



# **DYNAMIC BOARD GAME WEB APPLICATION DEVELOPMENT WITH ENHANCED FEATURES**

## **A PROJECT REPORT**

*Submitted by*

<b>SHALINI.B</b>	<b>(Reg.No.1920106087)</b>
<b>THANUSUYA SRI.S</b>	<b>(Reg.No.1920106099)</b>
<b>VARSHAA.M. J</b>	<b>(Reg.No.1920106102)</b>

*in partial fulfillment for the award of the degree  
of*

**BACHELOR OF TECHNOLOGY  
IN  
INFORMATION TECHNOLOGY  
SONA COLLEGE OF TECHNOLOGY**

**ANNA UNIVERSITY: CHENNAI 600 025**

**NOVEMBER 2023**

**ANNA UNIVERSITY : CHENNAI 600 025**

**BONAFIDE CERTIFICATE**

Certified that this project report “**DYNAMIC BOARD GAME WEB APPLICATION DEVELOPMENT WITH ENHANCED FEATURES**” is the bonafide work of “**SHALINI.B (1920106087), THANUSUYA SRI.S (1920106099), VARSHAA.M.J (1920106102)**” who carried out the project work under my supervision.

**SIGNATURE**

(Dr.J.Akilandeswari)

Professor

**HEAD OF THE DEPARTMENT**

Department of Information Technology

Sona College of Technology

Salem-636 005

**SIGNATURE**

(Mr.K.Thangaraj)

Associate Professor

**SUPERVISOR**

Department of Information Technology

Sona College of Technology

Salem-636 005

Submitted for Mini Project viva voice examination held on .....

**INTERNAL EXAMINER**

**EXTERNAL EXAMINER**

## **ABSTRACT**

In the realm of online gaming, this project introduces a novel 9x9 chess board platform, deviating from the conventional 8x8 standard to provide players with an expanded and enriched gaming experience. This project delves into the domain of online gaming, presenting a web-based board game application centered around a 9x9 board. The primary goal is to cultivate an engaging online gaming experience, incorporating features that not only enable real-time gameplay but also facilitate movement tracking and user interaction. Leveraging the capabilities of React JS, the system constructs a dynamic board game, recording player movements in a dedicated online MongoDB Atlas database. Unique to this application are additional functionalities, including a coin movement track sheet, timed game sessions, and robust movement validation with associated alerts. The envisioned system aims to elevate the overall gaming experience, providing players with a seamless, feature-rich platform for enjoying and preserving their gameplay history.

## ACKNOWLEDGEMENT

First and foremost, we thank to **power of almighty** for showing us inner peace and for all blessings. Special gratitude to our parents, for showing their support and love always.

We like to acknowledge the constant support provided by **Sri.C.Valliappa**, Chairman, for his consistent motivation in pursuing my project.

Our gratitude thanks to our Vice Chairmen **Sri. Chocko Valliappa** and **Sri. Thyagu Valliappa** who leads us in a narrow path towards success in all the way.

We are immensely grateful to our principal **Dr.V.Jayaprakash,M.E,Ph.D**, who has been our constant source of inspiration.

We express our sincere thanks to the Head of Information Technology, **Dr.J.Akilandeswari,M.E,Ph.D**, for providing adequate laboratory facilities to complete this thesis.

We feel elated to keep on record our heartfelt thanks and gratitude to our project guide **Mr.K.Thangaraj,M.E,(Ph.D)**, Associate Professor our steadfast inspiration, for his valuable guidance, untiring patience and diligent encouragement during the entire span of this project.

We feel proud in sharing this success with my staff members, non-teaching staffs and friends who helped directly or indirectly in completing this project successfully.

