

PROFESSIONAL TRAINING REPORT
at
Sathyabama Institute of Science and Technology
(Deemed to be University)

Submitted in partial fulfillment of the requirements for the award of
Bachelor of Technology in Information Technology

By
K. Sheshasena Reddy
REG. NO. 39120111



DEPARTMENT OF INFORMATION TECHNOLOGY
SCHOOL OF COMPUTING

SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY
JEPPIAAR NAGAR, RAJIV GANDHI SALAI,
CHENNAI – 600119, TAMILNADU

APRIL 2022



SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade “A” by
NAAC

(Established under Section 3 of UGC Act, 1956)
JEPPIAAR NAGAR, RAJIV GANDHI SALAI
CHENNAI– 600119
www.sathyabama.ac.in

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of
K.SHESHASENA REDDY (Reg. No: 39120111) who carried out the
project entitled “**Chess With Friends** ” under my supervision from
January 2022 to April 2022.

Internal Guide

Dr . JEBERSON RETNA RAJ

Head of the Department

Submitted for Viva voce Examination held on _____

Internal Examiner

External Examiner

DECLARATION

I, **K.Sheshasena Reddy** hereby declare that the project report entitled **Chess With Friends** done by me under the guidance of **Dr Jeberson Retna Raj** is submitted in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Information and Technology.

DATE: 10/04/2022

PLACE:Chennai



SIGNATURE OF THE CANDIDATE

Acknowledgement

I am pleased to acknowledge my sincere thanks to **Board of Management of SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T. Sasikala M.E., Ph.D, Dean**, School of Computing, **Dr. R. Subhashini, M.E., Ph.D. Heads of the Department of Information Technology** for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr Jeberson Retna Raj** , for his valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Information and Technology** who were helpful in many ways for the completion of the project.

TRAINING CERTIFICATE

ABSTRACT

Chess is a game of strategy and tactics for two players, played on an 8x8 chequered board. Although chess sets come in many varieties and colors, the traditional colors are white and black.

The aim of the game is to trap your opponent's king, which is called Checkmate. A game can also be won if your opponent gives up (in chess, we call this 'resigning'), and there are a variety of ways a game can end in a draw, in which case neither player wins. We'll look at these in more detail later on.

Each player has an army consisting of a king, a queen, two rooks, two bishops, and two knights, and eight pawns.

Each turn, you must move one piece to a new square. The player with the white pieces goes first, and after that the players take it in turns to move a piece. There are no dice in chess - every piece has its own way of moving, and it's up to you which one you want to move. Each piece also has the ability to capture, or 'take', an enemy piece.

TABLE OF CONTENTS

CHAPTER No.	TITLE	PAGE No
	ABSTRACT	6
1	INTRODUCTION	9
2.	AIM AND SCOPE OF THE PRESENT	11
	2.1 AIM	11
	2.2 SCOPE	11
	2.3 RULES	
3.	EXPERIMENTAL OR MATERIALS AND METHODS, USED	14
	3.1 Software Required	14
	3.1.1 Windows 10	
	3.1.2 Node JS	
	3.1.3 React JS	
	3.2 Hardware Required	18
	3.2.1 Hard Disk	
	3.2.2 RAM	
	3.2.3 Processor	
	3.2.4 Mouse	
	3.2.5 Keyboard	
	3.2.6 Speaker	
4.	RESULTS AND DISCUSSION, PERFORMANCE Recommendation	23
	4.1 Result and discussion	23
	4.2 Recommendation	24
	4.3 Screen shots	
	4.3.1 Input	
	4.3.2 Output	

5	SUMMARY AND CONCLUSIONS	26
	REFERENCES	27
	APPENDIX	28
	SOURCE CODE	28
	SAMPLE SCREEN	56

CHAPTER - 1

INTRODUCTION

1.1 ABOUT THE CONCEPT

Chess is a game for two players, dubbed White and Black. The goal is to capture your opponent's king. In the game, this is known as a checkmate. Chess is played on a board with 64 squares. Each player begins with 16 pieces, lined up in two rows. The first row is occupied by pieces called pawns. The next row contains: a king, a queen, two rooks, two bishops, and two knights.

The game begins with the neutral position. The White player (the player of the light colored pieces) moves first. Then each player takes a single turn. In fact, a player must move in turn. In other words a move cannot be skipped.

When setting up the pieces, keep in mind two things. The light colored square goes on the player's right, and Queens go on their color next to the Kings on the center files. You may not move a piece to a square already occupied by one of your own pieces.

You may capture an opposing piece by replacing that piece with one of your own pieces, if it can legally move there

The primary objective is to provide a user-friendly interactive environment to the users of the application that helps them to play and communicate with a lot of ease.

CHAPTER - 2

AIM AND SCOPE OF PRESENT INVESTIGATION

2.1 AIM

The aim of the project is to provide both a chess as well as chat facility in one application such that two users can play the chess against each other as well as they can chat among themselves on the network that is any number of computers connected via a network. So the users can play the chess in pair and in the meantime when any one of them wants to communicate with the opponent player, he can also do that simultaneously.

2.2 SCOPE

Basically this project has two main purposes:

Firstly, to encapsulate the chess gaming and chatting in one application so that a user can easily view and use both of them on a single window.

Secondly, to make these two applications to run simultaneously on the network. Generally, the chess game is normally played against the human user and the computer on which the user is operating, so the need is to make the chess game being played against two human users on two different machines on the network, no matter their physical location and as well as allowing chat among those two users.

RULES OF THE GAME

THE KING:

The King can move one square in any direction. A King can never move into check, or onto a square where it can be captured by an opponent's piece. If a King is not in check, and no other legal move is possible, then the position is said to be in stalemate. A stalemated game is a draw, or a tie.

The QUEEN:

The Queen can move to any square in any direction as long as her path is not blocked. Her range and the ability to attack many pieces at once are the source of her power.

The ROOK:

The Rook can move to any square along its file or row as long as its path is not blocked. Its range is the source of its power.

The BISHOP:

The Bishop can move to any square along its diagonals as long as its path is not blocked. Its range is the source of its power.

The KNIGHT:

The Knight is the only piece that can hop over other pieces in an L-shaped path. This ability makes it particularly powerful in the early stage of a game when the board is crowded with pieces.

The PAWN:

The Pawn may move only one square forward if its path is not blocked. However, it may move as an option one or two squares forward on its first move only. It may capture only diagonally one square. It may not capture forward. It may not move backward. The lowly Pawn usually does not last long, but if it is able to reach the 8th row or rank, then it can promote itself to any other piece except the King. A Pawn thus promoted is replaced by that piece. Therefore, it is possible to have more than one Queen, or two Rooks, Bishops, or Knights on the board at one time

The primary objective in chess is to checkmate your opponent's King. When a King cannot avoid capture then it is checkmated and the game is immediately over. If a King is threatened with capture, but has a means to escape, then it is said to be in check. A King cannot move into check, and if in check must move out of check immediately.

There are three ways you may move out of check:

- 1) Capture the checking piece;
- 2) Block the line of attack by placing one of your own pieces between the checking piece and the King.

(Of course, a Knight cannot be blocked.);

- 3) Move the King away from check. If a King is not in check, and no other legal move is possible, then the position is said to be in stalemate. A stalemated game is a draw, or a tie.

CHAPTER 3

EXPERIMENTAL OR MATERIALS AND METHODS

3.1 SOFTWARE REQUIREMENT

- Windows 7 or above
- Node JS
- Angular CLI

3.1.1 Windows 10 or above

Windows 10 is a Microsoft operating system for personal computers, tablets, embedded devices and internet of things devices.

Microsoft released Windows 10 in July 2015 as a follow-up to Windows 8. The company has said it will update Windows 10 in perpetuity rather than release a new, full-fledged operating system as a successor.

3.1.2 Node JS

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection, the callback is fired, but if there is no work to be done, Node.js will sleep.

This is in contrast to today's more common concurrency model, in which OS threads are employed. Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node.js are free from worries of dead-locking the process, since there are no locks. Almost no function in Node.js directly performs I/O, so the process never blocks except when the I/O is performed using synchronous methods of Node.js standard library. Because nothing blocks, scalable systems are very reasonable to develop in Node.js.

