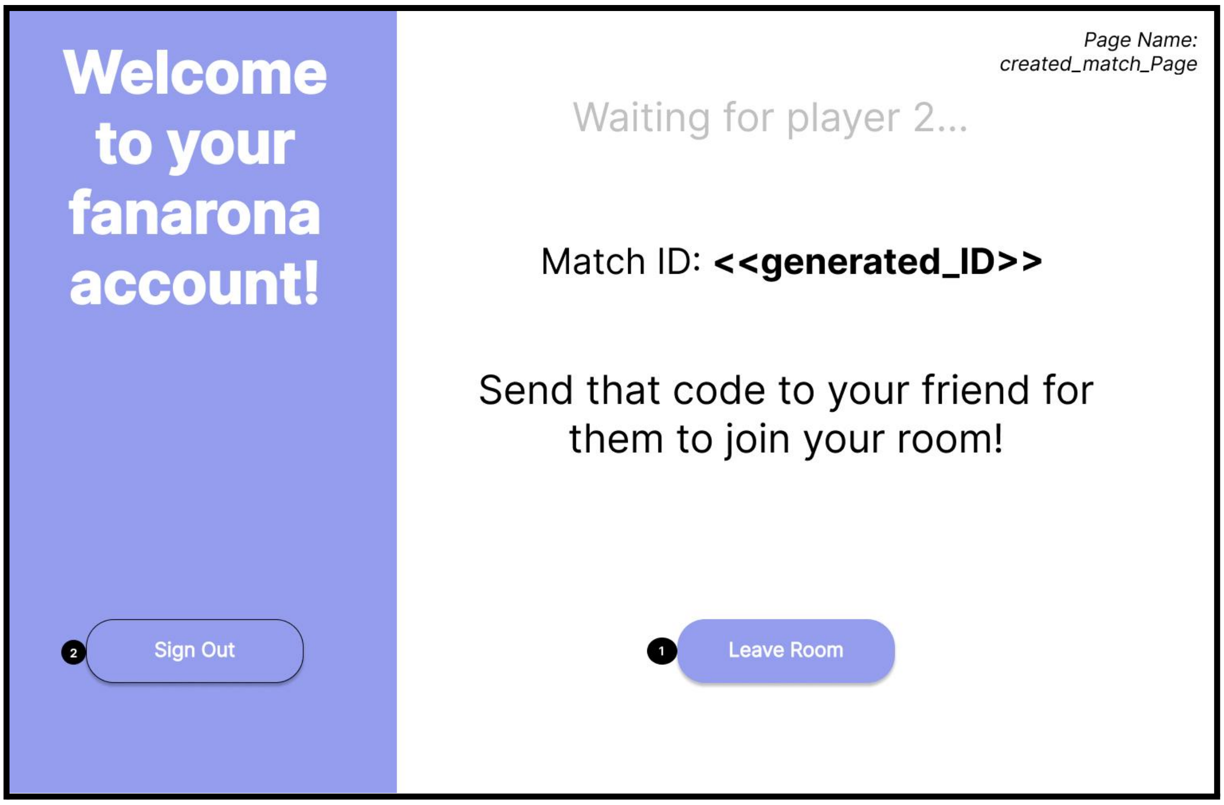
Confirm your Password

Password

Sign up


Welcome to your fanarona account!
Page Name: Account_Page
Username: <<username>>
Email: <<user_email>>
1 Create Game
2 Join Game
3 Sign Out


Welcome to your fanarona account!
Page Name: join_match_Page
Join Match
Match ID:
1 Join Room
2 Back
3 Sign Out