## **TABLE OF CONTENTS**

<b>CHAPTER NO.</b>	<b>TITLE</b>	<b>PAGE NO.</b>
	<b>ABSTRACT</b>	<b>iii</b>
	<b>LIST OF FIGURES</b>	<b>vii</b>
	<b>LIST OF SYMBOLS AND ABBREVIATIONS</b>	<b>viii</b>
<b>1.</b>	<b>INTRODUCTION</b>	<b>1</b>
	1.1 Conventional Board Games	1
	1.1.1 Standard 8x8 Chess Board	1
	1.1.2 Initial Origin	2
	1.1.3 Characteristics of 8x8	2
	1.1.4 Architecture	4
	1.1.5 Components	4
	1.1.6 Applications	6
	1.2 Variations	6
	1.2.1 Courier Chess	7
	1.2.2 Tamerlane Chess	7
	1.2.3 Grand Chess	8
<b>2.</b>	<b>LITERATURE SURVEY</b>	<b>9</b>
<b>3.</b>	<b>HARDWARE AND SOFTWARE REQUIREMENTS</b>	<b>12</b>
<b>4.</b>	<b>PROJECT DESCRIPTION</b>	<b>13</b>

	4.1 Aim	13
	4.2 Technological Stack	13
	4.2.1 React Js	14
	4.2.2 Node Js	14
	4.2.3 Mongo db Atlas	15
	4.3 Scope of the Project	15
	4.4 Web Application	15
	4.4.1. Phases involved	16
	4.4.1.1 Frontend	16
	4.4.1.2 Backend	17
	4.4.1.3 Database	18
	4.4.2 Application Pseudo code	20
	4.5 Code Simulator	27
5.	<b>RESULTS</b>	<b>28</b>
	5.1 Simulations	28
6.	<b>CONCLUSIONS AND FUTURE</b>	<b>31</b>
	<b>ENHANCEMENTS</b>	
	6.1 Conclusions	31
	6.2 Future Enhancements	31
	<b>APPENDICES</b>	<b>32</b>
	<b>REFERENCES</b>	<b>62</b>

## LIST OF FIGURES

FIGURE NO.	TITLE	PG NO.
1.1	Position-numbered Chess Board Architecture	4
1.2	Components of Chess	6
4.1	Piece Movements Database Structure	18
4.2	Flow graph of the application	19
4.3	Player Movement Flow Diagram	27
5.1	Game Application Page	28
5.2	Movement Track Sheet	29
5.3	Fallen Soldiers Block	30
5.4	Player turn block	30

## **LIST OF SYMBOLS AND ABBREVIATIONS**

### **SYMBOLS**

P	-	Pawn
K	-	King
Q	-	Queen
R	-	Rook
N	-	Knight
B	-	Bishop
X	-	Capture

### **ABBREVIATIONS**

CSS	-	Cascading Style Sheets
JS	-	JavaScript
NPM	-	Node Package Manager
JSON	-	JavaScript Object Notation
CORS	-	Cross-origin Resource Sharing
DB	-	Database



# **CHAPTER 1**

## **INTRODUCTION**

This chapter provides a basic introduction to conventional board games, including its architecture, characteristics, and initial origin. It also explains the basic components of the game and the variations that are already being used.

### **1.1 CONVENTIONAL BOARD GAMES**

Conventional chess embodies tradition and strategy, where players engage in a cerebral battle on a checkered battlefield. Beginning with a familiar setup, each player maneuvers pieces like rooks, knights, and bishops strategically, aiming to protect their own king while plotting the downfall of their opponent's. Throughout the game, tactical moves like castling or pawn promotion add depth to the strategic landscape. The pinnacle arrives in the endgame, where a player maneuvers skillfully to checkmate the opponent's king for victory, or a draw may arise from scenarios like stalemate. Available in various formats and played across physical boards or digital platforms, chess transcends boundaries, captivating millions worldwide with its intellectual appeal and competitive essence.

#### **1.1.1 Standard 8X8 Chess Board**

The 8x8 chess board is the classic battlefield for the game of chess. Composed of 64 squares in alternating light and dark colors, this square grid serves as the stage where strategic warfare between two opponents unfolds. Each player starts with 16 pieces of contrasting colors, strategically positioned along the board's rows. The arrangement allows for varied movement and unique abilities, creating a symphony of tactics and maneuvers. Despite its modest dimensions, the 8x8 chess board encapsulates an infinite realm of

possibilities, offering a canvas where intellect, foresight, and creativity converge. Within this structured framework lies the potential for endless variations and captivating strategies, making every game a thrilling and distinct narrative. It's not merely a board but a symbol of tradition, where generations have honed their skills, and where novices and grandmasters alike continue to be captivated by its timeless allure and the challenges it presents.