If some of this language is unfamiliar, there is a full article on Blocking vs. Non-Blocking.

Node.js is similar in design to, and influenced by, systems like Ruby's Event Machine and Python's Twisted. Node.js takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems, there is always a blocking call to start the event-loop. Typically, behavior is defined through callbacks at the beginning of a script, and at the end a server is started through a blocking call like `EventMachine::run()`. In Node.js, there is no such start-the-event-loop call. Node.js simply enters the event loop after executing the input script. Node.js exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript — the event loop is hidden from the user.

HTTP is a first-class citizen in Node.js, designed with streaming and low latency in mind. This makes Node.js well suited for the foundation of a web library or framework.

Node.js being designed without threads doesn't mean you can't take advantage of multiple cores in your environment. Child processes can be spawned by using our `child_process.fork()` API, and are designed to be easy to communicate with. Built upon that same interface is the `cluster` module, which allows you to share sockets between processes to enable load balancing over your cores.



3.1.3 REACT JS

This topic can help you understand React: what Angular is, what advantages it provides, and what you might expect as you start to build your applications.

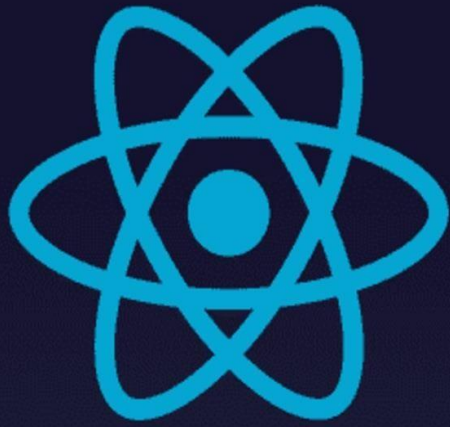
React is a JavaScript-based UI development library. Facebook and an open-source developer community run it. Although React is a library rather than a language, it is widely used in web development. The library first appeared in May 2013 and is now one of the most commonly used frontend libraries for web development.

Easy creation of dynamic applications: React makes it easier to create dynamic web applications because it requires less coding and offers more functionality, as opposed to JavaScript, where coding often gets complex very quickly.

Improved performance: React uses Virtual DOM, thereby creating web applications faster. Virtual DOM compares the components' previous states and updates only the items in the Real DOM that were changed, instead of updating all of the components again, as conventional web applications do.

Reusable components: Components are the building blocks of any React application, and a single app usually consists of multiple components. These components have their logic and controls, and they can be reused throughout the application, which in turn dramatically reduces the application's development time.

Unidirectional data flow: React follows a unidirectional data flow. This means that when designing a React app, developers often nest child components within parent components. Since the data flows in a single direction, it becomes easier to debug errors and know where a problem occurs in an application at the moment in question



ReactJS

3.2 HARDWARE REQUIREMENT

- Hard Disk
- RAM – 1 GB
- Processor – Dual Core or Above
- Mouse
- Keyboard
- Monitor
- Speaker

3.2.1 Hard Disk

A hard disk or fixed disk, is an electro-mechanical data storage device that uses magnetic storage to store and retrieve digital information using one or more rigid rapidly rotating disks (platters) coated with magnetic material.

During the mid-1990s the typical hard disk drive for a PC had a capacity of about 1 gigabyte. As of January 2019, desktop hard disk drives typically had a capacity of 1 to 6 terabytes, with the largest-capacity drives reaching 15 terabytes

3.2.2 RAM

Random access memory (RAM) is a type of data storage used in computers that is generally located on the motherboard. This type of memory is volatile and all information that was stored in RAM is lost when the computer is turned off.

The RAM chip may be individually mounted on the motherboard or in sets of several chips on a small board connected to the motherboard. Older memory types were in the form of chips called dual in-line package (DIP). Although DIP chips are still used today, the majority of memory is in the form of a module, a narrow printed circuit board attached to a connector on the motherboard. The three main memory circuit boards types containing chips are: RIMMs (Rambus in-line memory modules), DIMMs (dual in-line memory modules) and SIMMs (single in-line memory modules). Most motherboards today use DIMMs. Sometimes, people get confused about the difference between memory and storage, in part because both can be measured in megabytes (MB), gigabytes (GB) and terabytes (TB).

By contrast, storage is slower, but it can retain data when the device is powered down. So, for example, if a document has been saved to a hard drive prior to a power outage or system crash, the user will still be able to retrieve it when the system is back up and running.

3.2.3 Processor

A processor is an integrated electronic circuit that performs the calculations that run a computer. A processor performs arithmetical, logical, input/output (I/O) and other basic instructions that are passed from an operating system (OS). Most other processes are dependent on the operations of a processor.

Dual core is a CPU that has two distinct processors that work simultaneously in the same integrated circuit. This type of processor can function as efficiently as a single processor but can perform operations up to twice as quickly.

3.2.4 Mouse

A mouse is a small handheld input device that controls a computer screen's cursor or pointer in conjunction with the way it is moved on a flat surface. The mouse term name originates from its likeness to a small, corded and elliptical shaped device that looks like a mouse tail.

Some mouse devices have integrated features, such as extra buttons that may be programmed and assigned with different commands.

Because the mouse reduces the use of a keyboard, its invention and continuous innovation is considered one of the most important breakthroughs in computer ergonomics.

3.2.5 Keyboard

A computer keyboard is an input device used to enter characters and functions into the computer system by pressing buttons, or keys. It is the primary device used to enter text. A keyboard typically contains keys for individual letters, numbers and

special characters, as well as keys for specific functions. A keyboard is connected to a computer system using a cable or a wireless connection.

A computer keyboard is an input device used to enter characters and functions into the computer system by pressing buttons, or keys. It is the primary device used to enter text. A keyboard typically contains keys for individual letters, numbers and special characters, as well as keys for specific functions.

3.2.6 Monitor

A computer monitor is an electronic device that shows pictures for computers. Monitors often look similar to televisions. The main difference between a monitor and a television is that a monitor does not have a television tuner to change channels. Monitors often have higher display resolution than televisions.

The advent of display technology has paved the way for the continuous evolution of the monitor, whether it's for computers, television, mobile devices or any device that has a display. The current contenders for top-tier technology being used for display devices includes Super LCD 3 (SLCD3) and Super AMOLED. It should be noted that LED displays are actually just a kind of LCD display that use LED lights as backlight illumination.

3.2.7 Speaker

Most Important Speaker is required for this project because when we enter any alphabetical word or paragraph it convert it into sound which will come through the speaker of the system.

CHAPTER 4

RESULT AND DISCUSSION,PERFORMANCE ANALYSIS

4.1 Result & Discussion

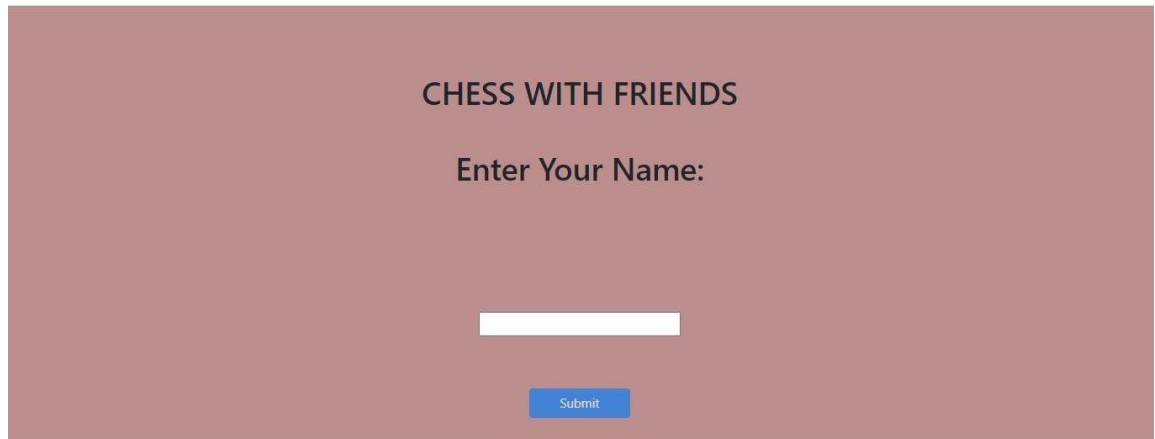
During the course of this research it was realized that building an chess application is a rigorous task. Consequently, a system that will providereal-time access of their moves and video chat to their opponents. The application has been successfully designed and developed to meet the requirements.

