

🗄 CHECKERS ONLINE 🗄

Software Design Document

Group Members	Zacharia Thottakara Yansen Tjandra Lucas Vitalos Andrew Yaros
Faculty Adviser	Filippos Vokolos, Ph.D

Revision History

Name	Date	Reason for Change	Revision
Andrew Yaros	8/4/2018	Initial setup/formatting.	0.1
Andrew Yaros Lucas Vitalos	8/9/2018	Added basic overviews of some topics.	0.2
Andrew Yaros Lucas Vitalos Zacharia Thottakara Yansen Tjandra	8/12/2018	Fleshing out points, adding diagrams, document cleanup and formatting. Final Version.	1.0

Table of Contents

Revision History	2
Table of Contents	3
Introduction	5
Purpose of Document	5
Scope of Document	5
Definitions, Acronyms, Abbreviations	5
System Overview	6
Description of Software	6
Technologies Used	6
System Architecture	6
Architectural Design Components	6
Design Rationale	7
Component Design	8
Overview	8
Server design	8
Server	9
GameManager	10
PlayerManager	12
GameResult	12
GameState	13
InviteResult	13
InviteState	14
RematchResult	14
RematchState	15
MoveResult	15
MoveState	16
Game	16
Player	17
Board	18
Move	20
Piece	20
King	21
Client HTML Interface	22
	3

State Diagram of HTML Client:	23
Human Interface Design	23
Menu systems	23
Checkers board interface	24
Server interface	25
References	26
Spark	26
Ubuntu Configuration	26

1. Introduction

1.1. Purpose of Document

- 1.1.1. This document's purpose is to describe the implementation of Checkers Online software, as described in the Checkers Online requirements document. Checkers Online is an online game, accessed through a web browser, allowing two players to remotely play a game of checkers.

1.2. Scope of Document

- 1.2.1. This document will describe in detail how to implement the Checkers Online Software, which will consist of several systems.

1.3. Definitions, Acronyms, Abbreviations

- 1.3.1. **Game State** - A state composed of: the state of the board, the state of all pieces, the state of all counters, and the active player.
 - 1.3.1.1. **Board State** - The possible moves of any currently selected piece
 - 1.3.1.2. **Pieces State** - The positions of every piece active or inactive.
 - 1.3.1.3. **Counter States** - The current values for certain statistics recorded in the game
 - 1.3.1.3.1. **Received** - Pieces the player has taken
 - 1.3.1.3.2. **Taken** - Pieces the opponent has taken
 - 1.3.1.3.3. **Number of moves** -over the whole game
 - 1.3.1.4. **Turn** - there is only one active player at a time since checkers is a turn based game.
- 1.3.2. **Client** - Any browser currently connected to the server
 - 1.3.2.1. **History** - A textual record of the User's past games
 - 1.3.2.1.1. In the form: PC3-D4,OF4-E5,...,CE1-G3,W
 - 1.3.2.2. **User** - Any person currently logged in.
 - 1.3.2.3. **Player** - Any user in an active game session.
- 1.3.3. **Server** - headless Ubuntu 17.04 installation that runs the main computations for the game.
 - 1.3.3.1. **UUID** - Unique User Identification (String)

2. System Overview

2.1. Description of Software

- 2.1.1. Checkers Online is an Online Checkers game which allows a player to remotely play a game of checkers with other players. When a player opens the application, they will play a game of checkers against the computer

2.2. Technologies Used

- 2.2.1. The system will utilize a client-server model for its operation. The server will be written in Java, and will utilize the Spark micro-framework, running on Ubuntu 17.04.
- 2.2.2. Checkers Online will target desktop web browsers, primarily Chrome and Firefox on Windows and macOS, Edge on Windows, and Safari on macOS.
 - 2.2.2.1. It will use both HTML and Javascript.
- 2.2.3. Atlassian BitBucket will be used for version control during the development of the system.