Page Name:
created_match_Page

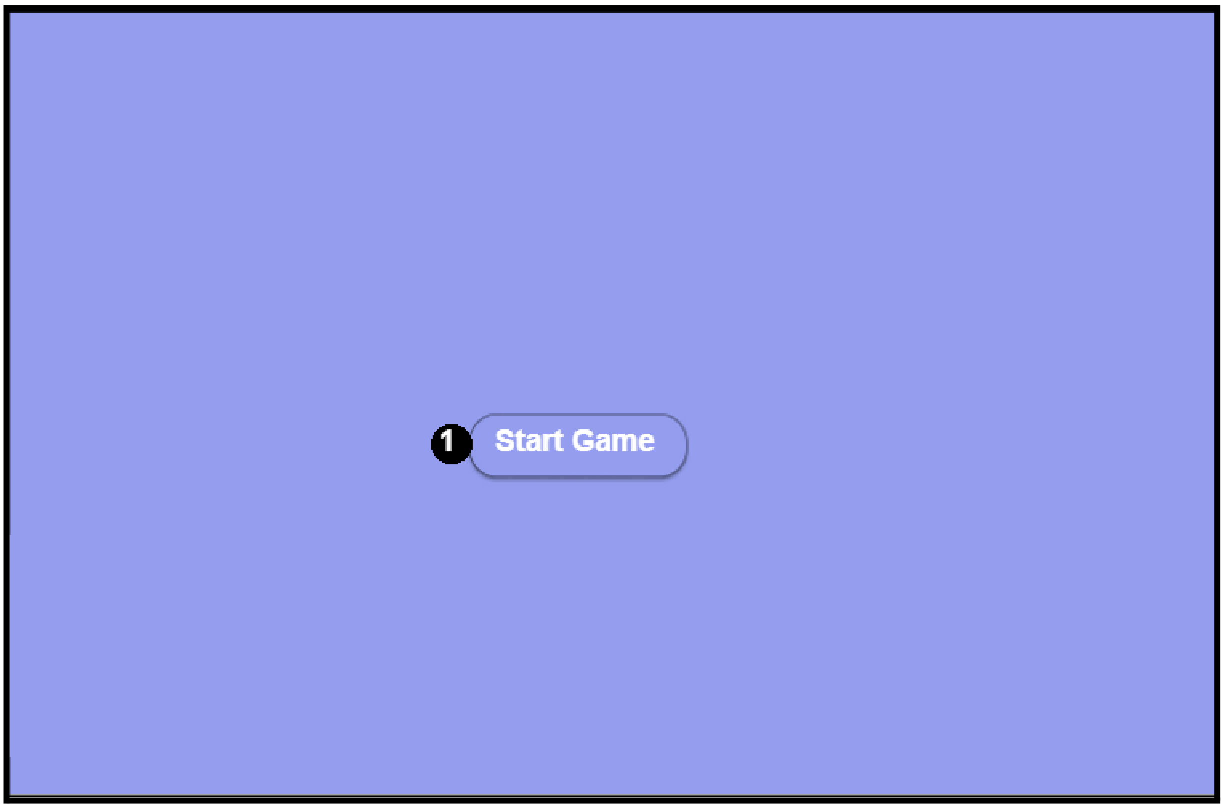
Welcome to your fanarona account!

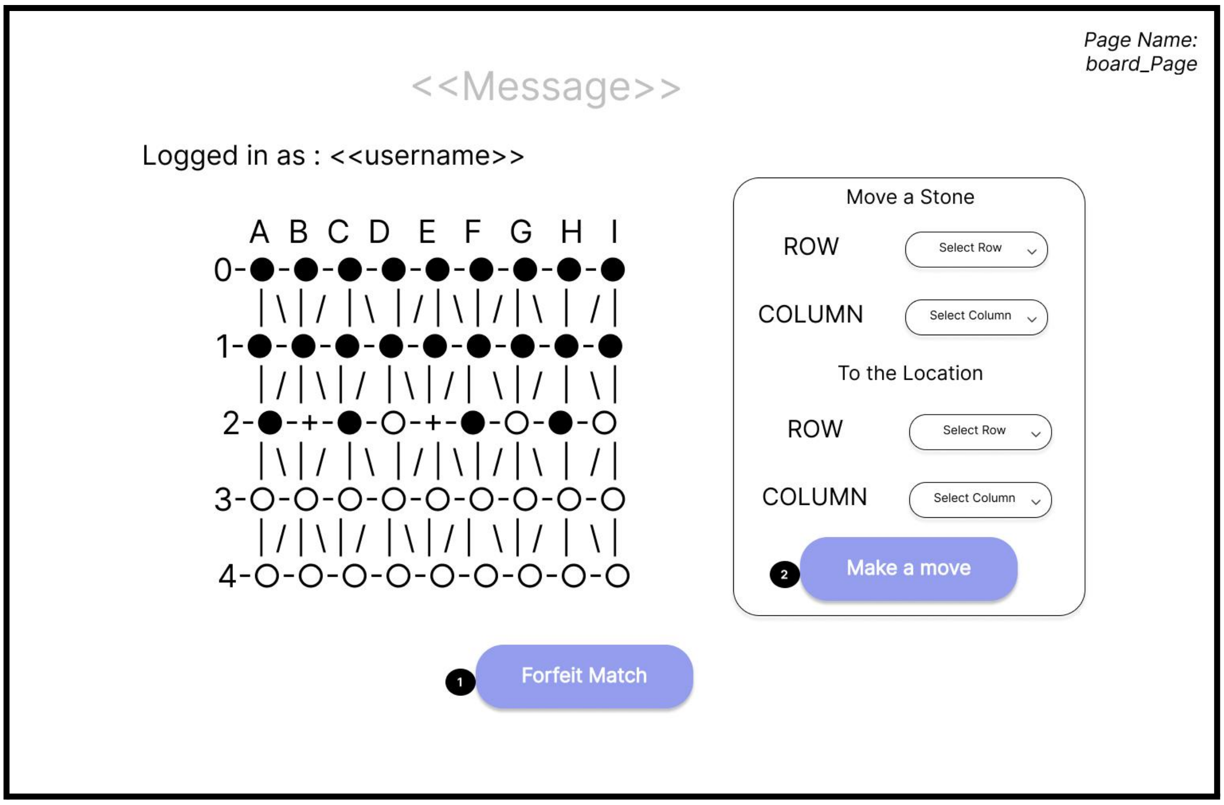
Waiting for player 2...