4.2 RECOMMENDATION

Recommendations The Chess With Friends meets a lot of expectations but would perform better if the following recommendations and suggestions are considered; i. System testing and maintenance should be performed regularly to avoid sudden system failures. ii. Updates and other modifications should be introduced with prior notice to users. iii. Administrator should be employed that can manage the interactions between rooms and game on one side and technicians on the other. This will aid ensure that proper support is provided for the system.

4.3 SCREENSHOTS

4.3.1 Input



CHESS WITH FRIENDS

Enter Your Name:

Submit

4.3.2 Output

Welcome to Chess with Friend!

Hey **sheshasena**, send the URL to your friend to start the game

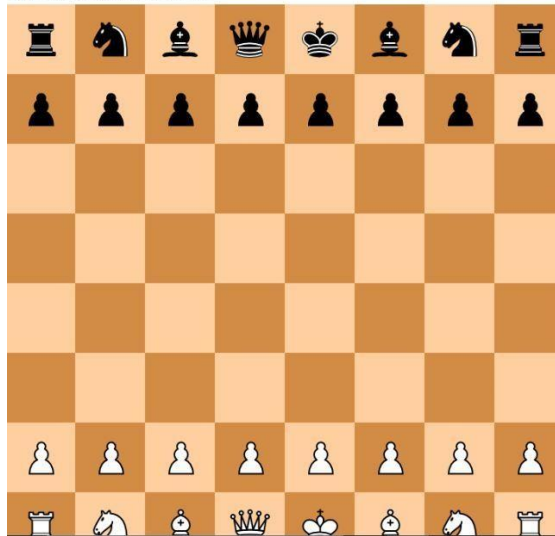
<http://localhost:3000/game/454cfa5a-c0fe-4a90-ae4e-3601f0735945>

Waiting for other opponent to join the game...

URL generated

Welcome to Chess with Friend!

Opponent: Random Name



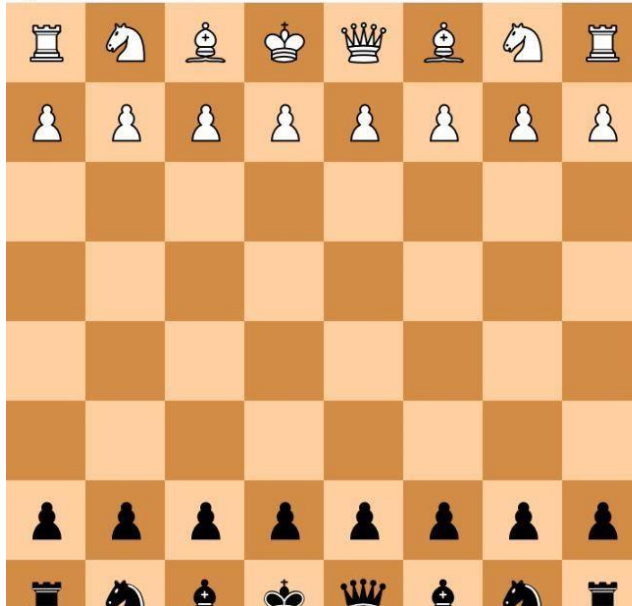
Video chat with your friend



Player 1 starts the Game

Welcome to Chess with Friend!

Opponent: sheshasena



sheshasena is video calling you

Accept



Video Chat Request

Welcome to Chess with Friend!

Opponent: Random Name



Video Chat Request Accepted by Player 2



Chess Game

CHAPTER 5

SUMMARY AND CONCLUSIONS

The main focus of this chapter is on the work has done and future enhancement of the project, the first section discuss the conclusion and summary of the whole Final Project and the second section is dedicated to the future enhancement of the project. Conclusion This Final Project aimed to create a practical and useable product, which can be considered as software and even an educational tool. A real life situation, which is a chess game in the current case, was modeled without any restrictions and based on its modelers personal understanding. Another aim of this Final Project was to analyze two agent base approaches. It was a fascinating and quite helpful experience for us to observe the differences between the two approaches by producing a practical work rather than conducting just a theoretical research. The users will ob-serve and realize the fact that it is a very open-ended model, allowing users to interact with it using their own imagination, which was the main idea behind all this effort. Hopefully, however, it has managed to lay the groundwork for further study of narrative in video games

REFERENCES

1. www.google.com
2. www.geeksforgeeks.org
3. www.wikipedia.org/html
4. www.wikipedia.org/css
5. www.wikipedia.org/javascript
6. www.wikipedia.org/react
7. www.tutorialspointsg.com
8. www.javaworld.com
9. www.angularjs.com
10. www.nodejs.com
11. www.github.com

APPENDIX

SOURCE CODE

- Chess.JS

```
import Chess from 'chess.js'
import ChessPiece from './chesspiece'
import Square from './square'
// when indexing, remember: [y][x].
/**
 * If the player color is black, make sure to invert the board.
 */

class Game {
  constructor(thisPlayersColorIsWhite) {
    this.thisPlayersColorIsWhite = thisPlayersColorIsWhite // once initialized, this value should
    never change.
    // console.log("this player's color is white: " + this.thisPlayersColorIsWhite)
    this.chessBoard = this.makeStartingBoard() // the actual chessBoard
    this.chess = new Chess()

    this.toCoord = thisPlayersColorIsWhite ? {
      0:8, 1:7, 2: 6, 3: 5, 4: 4, 5: 3, 6: 2, 7: 1
    } : {
      0:1, 1:2, 2: 3, 3: 4, 4: 5, 5: 6, 6: 7, 7: 8
    }

    this.toAlphabet = thisPlayersColorIsWhite ? {
      0:"a", 1:"b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 7: "h"
    } : {
      0:"h", 1:"g", 2: "f", 3: "e", 4: "d", 5: "c", 6: "b", 7: "a"
    }

    this.toCoord2 = thisPlayersColorIsWhite ? {
      8:0, 7:1, 6: 2, 5: 3, 4: 4, 3: 5, 2: 6, 1: 7
    } : {
      1:0, 2:1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 7
    }

    this.toAlphabet2 = thisPlayersColorIsWhite ? {
      "a":0, "b":1, "c":2, "d":3, "e":4, "f":5, "g":6, "h":7
    } : {
      "h":0, "g":1, "f":2, "e":3, "d":4, "c":5, "b":6, "a":7
    }

    this.nQueens = 1
  }

  getBoard() {
    return this.chessBoard
  }
}
```

```

}

// nextPlayersTurn() {
//   this.isWhitesTurn = !this.isWhitesTurn
// }

setBoard(newBoard) {
  this.chessBoard = newBoard
}

movePiece(pieceId, to, isMyMove) {

  const to2D = isMyMove ? {
    105:0, 195:1, 285: 2, 375: 3, 465: 4, 555: 5, 645: 6, 735: 7
  } : {
    105:7, 195:6, 285: 5, 375: 4, 465: 3, 555: 2, 645: 1, 735: 0
  }

  var currentBoard = this.getBoard()
  const pieceCoordinates = this.findPiece(currentBoard, pieceId)

  // can't find piece coordinates (piece doesn't exist on the board)
  if (!pieceCoordinates) {
    return
  }

  const y = pieceCoordinates[1]
  const x = pieceCoordinates[0]

  // new coordinates
  const to_y = to2D[to[1]]
  const to_x = to2D[to[0]]

  const originalPiece = currentBoard[y][x].getPiece()

  if (y === to_y && x === to_x) {
    return "moved in the same position."
  }

  /**
   * In order for this method to do anything meaningful,
   * the 'reassign const' line of code must run. Therefore,
   * for it to run, we must check first that the given move is valid.
   */

  const isPromotion = this.isPawnPromotion(to, pieceId[1])
  const moveAttempt = !isPromotion ? this.chess.move({
    from: this.toChessMove([x, y], to2D),
    to: this.toChessMove(to, to2D),
    piece: pieceId[1]})
    :
    this.chess.move({
      from: this.toChessMove([x, y], to2D),
      to: this.toChessMove(to, to2D),
      piece: pieceId[1],