3. System Architecture

3.1. Architectural Design Components

- 3.1.1. Networking - This component is fairly simple. Upon Opening the webpage, the client device will create an session on the server to maintain communication.
- 3.1.2. Communication will occur as Get and POST requests upon event from the client machine.
 - 3.1.2.1. The server will only ever respond to the user. No commands are pushed
 - 3.1.2.2. When the user is waiting (Joining game, waiting for other player to move), the client will simply be reading on the socket waiting for data from the server
- 3.1.3. Spark micro-framework server - responsible for:
 - 3.1.3.1. Computing legal moves
 - 3.1.3.2. Keeping the game state when players are in a match

- 3.1.3.3. Keeping track of active users
- 3.1.3.4. Matchmaking
- 3.1.3.5. Data Formatting
 - 3.1.3.5.1. Data will be processed serverside and the simplest data possible will be sent back to the client for updates
- 3.1.4. HTML/Javascript client - The client will consist of a menu system for accessing the client and a separate page for accessing the game. The client is responsible for:
 - 3.1.4.1. Graphical interface -visual depiction that the user will interact with
 - 3.1.4.2. Get and post request to server
 - 3.1.4.3. Basic HTML navigation

3.2. Design Rationale

- 3.2.1. Why Client-Server Relationship?

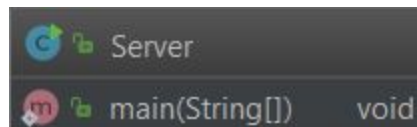
To allow for faster computation, we contemplated trying to create a peer-to-peer model, so all computation would happen immediately and locally. While this would have made for faster computation, we realized we needed a centralized source to manage all of the users and the matchmaking. With the Client-Server model, we are able to generate a UUID per user when they connect, and free them up when they disconnect. This makes it less likely for joining a random game to map to a user no longer signed in.
- 3.2.2. Why have all calculation done on the server?

To have the client do calculation for possible moves per piece, the client would have to wait to get the updated board and piece states from the server, then do the calculations before the client is ready to highlight possible moves. We saw this as a problem. Instead we opted to include the possible moves in the piece states. That way when the server receives an updated move, it can update both states, and do the move calculations to update the individual pieces moves and send all of that information to the client. This means there is no calculation necessary by the client, membership checking a set of possible moves.

- 4.2.1.2. GameManager
Keeps track of active games, rematch requests, and invites requests.
- 4.2.1.3. PlayerManager
Generates user IDs
- 4.2.1.4. "Result" classes
Objects to represent the result of a particular action or request.
Each one has a corresponding "State" enumeration.
 - 4.2.1.4.1. GameResult
State of a game after a move or other game state change
 - 4.2.1.4.2. InviteResult
State of an invite request
 - 4.2.1.4.3. RematchResult
State of a rematch request
 - 4.2.1.4.4. MoveResult
State of a move
- 4.2.1.5. Game
A single active game, which contains a Board and an ArrayList of players.
- 4.2.1.6. Player
Contains data about an individual user, including their UUID.
- 4.2.1.7. Board
Contains an two-dimensional ArrayList of pieces representing the state of the board itself, as well as two-dimensional ArrayLists representing the sets of captured pieces and pieces in play.
- 4.2.1.8. Move
Represents a single move.
- 4.2.1.9. Piece
Base class for pieces. Simple object with integer ID.
- 4.2.1.10. King
Decorator for pieces with a base piece and an optional "hat" piece.

What follows are detailed breakdowns of these classes.