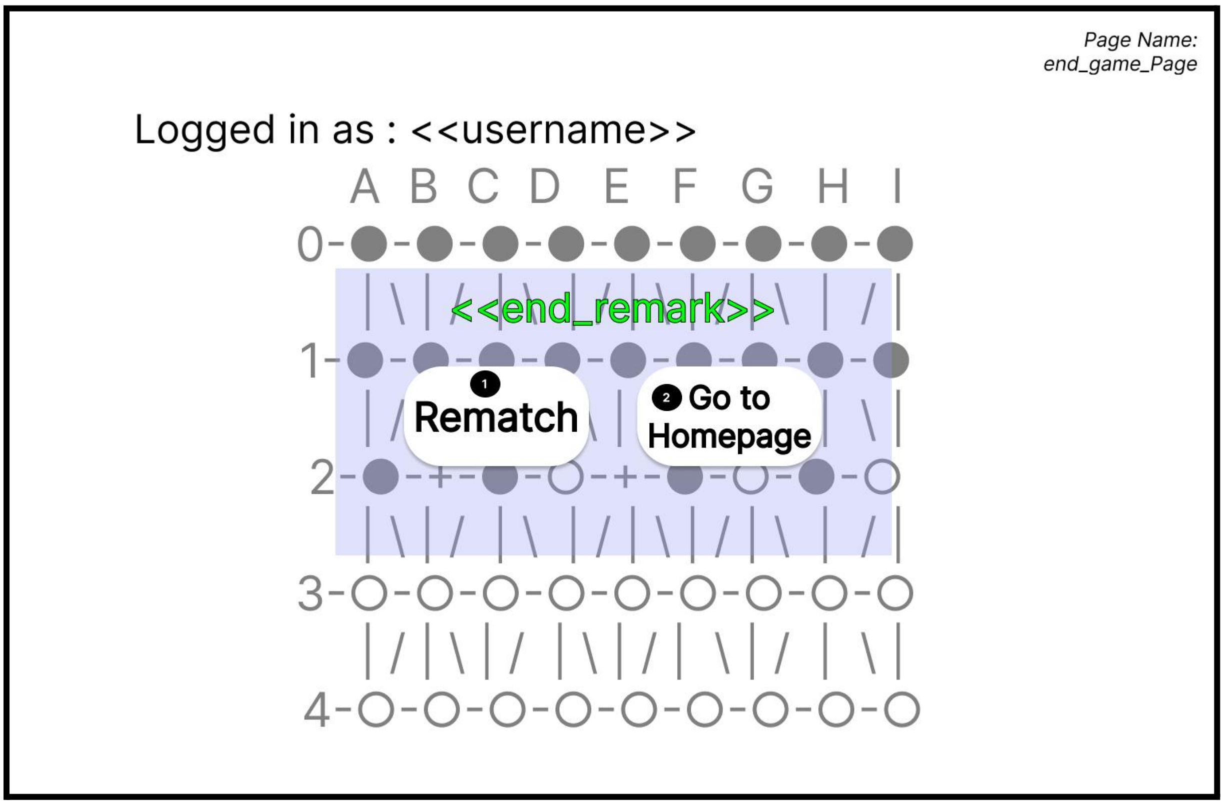
Match ID: **<<generated_ID>>**

Send that code to your friend for them to join your room!