```

```

        promotion: 'q'
    })

    console.log(moveAttempt)
    // console.log(isPromotion)

    if (moveAttempt === null) {
        return "invalid move"
    }

    if (moveAttempt.flags === 'e') {
        const move = moveAttempt.to
        const x = this.toAlphabet2[move[0]]
        let y
        if (moveAttempt.color === 'w') {
            y = parseInt(move[1], 10) - 1
        } else {
            y = parseInt(move[1], 10) + 1
        }
        currentBoard[this.toCoord2[y]][x].setPiece(null)
    }

    // Check castling
    const castle = this.isCastle(moveAttempt)
    if (castle.didCastle) {
        /**
         * The main thing we are doing here is moving the right rook
         * to the right position.
         *
         * - Get original piece by calling getPiece() on the original [x, y]
         * - Set the new [to_x, to_y] to the original piece
         * - Set the original [x, y] to null
         */

        const originalRook = currentBoard[castle.y][castle.x].getPiece()
        currentBoard[castle.to_y][castle.to_x].setPiece(originalRook)
        currentBoard[castle.y][castle.x].setPiece(null)
    }

    //___actually changing the board model___

    const reassign = isPromotion ? currentBoard[to_y][to_x].setPiece(
        new ChessPiece(
            'queen',
            false,
            pieceId[0] === 'w' ? 'white' : 'black',
            pieceId[0] === 'w' ? 'wq' + this.nQueens : 'bq' + this.nQueens))
        : currentBoard[to_y][to_x].setPiece(originalPiece)

    if (reassign !== "user tried to capture their own piece") {
        currentBoard[y][x].setPiece(null)
    } else {

```

```

        return reassign
    }

    //___actually changing the board model___

    const checkMate = this.chess.in_checkmate() ? " has been checkmated" : " has not been checkmated"
    console.log(this.chess.turn() + checkMate)
    if (checkMate === " has been checkmated") {
        return this.chess.turn() + checkMate
    }
    // changes the fill color of the opponent's king that is in check
    const check = this.chess.in_check() ? " is in check" : " is not in check"
    console.log(this.chess.turn() + check)
    if (check === " is in check") {
        return this.chess.turn() + check
    }

    console.log(currentBoard)
    // update board
    this.setBoard(currentBoard)
}

isCastle(moveAttempt) {
    /**
     * Assume moveAttempt is legal.
     *
     * {moveAttempt} -> {boolean x, y to_x, to_y}
     *
     * returns if a player has castled, the final position of
     * the rook (to_x, to_y), and the original position of the rook (x, y)
     */

    const piece = moveAttempt.piece
    const move = {from: moveAttempt.from, to: moveAttempt.to}

    const isBlackCastle = ((move.from === 'e1' && move.to === 'g1') || (move.from === 'e1' && move.to
    === 'c1'))
    const isWhiteCastle = (move.from === 'e8' && move.to === 'g8') || (move.from === 'e8' && move.to
    === 'c8')

    if (!(isWhiteCastle || isBlackCastle) || piece !== 'k') {
        return {
            didCastle: false
        }
    }

    let originalPositionOfRook
    let newPositionOfRook

    if ((move.from === 'e1' && move.to === 'g1')) {
        originalPositionOfRook = 'h1'
    }

```

```

        newPositionOfRook = 'f1'
    } else if ((move.from === 'e1' && move.to === 'c1')) {
        originalPositionOfRook = 'a1'
        newPositionOfRook = 'd1'
    } else if ((move.from === 'e8' && move.to === 'g8')) {
        originalPositionOfRook = 'h8'
        newPositionOfRook = 'f8'
    } else { // e8 to c8
        originalPositionOfRook = 'a8'
        newPositionOfRook = 'd8'
    }
}

return {
    didCastle: true,
    x: this.toAlphabet2[originalPositionOfRook[0]],
    y: this.toCoord2[originalPositionOfRook[1]],
    to_x: this.toAlphabet2[newPositionOfRook[0]],
    to_y: this.toCoord2[newPositionOfRook[1]]
}
}

isPawnPromotion(to, piece) {
    const res = piece === 'p' && (to[1] === 105 || to[1] === 735)
    if (res) {
        this.nQueens += 1
    }
    return res
}

toChessMove(finalPosition, to2D) {

    let move

    if (finalPosition[0] > 100) {
        move = this.toAlphabet[to2D[finalPosition[0]]] + this.toCoord[to2D[finalPosition[1]]]
    } else {
        move = this.toAlphabet[finalPosition[0]] + this.toCoord[finalPosition[1]]
    }

    // console.log("proposed move: " + move)
    return move
}

findPiece(board, pieceId) {
    // ChessBoard, String -> [Int, Int]
    // console.log("piecetofind: " + pieceId)
    for (var i = 0; i < 8; i++) {
        for (var j = 0; j < 8; j++) {
            if (board[i][j].getPieceIdOnThisSquare() === pieceId) {
                return [j, i]
            }
        }
    }
}
}

```

```

makeStartingBoard() {
  const backRank = ["rook", "knight", "bishop", "queen", "king", "bishop", "knight", "rook"]
  var startingChessBoard = []
  for (var i = 0; i < 8; i++) {
    startingChessBoard.push([])
    for (var j = 0; j < 8; j++) {
      // j is horizontal
      // i is vertical
      const coordinatesOnCanvas = [((j + 1) * 90 + 15), ((i + 1) * 90 + 15)]
      const emptySquare = new Square(j, i, null, coordinatesOnCanvas)

      startingChessBoard[i].push(emptySquare)
    }
  }
  const whiteBackRankId = ["wr1", "wn1", "wb1", "wq1", "wk1", "wb2", "wn2", "wr2"]
  const blackBackRankId = ["br1", "bn1", "bb1", "bq1", "bk1", "bb2", "bn2", "br2"]
  for (var j = 0; j < 8; j += 7) {
    for (var i = 0; i < 8; i++) {
      if (j == 0) {
        // top
        // console.log(backRank[i])
        startingChessBoard[j][this.thisPlayersColorIsWhite ? i : 7 - i].setPiece(new
ChessPiece(backRank[i], false, this.thisPlayersColorIsWhite ? "black" : "white",
this.thisPlayersColorIsWhite ? blackBackRankId[i] : whiteBackRankId[i]))
        startingChessBoard[j + 1][this.thisPlayersColorIsWhite ? i : 7 - i].setPiece(new
ChessPiece("pawn", false, this.thisPlayersColorIsWhite ? "black" : "white", this.thisPlayersColorIsWhite
? "bp" + i : "wp" + i))
      } else {
        // bottom
        startingChessBoard[j - 1][this.thisPlayersColorIsWhite ? i : 7 - i].setPiece(new
ChessPiece("pawn", false, this.thisPlayersColorIsWhite ? "white" : "black", this.thisPlayersColorIsWhite
? "wp" + i : "bp" + i))
        startingChessBoard[j][this.thisPlayersColorIsWhite ? i : 7 - i].setPiece(new
ChessPiece(backRank[i], false, this.thisPlayersColorIsWhite ? "white" : "black",
this.thisPlayersColorIsWhite ? whiteBackRankId[i] : blackBackRankId[i]))
      }
    }
  }
  return startingChessBoard
}
}

export default Game