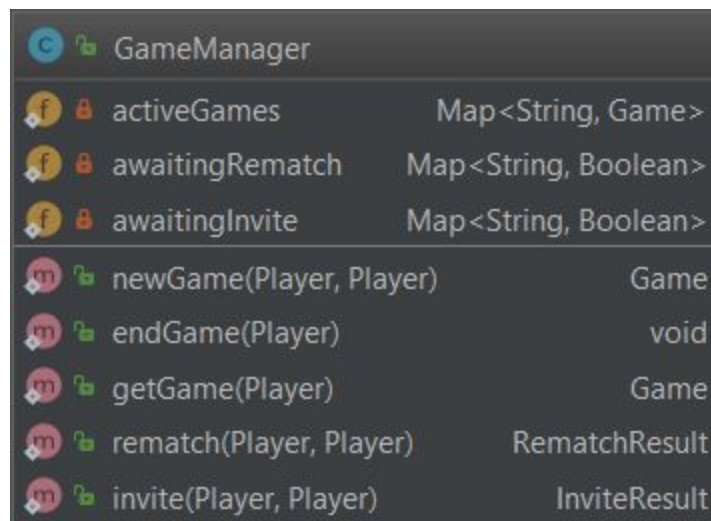
Server



Name	Signature	Description
------	-----------	-------------

main()	String[] → void	This the main routine. It creates Spark routes, which implicitly starts up a server listening on the port we specify (see References #6.1.1). Through these routes we will handle all client-server interaction. Requests to the routes we set up will be handled by callback functions, which we will specify as lambda expressions.
--------	-----------------	---

GameManager

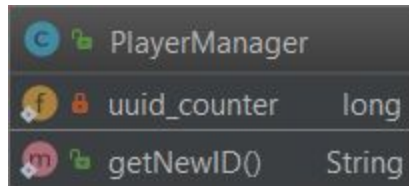





	GameManager	
	activeGames	Map<String, Game>
	awaitingRematch	Map<String, Boolean>
	awaitingInvite	Map<String, Boolean>
	newGame(Player, Player)	Game
	endGame(Player)	void
	getGame(Player)	Game
	rematch(Player, Player)	RematchResult
	invite(Player, Player)	InviteResult

Name	Signature	Description
activeGames	Map<String, Game>	Map of player UUIDs to active games. Permits one active game per player.
awaitingRematch	Map<String, Boolean>	Maps player UUIDs to booleans indicating that a user has either requested or declined a

		rematch.
awaitingInvite	Map<String, Boolean>	Maps player UUIDs to booleans indicating that a user has either requested or declined an invite.
newGame()	Player, Player → Game	Creates a new game with the specified players and returns the game object. Also adds that game object to the map of active games for both players.
endGame()	Player → void	Removes the entry for the given player from the map of active games, if one exists.
getGame()	Player → Game	Gets the active game for the specified player if one exists.
rematch()	Player, Player → RematchResult	Submits a rematch request, which will have the effect of either accepting the other player's request or waiting for the other player to accept or deny the request.
invite()	Player, Player → InviteResult	Submits an invite request, which will have the effect of either accepting the other player's invite request or waiting for the other player to accept or deny the request.


PlayerManager




	PlayerManager	
	uuid_counter	long
	getNewID()	String

Name	Signature	Description
uuid_counter	long	Counter for generation of UUIDs.
getNewID()	void → String	Returns a new UUID as a string in the form "UXXXXX" (e.g. "U00012")

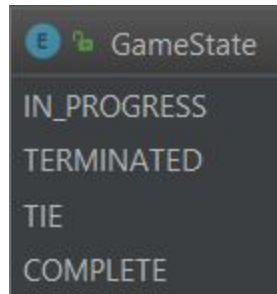
GameResult



	GameResult	
	loser	Player
	winner	Player
	state	GameState

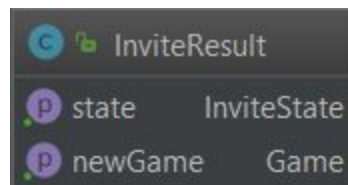
Name	Signature	Description
loser	Player	The loser of the game, if there is one.
winner	Player	The winner of the game, if there is one.
state	GameState	The current state of the game.