Page Name:
match#load_match




Page Name:
board_Page
<<Message>>
Logged in as : <<username>>
A B C D E F G H I
0-●-●-●-●-●-●-●-●-●
|\|/|\|/|\|/|\|/|
1-●-●-●-●-●-●-●-●-●
|/|\|/|\|/|\|/|\|
2-●-+-●-○-+-●-○-●-○
|\|/|\|/|\|/|\|/|
3-○-○-○-○-○-○-○-○-○
|/|\|/|\|/|\|/|\|
4-○-○-○-○-○-○-○-○-○
Move a Stone
ROW
Select Row
COLUMN
Select Column
To the Location
ROW
Select Row
COLUMN
Select Column
2
Make a move
1
Forfeit Match


Page Name:
end_game_Page
Logged in as : <<username>>
A B C D E F G H I
0-●-●-●-●-●-●-●-●-●
<<end_remark>>
1-●-●-●-●-●-●-●-●-●
1
Rematch
2 Go to Homepage
2-●-+-●-○-+-●-○-●-○
3-○-○-○-○-○-○-○-○-○
|/|\|/|\|/|\|/|\|
4-○-○-○-○-○-○-○-○-○

Description of components within Web Pages

Each of the webpages shown have a title at the top right hand corner that ends with a suffix *"_Page"*. This is used to refer to each webpage when explaining the URL messages associated with each component for that particular page. Beside each component that has an URL there is an identifier, like this one here: ❶

The description of the URL messages for each component within a web page is given below:

Page name: Sign_In_Page			
Identifier	Component type	Action	URL message
1	Button	On Click	sign_up_path
2	Button	On Click	login_path

Page name: Sign_Up_Page			
Identifier	Component type	Action	URL message
1	Button	On Click	login_path
2	Button	On Click	sign_up_path

Page name: Account_Page			
<<username>> : This is a placeholder that would display the username of the user logged into the game.			
<<user_email>> :This is a placeholder that would display the email of the user logged into the game.			
Identifier	Component type	Action	URL message
1	Button	On Click	create_path
2	Button	On Click	join_path
3	Button	On Click	log_out_path

Page name: join_match_Page			
Identifier	Component type	Action	URL message
1	Button	On Click	join_path
2	Button	On Click	login_path(userID : @user.ID)
3	Button	On Click	log_out_path

Page name: created_match_Page			
<<generated_ID>> : This is a placeholder for a random ID of length 7, created from a mixture of alphabets and numbers.			
Identifier	Component type	Action	URL message
1	Button	On Click	login_path(userID : @user.ID)
3	Button	On Click	log_out_path

Page name: In_game_Page			
<<Message>> : This is a placeholder for messages to the player, indicating what is happening with the game. The expected messages would be: ● Waiting for <player2>'s turn ● Your turn ● Invalid move, try another move.			
Identifier	Component type	Action	URL message
1	Button	On Click	forfeit_path

Page name: match#load_match			
● After both players have joined the room, they will both get this page with a button to start the match which says ("Start Game"). Whenever the first player clicks on this button, it will start the match by sending a message through the channel informing the other player that the game has been started thus loading the board_page			
Identifier	Component type	Action	URL message

1	Button	On Click	load_match_path

Page name: board_Page

<<username>> : This is a placeholder that would display the username of the user logged into the game.

<<Message>> : This is a placeholder for messages to the player, indicating what is happening with the game. The expected messages would be:

- Waiting for <player2>’s turn
- Your turn
- Capture move performed
- Withdrawal move performed
- Invalid move, try another move.

Board related algorithms(front end side)

- The ‘refresh’ URL is called each time after make a move, and initially to set up the board. This returns a 2d array of enums (:white, :black, or nil) that represent if there is a white stone, black stone, or neither
- updateBoardView(board: 2darray[row][col] -> enum)
    - Enum used for colour = :black, :white; nil is used if undefined
    - Prints values to a textbox that represents the board as a text based array.
    - Print `"  A B C D E F G H I"` for the labeling of the game board columns.
        - Loop through each row in the game board:
            1. Print the current index of the loop starting at 0 in order to label the row of the game board.
            2. Print: `" |\|/|\|/|\|/|\|/|(newline)"` if the current row is even and not the first row.
            3. Print `" |/|\|/|\|/|\|/|\|(newline)”` if the current row is odd and not the first row.
            4. Loop through each column in the row:
                a. If the location at the current index is empty:
                    i. Print `"-+"`.
                b. If the Enumeration is :black:
                    i. Print `"-●"`.
                c. If the Enumeration is :white:
                    i. Print `"-○"`.
            5. Print `"(newline)"`

Identifier	Component type	Action	URL message
1	Button	On Click	forfeit_path
2	Button	On Click	make_move_path

Page name: end_game_Page			
<<username>> : This is a placeholder that would display the username of the user logged into the game.			
<<end_remark>> : This is a placeholder that would display whether the logged in player lost or won the match.			
Identifier	Component type	Action	URL message
1	Button	On Click	create_path
2	Button	On Click	login_path(userID : @user.ID)