```

-Chesspiece.JS

```
class ChessPiece {
  constructor(name, isAttacked, color, id) {
    this.name = name // string
    this.isAttacked = isAttacked // boolean
    this.color = color // string
    this.id = id // string
  }

  setSquare(newSquare) {
    // set the square this piece is sitting top of.
    // on any given piece (on the board), there will always be a piece on top of it.
    // console.log(newSquare)
    if (newSquare === undefined) {
      this.squareThisPieceIsOn = newSquare
      return
    }

    if (this.squareThisPieceIsOn === undefined) {
      this.squareThisPieceIsOn = newSquare
      newSquare.setPiece(this)
    }

    const isNewSquareDifferent = this.squareThisPieceIsOn.x !== newSquare.x ||
this.squareThisPieceIsOn.y !== newSquare.y

    if (isNewSquareDifferent) {
      // console.log("set")
      this.squareThisPieceIsOn = newSquare
      newSquare.setPiece(this)
    }
  }

  getSquare() {
    return this.squareThisPieceIsOn
  }
}

export default ChessPiece
```

-Square.JS

```
class Square {
  constructor(x, y, pieceOnThisSquare, canvasCoord) {
    this.x = x // Int 0 < x < 7
    this.y = y // Int 0 < y < 7
    this.canvasCoord = canvasCoord
    this.pieceOnThisSquare = pieceOnThisSquare // ChessPiece || null
  }

  setPiece(newPiece) {
    if (newPiece === null && this.pieceOnThisSquare === null) {
```

```

        return
    } else if (newPiece === null) {
        // case where the function caller wants to remove the piece that is on this square.
        this.pieceOnThisSquare.setSquare(undefined)
        this.pieceOnThisSquare = null
    } else if (this.pieceOnThisSquare === null) {
        // case where the function caller wants assign a new piece on this square
        this.pieceOnThisSquare = newPiece
        newPiece.setSquare(this)
    } else if (this.getPieceIdOnThisSquare() !== newPiece.id && this.pieceOnThisSquare.color !==
newPiece.color) {
        // case where the function caller wants to change the piece on this square. (only different
color allowed)
        console.log("capture!")
        this.pieceOnThisSquare = newPiece
        newPiece.setSquare(this)
    } else {
        return "user tried to capture their own piece"
    }
}

removePiece() {
    this.pieceOnThisSquare = null
}

getPiece() {
    return this.pieceOnThisSquare
}

getPieceIdOnThisSquare() {
    if (this.pieceOnThisSquare === null) {
        return "empty"
    }
    return this.pieceOnThisSquare.id
}

isOccupied() {
    return this.pieceOnThisSquare !== null
}

getCoord() {
    return [this.x, this.y]
}

getCanvasCoord() {
    return this.canvasCoord
}
}

export default Square

```

-Chessgame.JS

```

import React from 'react'
import Game from '../model/chess'
import Square from '../model/square'
import { Stage, Layer } from 'react-konva';

```

```

import Board from '../assets/chessBoard.png'
import useSound from 'use-sound'
import chessMove from '../assets/moveSoundEffect.mp3'
import Piece from './piece'
import piecemap from './piecemap'
import { useParams } from 'react-router-dom'
import { ColorContext } from '../../context/colorcontext'
import VideoChatApp from '../../connection/videochat'
const socket = require('../../connection/socket').socket

class ChessGame extends React.Component {

  state = {
    gameState: new Game(this.props.color),
    draggedPieceTargetId: "", // empty string means no piece is being dragged
    playerTurnToMoveIsWhite: true,
    whiteKingInCheck: false,
    blackKingInCheck: false
  }

  componentDidMount() {
    console.log(this.props.myUserName)
    console.log(this.props.opponentUserName)
    // register event listeners
    socket.on('opponent move', move => {
      // move == [pieceId, finalPosition]
      // console.log("opponent's move: " + move.selectedId + ", " + move.finalPosition)
      if (move.playerColorThatJustMovedIsWhite !== this.props.color) {
        this.movePiece(move.selectedId, move.finalPosition, this.state.gameState, false)
        this.setState({
          playerTurnToMoveIsWhite: !move.playerColorThatJustMovedIsWhite
        })
      }
    })
  }

  startDragging = (e) => {
    this.setState({
      draggedPieceTargetId: e.target.attrs.id
    })
  }

  movePiece = (selectedId, finalPosition, currentGame, isMyMove) => {
    /**
     * "update" is the connection between the model and the UI.
     * This could also be an HTTP request and the "update" could be the server response.
     * (model is hosted on the server instead of the browser)
     */
    var whiteKingInCheck = false
    var blackKingInCheck = false
    var blackCheckmated = false
    var whiteCheckmated = false
    const update = currentGame.movePiece(selectedId, finalPosition, isMyMove)
  }
}

```

```

    if (update === "moved in the same position.") {
      this.revertToPreviousState(selectedId) // pass in selected ID to identify the piece that
messed up
      return
    } else if (update === "user tried to capture their own piece") {
      this.revertToPreviousState(selectedId)
      return
    } else if (update === "b is in check" || update === "w is in check") {
      // change the fill of the enemy king or your king based on which side is in check.
      // play a sound or something
      if (update[0] === "b") {
        blackKingInCheck = true
      } else {
        whiteKingInCheck = true
      }
    } else if (update === "b has been checkmated" || update === "w has been checkmated") {
      if (update[0] === "b") {
        blackCheckmated = true
      } else {
        whiteCheckmated = true
      }
    } else if (update === "invalid move") {
      this.revertToPreviousState(selectedId)
      return
    }
  }

  // let the server and the other client know your move
  if (isMyMove) {
    socket.emit('new move', {
      nextPlayerColorToMove: !this.state.gameState.thisPlayersColorIsWhite,
      playerColorThatJustMovedIsWhite: this.state.gameState.thisPlayersColorIsWhite,
      selectedId: selectedId,
      finalPosition: finalPosition,
      gameId: this.props.gameId
    })
  }

  this.props.playAudio()

  // sets the new game state.
  this.setState({
    draggedPieceTargetId: "",
    gameState: currentGame,
    playerTurnToMoveIsWhite: !this.props.color,
    whiteKingInCheck: whiteKingInCheck,
    blackKingInCheck: blackKingInCheck
  })

  if (blackCheckmated) {
    alert("WHITE WON BY CHECKMATE!")
  } else if (whiteCheckmated) {
    alert("BLACK WON BY CHECKMATE!")
  }
}

```

```

endDragging = (e) => {
  const currentGame = this.state.gameState
  const currentBoard = currentGame.getBoard()
  const finalPosition = this.inferCoord(e.target.x() + 90, e.target.y() + 90, currentBoard)
  const selectedId = this.state.draggedPieceTargetId
  this.movePiece(selectedId, finalPosition, currentGame, true)
}

revertToPreviousState = (selectedId) => {
  /**
   * Should update the UI to what the board looked like before.
   */
  const oldGS = this.state.gameState
  const oldBoard = oldGS.getBoard()
  const tmpGS = new Game(true)
  const tmpBoard = []

  for (var i = 0; i < 8; i++) {
    tmpBoard.push([])
    for (var j = 0; j < 8; j++) {
      if (oldBoard[i][j].getPieceIdOnThisSquare() === selectedId) {
        tmpBoard[i].push(new Square(j, i, null, oldBoard[i][j].canvasCoord))
      } else {
        tmpBoard[i].push(oldBoard[i][j])
      }
    }
  }

  // temporarily remove the piece that was just moved
  tmpGS.setBoard(tmpBoard)

  this.setState({
    gameState: tmpGS,
    draggedPieceTargetId: "",
  })

  this.setState({
    gameState: oldGS,
  })
}

inferCoord = (x, y, chessBoard) => {
  // console.log("actual mouse coordinates: " + x + ", " + y)
  /**
   * Should give the closest estimate for new position.
   */
  var hashmap = {}
  var shortestDistance = Infinity
  for (var i = 0; i < 8; i++) {
    for (var j = 0; j < 8; j++) {
      const canvasCoord = chessBoard[i][j].getCanvasCoord()
      // calculate distance
      const delta_x = canvasCoord[0] - x
      const delta_y = canvasCoord[1] - y
      const newDistance = Math.sqrt(delta_x**2 + delta_y**2)
      hashmap[newDistance] = canvasCoord
    }
  }