GameState



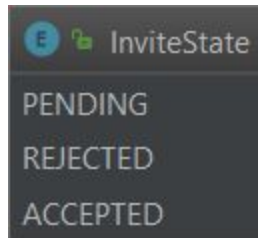
Name	Description
IN_PROGRESS	Indicates that the game is ongoing.
TERMINATED	Indicates that the game ended abnormally.
TIE	Indicates the game ended in a tie.
COMPLETED	Indicates that the game ended normally.

InviteResult



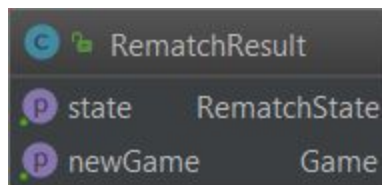
Name	Signature	Description
state	InviteState	The current state of the invitation.
newGame	Game	The new game with the inviter and invitee, if it exists.

InviteState



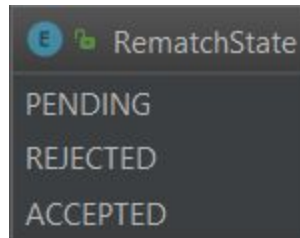
Name	Description
PENDING	Indicates that the request is awaiting acceptance or rejection.
REJECTED	Indicates that the request was rejected.
ACCEPTED	Indicates that the request was accepted.

RematchResult



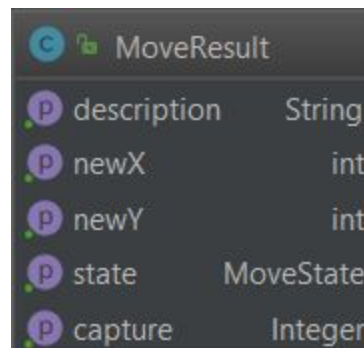
Name	Signature	Description
state	RematchState	The current state of the rematch request.
newGame	Game	The new game with the requester and requestee, if it exists.

RematchState



Name	Description
PENDING	Indicates that the request is awaiting acceptance or rejection.
REJECTED	Indicates that the request was rejected.
ACCEPTED	Indicates that the request was accepted.

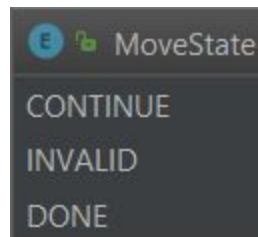
MoveResult



Name	Signature	Description
description	String	Description of the move, for the client's move list.

newX	int	The new x coordinate for the moved piece.
newY	int	The new y coordinate for the moved piece.
state	MoveState	The state of the move.
capture	Integer	The ID of the captured piece, if there is one.

MoveState



Name	Description
CONTINUE	Indicates that the player's turn must continue (they are able to capture additional pieces).
INVALID	Indicates that the player's move was invalid.
DONE	Indicates that the player's turn is over.

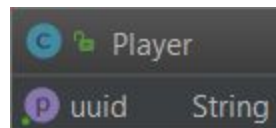
Game



Name	Signature	Description
------	-----------	-------------































prioritize()	Player, Player → ArrayList<Player>	Determines turn order (priority) by a simulated coin flip.
move()	Int, Int, Int, Int → MoveResult	Takes old coordinates and new coordinates and attempts to move the piece at the old coordinates to the new location.
checkGameState()	void → GameResult	Gets the current state of the game.
players	ArrayList<Player>	The game's players.
board	Board	The game board for this game.

Player



Name	Signature	Description
uuid	String	Unique user identifier.

Board

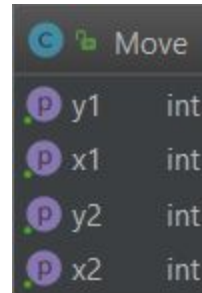
  Board	
  layout	ArrayList<ArrayList<Piece>>
  piecesInPlay	ArrayList<ArrayList<Piece>>
  capturedPieces	ArrayList<ArrayList<Piece>>
  moveList	ArrayList<String>
  moveCount	int
  lastCapture	int
  side(Piece)	int
  slot(Piece)	int
  move(int, int, int, int)	MoveResult
  capture(Piece)	void
  canCapture(int, int)	boolean
  crown(Piece)	Piece
  hasLegalMove(int)	boolean
  inStalemate()	boolean