```

```

        if (newDistance < shortestDistance) {
            shortestDistance = newDistance
        }
    }
}

return hashmap[shortestDistance]
}

render() {
    // console.log(this.state.gameState.getBoard())
    // console.log("it's white's move this time: " + this.state.playerTurnToMoveIsWhite)
    /*
        Look at the current game state in the model and populate the UI accordingly
    */
    // console.log(this.state.gameState.getBoard())

    return (
        <React.Fragment>
        <div style = {{
            backgroundImage: `url(${Board})`,
            width: "720px",
            height: "720px"}}
        >
            <Stage width = {720} height = {720}>
                <Layer>
                    {this.state.gameState.getBoard().map((row) => {
                        return (<React.Fragment>
                            {row.map((square) => {
                                if (square.isOccupied()) {
                                    return (
                                        <Piece
                                            x = {square.getCanvasCoord()[0]}
                                            y = {square.getCanvasCoord()[1]}
                                            imgurls = {piecemap[square.getPiece().name]}
                                            isWhite = {square.getPiece().color === "white"}
                                            draggedPieceTargetId = {this.state.draggedPieceTargetId}
                                            onDragStart = {this.startDragging}
                                            onDragEnd = {this.endDragging}
                                            id = {square.getPieceIdOnThisSquare()}
                                            thisPlayersColorIsWhite = {this.props.color}
                                            playerTurnToMoveIsWhite =
{this.state.playerTurnToMoveIsWhite}

                                            whiteKingInCheck = {this.state.whiteKingInCheck}
                                            blackKingInCheck = {this.state.blackKingInCheck}
                                        />)
                                    }
                                }
                                return
                            })}
                        </React.Fragment>)
                    }}
                </Layer>
            </Stage>
        </div>
    </React.Fragment>)
    }
}

```

```

const ChessGameWrapper = (props) => {
  /**
   * player 1
   *   - socketId 1
   *   - socketId 2 ???
   * player 2
   *   - socketId 2
   *   - socketId 1
   */

  // get the gameId from the URL here and pass it to the chessGame component as a prop.
  const domainName = 'http://localhost:3000'
  const color = React.useContext(ColorContext)
  const { gameId } = useParams()
  const [play] = useSound(chessMove);
  const [opponentSocketId, setOpponentSocketId] = React.useState('')
  const [opponentDidJoinTheGame, didJoinGame] = React.useState(false)
  const [opponentUserName, setUserName] = React.useState('')
  const [gameSessionDoesNotExist, doesntExist] = React.useState(false)

  React.useEffect(() => {
    socket.on("playerJoinedRoom", statusUpdate => {
      console.log("A new player has joined the room! Username: " + statusUpdate.userName + ", Game
id: " + statusUpdate.gameId + " Socket id: " + statusUpdate.mySocketId)
      if (socket.id !== statusUpdate.mySocketId) {
        setOpponentSocketId(statusUpdate.mySocketId)
      }
    })

    socket.on("status", statusUpdate => {
      console.log(statusUpdate)
      alert(statusUpdate)
      if (statusUpdate === 'This game session does not exist.' || statusUpdate === 'There are
already 2 people playing in this room.') {
        doesntExist(true)
      }
    })

    socket.on('start game', (opponentUserName) => {
      console.log("START!")
      if (opponentUserName !== props.myUserName) {
        setUserName(opponentUserName)
        didJoinGame(true)
      } else {
        // in chessGame, pass opponentUserName as a prop and label it as the enemy.
        // in chessGame, use reactContext to get your own userName
        // socket.emit('myUserName')
        socket.emit('request username', gameId)
      }
    })
  })
}

```

```

socket.on('give userName', (socketId) => {
  if (socket.id !== socketId) {
    console.log("give userName stage: " + props.myUserName)
    socket.emit('recieved userName', {userName: props.myUserName, gameId: gameId})
  }
})

socket.on('get Opponent UserName', (data) => {
  if (socket.id !== data.socketId) {
    setUserName(data.userName)
    console.log('data.socketId: data.socketId')
    setOpponentSocketId(data.socketId)
    didJoinGame(true)
  }
})
}, [])

return (
  <React.Fragment>
    {opponentDidJoinTheGame ? (
      <div>
        <h4> Opponent: {opponentUserName} </h4>
        <div style={{ display: "flex" }}>
          <ChessGame
            playAudio={play}
            gameId={gameid}
            color={color.didRedirect}
          />
          <VideoChatApp
            mySocketId={socket.id}
            opponentSocketId={opponentSocketId}
            myUserName={props.myUserName}
            opponentUserName={opponentUserName}
          />
        </div>
        <h4> You: {props.myUserName} </h4>
      </div>
    ) : gameSessionDoesNotExist ? (
      <div>
        <h1 style={{ textAlign: "center", marginTop: "200px" }}> :( </h1>
      </div>
    ) : (
      <div>
        <h1
          style={{
            textAlign: "center",
            marginTop: String(window.innerHeight / 8) + "px",
          }}
        >
          Hey <strong>{props.myUserName}</strong>, send the URL to your friend to start the game
        </h1>
        <textarea
          style={{ marginLeft: String((window.innerWidth / 2) - 290) + "px", marginTop: "30" + "px",
            width: "580px", height: "30px"}}
          onFocus={(event) => {

```

```

        console.log('sd')
        event.target.select()
    }}
    value = {domainName + "/game/" + gameid}
    type = "text">
</textarea>
<br><br>

<h1 style={{ textAlign: "center", marginTop: "100px" }}>
    {" "}
    Waiting for other opponent to join the game...{" "}
</h1>
</div>
    })
  </React.Fragment>
);
};

export default ChessGameWrapper

```

-Piece.JS

```

import React from 'react'
import { Image } from 'react-konva';
import useImage from 'use-image'

const Piece = (props) => {
  const choiceOfColor = props.isWhite ? 0 : 1
  const [image] = useImage(props.imgurls[choiceOfColor]);
  const isDragged = props.id === props.draggedPieceTargetId

  const canThisPieceEvenBeMovedByThisPlayer = props.isWhite === props.thisPlayersColorIsWhite
  const isItThatPlayersTurn = props.playerTurnToMoveIsWhite === props.thisPlayersColorIsWhite

  const thisWhiteKingInCheck = props.id === "wk1" && props.whiteKingInCheck
  const thisBlackKingInCheck = props.id === "bk1" && props.blackKingInCheck

  // console.log("this piece ID:" + props.thisPieceTargetId)
  // console.log("dragged piece ID:" + props.draggedPieceTargetId)
  return <Image image={image}
    x = {props.x - 90}
    y = {props.y - 90}
    draggable = {canThisPieceEvenBeMovedByThisPlayer && isItThatPlayersTurn}
    width = {isDragged ? 75 : 60}
    height = {isDragged ? 75 : 60}
    onDragStart = {props.onDragStart}
    onDragEnd = {props.onDragEnd}
    fill = {(thisWhiteKingInCheck && "red") || (thisBlackKingInCheck && "red")}
    id = {props.id}
  />;
};

export default Piece

```

-Socket.JS

```
import io from 'socket.io-client'

const URL = 'http://localhost:8000'

const socket = io(URL)

var mySocketId
// register preliminary event listeners here:

socket.on("createNewGame", statusUpdate => {
  console.log("A new game has been created! Username: " + statusUpdate.userName + ", Game id: " +
statusUpdate.gameId + " Socket id: " + statusUpdate.mySocketId)
  mySocketId = statusUpdate.mySocketId
})

export {
  socket,
  mySocketId
}
```

-Videochat.JS

```
import React, { useEffect, useState, useRef } from 'react';
import Peer from "simple-peer";
import styled from "styled-components";
const socket = require('../connection/socket').socket

const Container = styled.div`
  height: 100vh;
  width: 100%;
  flex-direction: column;
`;

const Row = styled.div`
  width: 100%;
`;

const Video = styled.video`
  border: 1px solid blue;
`;

function VideoChatApp(props) {
  /**
   * initial state: both player is neutral and have the option to call each other
   *
   * player 1 calls player 2: Player 1 should display: 'Calling {player 2 username},' and the
   * 'CallPeer' button should disappear for Player 1.
   * Player 2 should display '{player 1 username} is calling you' and
   * the 'CallPeer' button for Player 2 should also disappear.
   *
   * Case 1: player 2 accepts call - the video chat begins and there is no button to end it.
   */
}
```

```

*
* Case 2: player 2 ignores player 1 call - nothing happens. Wait until the connection times out.
*
*/

const [stream, setStream] = useState();
const [receivingCall, setReceivingCall] = useState(false);
const [caller, setCaller] = useState("");
const [callerSignal, setCallerSignal] = useState();
const [callAccepted, setCallAccepted] = useState(false);
const [isCalling, setIsCalling] = useState(false)
const userVideo = useRef();
const partnerVideo = useRef();

useEffect(() => {
  navigator.mediaDevices.getUserMedia({ video: true, audio: true }).then(stream => {
    setStream(stream);
    if (userVideo.current) {
      userVideo.current.srcObject = stream;
    }
  })

  socket.on("hey", (data) => {
    setReceivingCall(true);
    setCaller(data.from);
    setCallerSignal(data.signal);
  })
}, []);

function callPeer(id) {
  setIsCalling(true)
  const peer = new Peer({
    initiator: true,
    trickle: false,
    stream: stream,
  });

  peer.on("signal", data => {
    socket.emit("callUser", { userToCall: id, signalData: data, from: props.mySocketId})
  })

  peer.on("stream", stream => {
    if (partnerVideo.current) {
      partnerVideo.current.srcObject = stream;
    }
  });

  socket.on("callAccepted", signal => {
    setCallAccepted(true);
    peer.signal(signal);
  })
}

function acceptCall() {
  setCallAccepted(true);
  setIsCalling(false)
}

```

```

const peer = new Peer({
  initiator: false,
  trickle: false,
  stream: stream,
});
peer.on("signal", data => {
  socket.emit("acceptCall", { signal: data, to: caller })
})

peer.on("stream", stream => {
  partnerVideo.current.srcObject = stream;
});

peer.signal(callerSignal);
}

let UserVideo;
if (stream) {
  UserVideo = (
    <Video playsInline muted ref={userVideo} autoPlay style = {{width: "50%", height: "50%"}} />
  );
}

let mainView;

if (callAccepted) {
  mainView = (
    <Video playsInline ref={partnerVideo} autoPlay style = {{width: "100%", height: "100%"}} />
  );
} else if (receivingCall) {
  mainView = (
    <div>
      <h1>{props.opponentUserName} is video calling you</h1>
      <button onClick={acceptCall}><h1>Accept</h1></button>
    </div>
  )
} else if (isCalling) {
  mainView = (
    <div>
      <h1>Currently calling {props.opponentUserName}...</h1>
    </div>
  )
} else {
  mainView = (
    <button onClick = {() => {
      callPeer(props.opponentSocketId)
    }}><h1>Video chat with your friend</h1></button>
  )
}

return (<Container>
  <Row>
    {mainView}
    {UserVideo}
  </Row>

```

```

    </Container>);
  }

export default VideoChatApp;

```

-JoinGame.JS

```

import React from 'react'
import { useParams } from 'react-router-dom'
const socket = require('../connection/socket').socket

/**
 * 'Join game' is where we actually join the game room.
 */

const JoinGameRoom = (gameid, userName, isCreator) => {
  /**
   * For this browser instance, we want
   * to join it to a gameRoom. For now
   * assume that the game room exists
   * on the backend.
   *
   *
   * TODO: handle the case when the game room doesn't exist.
   */
  const idData = {
    gameId : gameid,
    userName : userName,
    isCreator: isCreator
  }
  socket.emit("playerJoinGame", idData)
}

const JoinGame = (props) => {
  /**
   * Extract the 'gameId' from the URL.
   * the 'gameId' is the gameRoom ID.
   */
  const { gameId } = useParams()
  JoinGameRoom(gameid, props.userName, props.isCreator)
  return <div>
    <h1 style = {{textAlign: "center" }}><strong>Welcome to Chess with Friend!</strong></h1>
  </div>
}

export default JoinGame

```

-JoinRoom.JS

```

import React from 'react'
import JoinGame from './joingame'
import ChessGame from '../chess/ui/chessgame'

/**
 * Onboard is where we create the game room.
 */

class JoinRoom extends React.Component {
  state = {
    didGetUserName: false,
    inputText: ""
  }

  constructor(props) {
    super(props);
    this.textArea = React.createRef();
  }

  typingUserName = () => {
    // grab the input text from the field from the DOM
    const typedText = this.textArea.current.value

    // set the state with that text
    this.setState({
      inputText: typedText
    })
  }

  render() {
    return (<React.Fragment>
      {
        this.state.didGetUserName ?
        <React.Fragment>
          <JoinGame userName = {this.state.inputText} isCreator = {false}/>
          <ChessGame myUserName = {this.state.inputText}/>
        </React.Fragment>
        :
        <div style={{backgroundColor:"white"}} >
          <h1 style={{textAlign: "center", marginTop: String((window.innerHeight / 3)) +
"px"}}>CHESS WITH FRIENDS <br></br><br></br>Enter Your Name:</h1>

          <input style={{marginLeft: String((window.innerWidth / 2) - 120) + "px", width:
"240px", marginTop: "62px"}}
            ref = {this.textArea}
            onInput = {this.typingUserName}></input>

          <button className="btn btn-primary"
            style = {{marginLeft: String((window.innerWidth / 2) - 60) + "px", width:
"120px", marginTop: "62px"}}
            disabled = {!(this.state.inputText.length > 0)}
            onClick = {() => {
              // When the 'Submit' button gets pressed from the username screen,
              // We should send a request to the server to create a new room with
              // the uuid we generate here.

```

```

        this.setState({
          didGetUserName: true
        })
      }}>Submit</button>
    </div>
  }
</React.Fragment>)
}
}

export default JoinRoom

```

-ObBoard.JS

```

import React from 'react'
import { Redirect } from 'react-router-dom'
import uuid from 'uuid/v4'
import { ColorContext } from '../context/colorcontext'
import './style.css'
const socket = require('../connection/socket').socket

/**
 * Onboard is where we create the game room.
 */

class CreateNewGame extends React.Component {
  state = {
    didGetUserName: false,
    inputText: '',
    gameId: ''
  }

  constructor(props) {
    super(props);
    this.textArea = React.createRef();
  }

  send = () => {
    /**
     * This method should create a new room in the '/' namespace
     * with a unique identifier.
     */
    const newGameRoomId = uuid()

    // set the state of this component with the gameId so that we can
    // redirect the user to that URL later.
    this.setState({
      gameId: newGameRoomId
    })

    // emit an event to the server to create a new room
    socket.emit('createNewGame', newGameRoomId)
  }

  typingUserName = () => {

```

```

    // grab the input text from the field from the DOM
    const typedText = this.textArea.current.value

    // set the state with that text
    this.setState({
      inputText: typedText
    })
  }

  render() {
    // !!! TODO: edit this later once you have bought your own domain.

    return ( <div className="backk" >
      {
        this.state.didGetUserName ?

          <Redirect to = {"/game/" + this.state.gameId}><button className="btn btn-success" style =
{{marginLeft: String((window.innerWidth / 2) - 60) + "px", width: "120px"}}>Start
Game</button></Redirect>

          :

          <div>
            <h1></h1>
            <h1 style={{textAlign: "center", top:"100px", padding:"80px", marginTop:
String((window.innerHeight / 3)) + "px"}}>CHESS WITH FRIENDS <br></br><br></br>Enter Your Name:</h1>

            <input style={{marginLeft: String((window.innerWidth / 2) - 120) + "px", width:
"240px", marginTop: "62px"}}
              ref = {this.textArea}
              onInput = {this.typingUserName}></input>

            <button className="btn btn-primary"
              style = {{marginLeft: String((window.innerWidth / 2) - 60) + "px", width:
"120px", marginTop: "62px"}}
              disabled = {!(this.state.inputText.length > 0)}
              onClick = {() => {
                // When the 'Submit' button gets pressed from the username screen,
                // We should send a request to the server to create a new room with
                // the uuid we generate here.
                this.props.didRedirect()
                this.props.setUserName(this.state.inputText)
                this.setState({
                  didGetUserName: true
                })
                this.send()
              }}>Submit</button>

          </div>
        }
      </div> )
    }
  }

const Onboard = (props) => {
  const color = React.useContext(ColorContext)

  return <CreateNewGame didRedirect = {color.playerDidRedirect} setUserName = {props.setUserName}/>