Name	Signature	Description
layout	ArrayList<ArrayList<Piece>>	A two-dimensional ArrayList of Pieces representing the current layout of the pieces on the board.
piecesInPlay	ArrayList<ArrayList<Piece>>	A two-dimensional ArrayList of Pieces representing the pieces currently in play.
capturedPieces	ArrayList<ArrayList<Piece>>	A two-dimensional ArrayList of pieces representing the captured pieces.
moveList	ArrayList<String>	The list of moves in made in the game so

		far.
moveCount	int	The number of moves made in total (by both players).
lastCapture	int	The move on which the last capture was made.
side()	Piece → int	The side that the piece belongs to -- 0 to represent the player who went first, 1 to represent the player who went second. Used to index the piecesInPlay and capturedPieces lists.
slot()	Piece → int	The "slot" of the piece -- a number between 0 and 11. Used to index the piecesInPlay and capturedPieces lists.
capture()	Piece → void	Takes the specified piece out of play and puts it in the list of captured pieces.
canCapture()	int, int → boolean	Returns true if there is a possible capture for a piece at the given coordinates, false otherwise.
crown()	Piece → Piece	Crowns a piece and returns the crowned piece. Uses one of the player's formerly lost pieces to crown the new piece, if one is available.
getLegalMoves()	int → ArrayList<Move>	Get the list of legal moves for a the given side.
inStalemate()	void → boolean	Returns true if there hasn't been a capture in

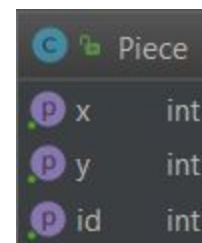
		the last 40 turns, false otherwise.
--	--	-------------------------------------

Move



Name	Signature	Description
y1	int	Initial y coordinate of a piece.
x1	int	Initial x coordinate of a piece.
y2	int	Intended y coordinate of a piece.
x2	int	Intended x coordinate of a piece.

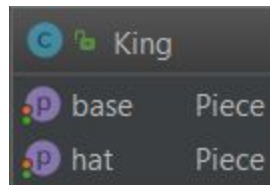
Piece



Name	Signature	Description
x	int	X coordinate of the piece.
y	int	Y coordinate of the piece.

id	int	The piece's ID. Odd IDs belong to first player, Even IDs belong to second. Will be a number between 0 and 23.
----	-----	---

King



Name	Signature	Description
base	Piece	The base piece for the king.
hat	Piece	The piece used to crown this one, if there is one.