```

```
}

export default Onboard
```

-App.JS

```
import React from 'react';
import { BrowserRouter as Router, Route, Redirect, Switch } from 'react-router-dom';
import JoinRoom from './onboard/joinroom'
import { ColorContext } from './context/colorcontext'
import Onboard from './onboard/onboard'
import JoinGame from './onboard/joingame'
import ChessGame from './chess/ui/chessgame'
/*
 * Frontend flow:
 *
 * 1. user first opens this app in the browser.
 * 2. a screen appears asking the user to send their friend their game URL to start the game.
 * 3. the user sends their friend their game URL
 * 4. the user clicks the 'start' button and waits for the other player to join.
 * 5. As soon as the other player joins, the game starts.
 *
 * Other player flow:
 * 1. user gets the link sent by their friend
 * 2. user clicks on the link and it redirects to their game. If the 'host' has not yet
 *    clicked the 'start' button yet, the user will wait for when the host clicks the start button.
 *    If the host decides to leave before they click on the "start" button, the user will be notified
 *    that the host has ended the session.
 * 3. Once the host clicks the start button or the start button was already clicked on
 *    before, that's when the game starts.
 * Onboarding screen =====> Game start.
 *
 * Every time a user opens our site from the '/' path, a new game instance is automatically created
 * on the back-end. We should generate the uuid on the frontend, send the request with the uuid
 * as a part of the body of the request. If any player leaves, then the other player wins automatically.
 *
 */

function App() {

  const [didRedirect, setDidRedirect] = React.useState(false)

  const playerDidRedirect = React.useCallback(() => {
    setDidRedirect(true)
  }, [])

  const playerDidNotRedirect = React.useCallback(() => {
    setDidRedirect(false)
  }, [])

  const [userName, setUserName] = React.useState('')
```

```

    return (
      <ColorContext.Provider value = {{didRedirect: didRedirect, playerDidRedirect: playerDidRedirect,
playerDidNotRedirect: playerDidNotRedirect}}>
        <Router>
          <Switch>
            <Route path = "/" exact>
              <Onboard setUserName = {setUserName}/>
            </Route>
            <Route path = "/game/:gameid" exact>
              {didRedirect ?
                <React.Fragment>
                  <JoinGame userName = {userName} isCreator = {true} />
                  <ChessGame myUserName = {userName} />
                </React.Fragment>
                :
                <JoinRoom />}
            </Route>
            <Redirect to = "/" />
          </Switch>
        </Router>
      </ColorContext.Provider>);
    }
  }

export default App;

```

Backend- AppLogic.JS

```

var io
var gameSocket
// gamesInSession stores an array of all active socket connections
var gamesInSession = []

const initializeGame = (sio, socket) => {

  io = sio
  gameSocket = socket

  gamesInSession.push(gameSocket)

  gameSocket.on("disconnect", onDisconnect)

  gameSocket.on("new move", newMove)

  gameSocket.on("createNewGame", createNewGame)

  gameSocket.on("playerJoinGame", playerJoinsGame)

```

```

gameSocket.on('request username', requestUserName)

gameSocket.on('recieved userName', recievedUserName)

videoChatBackend()
}

function videoChatBackend() {
  // main function listeners
  gameSocket.on("callUser", (data) => {
    io.to(data.userToCall).emit('hey', {signal: data.signalData, from: data.from});
  })

  gameSocket.on("acceptCall", (data) => {
    io.to(data.to).emit('callAccepted', data.signal);
  })
}

function playerJoinsGame(idData) {
  /**
   * Joins the given socket to a session with it's gameId
   */

  // A reference to the player's Socket.IO socket object
  var sock = this

  // Look up the room ID in the Socket.IO manager object.
  var room = io.sockets.adapter.rooms[idData.gameId]
  // console.log(room)

  // If the room exists...
  if (room === undefined) {
    this.emit('status' , "This game session does not exist." );
    return
  }
  if (room.length < 2) {
    // attach the socket id to the data object.
    idData.mySocketId = sock.id;

    // Join the room
    sock.join(idData.gameId);

    console.log(room.length)

    if (room.length === 2) {
      io.sockets.in(idData.gameId).emit('start game', idData.userName)
    }

    // Emit an event notifying the clients that the player has joined the room.
    io.sockets.in(idData.gameId).emit('playerJoinedRoom', idData);

  } else {

```

```

        // Otherwise, send an error message back to the player.
        this.emit('status' , "There are already 2 people playing in this room." );
    }
}

function createNewGame(gameId) {
    // Return the Room ID (gameId) and the socket ID (mySocketId) to the browser client
    this.emit('createNewGame', {gameId: gameId, mySocketId: this.id});

    // Join the Room and wait for the other player
    this.join(gameId)
}

function newMove(move) {
    /**
     * First, we need to get the room ID in which to send this message.
     * Next, we actually send this message to everyone except the sender
     * in this room.
     */

    const gameId = move.gameId

    io.to(gameId).emit('opponent move', move);
}

function onDisconnect() {
    var i = gamesInSession.indexOf(gameSocket);
    gamesInSession.splice(i, 1);
}

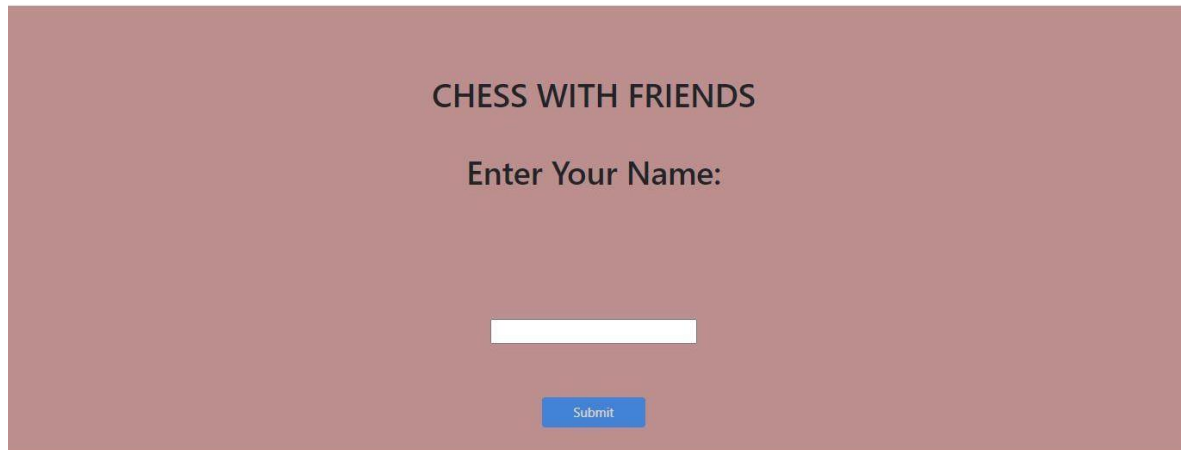
function requestUserName(gameId) {
    io.to(gameId).emit('give userName', this.id);
}

function recievedUserName(data) {
    data.socketId = this.id
    io.to(data.gameId).emit('get Opponent UserName', data);
}

exports.initializeGame = initializeGame

```

SAMPLE SCREENS



CHESS WITH FRIENDS

Enter Your Name:

Submit

Welcome to Chess with Friend!

Hey **sheshasena**, send the URL to your friend to start the game

Waiting for other opponent to join the game...