4.3. Client HTML Interface

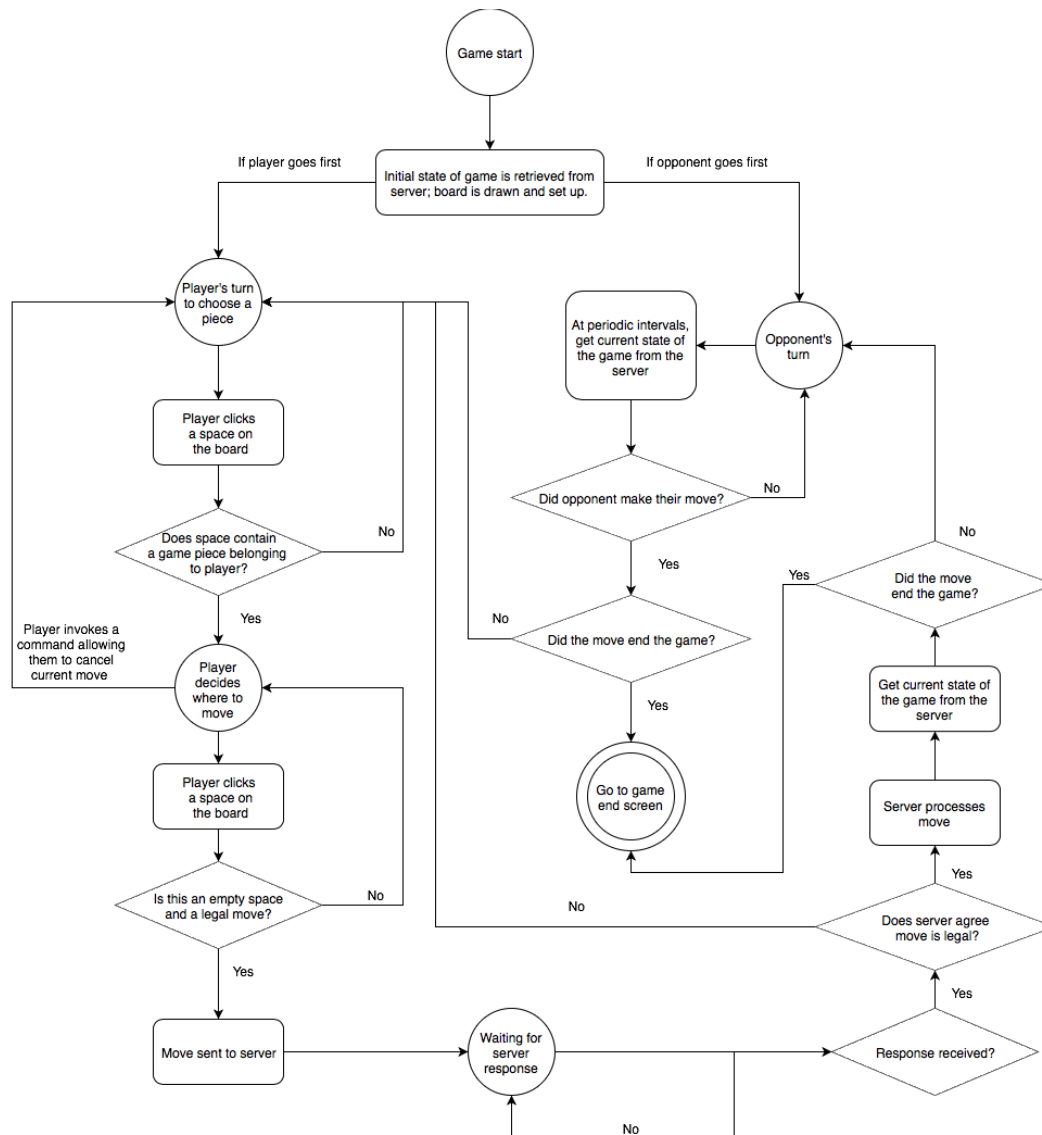
- 4.3.1. The client consists of an HTML page, a Javascript file, and a CSS file.
- 4.3.2. The client will contact the server using GET and POST requests.
- 4.3.3. The HTML page will contain several canvas elements. Some canvases will overlap others; the order in which they are drawn is defined by a z-index parameter.

Canvas	Purpose	Misc.
Board	For drawing the 8 x 8 game board	Z-index: 1
Pieces	For drawing game pieces, overlaid on Board	Z-index: 2
Overlay	For highlighting spaces on the board, overlaid on Pieces	Z-index: 3
PlayerPieces	For displaying pieces the player has captured from the opponent, displayed underneath Board	
OppoPieces	For displaying pieces the opponent has taken from the player, displayed underneath Board	

- 4.3.4. The game client, written in HTML and javascript, will contain its own representation of the current state of the game, including:
 - 4.3.4.1. The color of the player (black or red)
 - 4.3.4.2. Locations of pieces currently on the board
 - 4.3.4.3. Which pieces are kings
 - 4.3.4.4. Captured pieces, which are represented underneath the board
 - 4.3.4.5. Whether or not it is currently the players turn
- 4.3.5. The client's information on the game will be periodically updated from the server.
- 4.3.6. At the start of the game, the client will load the initial state of the board from the server.
- 4.3.7. Upon receiving new information from the Board class for the current game, the board will be redrawn, including any accompanying pieces.
- 4.3.8. If it is the player's turn to play, each space on the board will be clickable. To move, a player must click on a space on the board. An event listener will determine if the click is on the board, and which space on the board was clicked. Each space on the board are represented by an X and Y

coordinate. Depending on the current state of the client, the app will send the coordinates to an appropriate function.

4.4. State Diagram of HTML Client:



5. Human Interface Design

5.1. Menu systems

5.1.1. Client System

5.1.1.1. The Menu system interface will be navigated through button presses in HTML Triggering Javascript calls to the Server for

information and Pages. Some buttons may also be simple http links to other pages.

- Components
 - Main menu
 - Options
 - Join Game By UUID -Javascript button with textbox
 - Join Random Game -Javascript button
 - Info Page -HTML hyperlink button
 - Displayed
 - Game name -Checkers Online
 - Player UUID
 - Current Time
 - Info Menu
 - HTML hyperlink to about page
 - Javascript button to History Page
 - Game end Menu
 - Rematch? -javascript button to Rematch Offer Menu
 - Main Menu -HTML hyperlink to Main menu
 - Rematch Offer Menu
 - Yes -Javascript Get request
 - No -HTML hyperlink to Main menu
 - Pages
 - About Page
 - Displays information about the game and developers
 - History Page
 - Displays information about the Current User's game session
 - Here a user can look at a textual history of previous games played
 - Game Window
 - This interface is defined below in section 5.2. To see a more accurate depiction of the Game window can be found as Figure 01 of the Requirements document. With a description in Section 3.2.3 of the Requirements Document.

5.2. Checkers board interface

- 5.2.1. At the center of the screen will be the main interactable component, the board. The board is an 8x8 checkerboard with a red and black color scheme, and labeled axes, 1-8 on the bottom and A-H on the side. On the top three rows, only on the black positions, the opponents pieces will be

placed. On the bottom three rows, only on the black positions, the players pieces will be placed.

5.2.2. In-Game Interaction States

5.2.2.1. The the player may interact with their pieces only. When a piece is clicked on, a javascript command will be sent to the server and the server will send a response of an updated board. This response will highlight whatever places the player is able to move, and update the game board accordingly. This feature is only available while it is the players turn

5.2.2.2. Once viewing the highlighted board, the user may click on a highlighted position or choose another piece to change which piece is selected.

5.2.2.3. When a highlighted position is selected the client will send a get request to the server to update the move selection and change who's turn it is.

5.2.2.4. Finally the game will update on both client devices and swap to the opponent's move

5.2.3. The other interactable is an exit button in the top right corner with which a user can exit any game and immediately be taken back to the main menu.

5.2.3.1. Once a user has exited the game a new game will need to be started, there are no resumes.

5.2.4. While it is not the player's turn, the game will only allow the option to exit. All piece buttons are turned off.

5.3. Server interface

5.3.1. The server will be running on a headless version of Ubuntu 17.04. Interfacing with the server will happen via ssh and the command line interface.

5.3.1.1. There will be version control on the server with make targets for building and starting the server.

5.3.1.2. The server can be restarted and configured to spin up the service on restart

5.3.1.3. The manual launch will simply be a file to run with the potential for command line arguments to define how the service runs.

6. References

6.1. Spark

- 6.1.1. <http://sparkjava.com/documentation>

6.2. Ubuntu Configuration

- 6.2.1. <https://help.ubuntu.com/lts/serverguide/index.html>

6.3. HTML/Javascript

- 6.3.1. <https://developer.mozilla.org/en-US/docs/Web/HTML>
- 6.3.2. <https://www.ibm.com/developerworks/library/wa-canvashtml5layering/index.html>

6.4. Java

- 6.4.1. <https://docs.oracle.com/javase/8/docs/api/>