### **1.1.2 Initial Origin**

In the 19th century, competitive chess became more organized, leading to the formulation of standardized rules and the establishment of international chess tournaments. Online chess is chess that is played over the Internet, allowing players to play against each other in real time. This is done through the use of Internet chess servers, which often include a system to pair up individual players based on certain criteria.

### **1.1.3 Characteristics of 8X8**

- **Board Size:**

The chessboard consists of 64 squares arranged in an 8x8 grid.

- **Piece Setup:**

Each player begins with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns.

- **Initial Placement:**

Pieces are initially arranged in specific positions on the board, with pawns in the front row and other pieces behind them.

- **Movement Rules:**

Each type of piece has distinct movement rules. For instance, pawns move forward but capture diagonally, rooks move horizontally or

vertically, bishops move diagonally, knights have a unique L-shaped move, queens can move horizontally, vertically, or diagonally, and kings can move one square in any direction.

- **Objective:**

The primary objective is to checkmate the opponent's king, meaning the king is in a position to be captured ('in check') with no legal moves to escape.

- **Special Moves:**

En passant: A pawn-capturing move where a pawn captures an opponent's pawn that has moved two squares forward from its starting position.

Castling: A king and rook move that can only be performed under certain conditions and is aimed at king safety.

- **Pawn Promotion:**

When a pawn reaches the eighth rank on the opponent's side of the board, it is promoted to any other piece (except a king), usually a queen.

- **Stalemate:**

If a player has no legal moves and their king is not in check, the game results in a stalemate, declaring the game a draw.

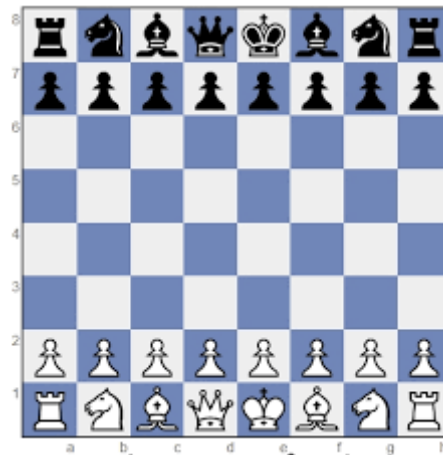
- **Draw Conditions:**

Draws can also occur through threefold repetition, the fifty-move rule, or mutual agreement between players.

- **Time Control:**

Chess games often have time controls, specifying the maximum time each player can take for their moves.

### 1.1.4 Architecture



**Figure 1.1 Position-numbered Chess Board Architecture**

This is the overall architecture of the chess board. This initial setup should be followed by everyone who is playing.

### 1.1.5 Components

The components of chess are the pieces with which it is played. The components are given as follows

- Pawn
- Knight
- Rook
- Bishop
- King
- Queen

#### 1. Pawn

Pawns are the smallest pieces in chess. They move forward but capture diagonally. On their first move, pawns can advance two squares. When they reach the opponent's back rank, they can be promoted to any other piece (except another pawn or king) if they reach the eighth rank.

## **2. Knight**

Knights move uniquely, in an L-shape (two squares in one direction and then one square perpendicular to that). Knights are the only pieces that can "jump" over other pieces. They can be quite tricky and are known for their ability to fork and attack multiple pieces simultaneously.

## **3. Rook**

It is also known as a castle or tower. Rooks move horizontally or vertically across the board. They cannot move diagonally like a bishop. They are powerful in open positions and can control ranks and files.

## **4. Bishop**

Each player starts with two bishops, one on a light square and one on a dark square. Bishops move diagonally and stay on their initial color throughout the game. They complement each other as one operates on light squares, and the other operates on dark squares.

## **5. King**

The most crucial piece, and the game's objective is to checkmate the opponent's king. The king moves one square in any direction (horizontally, vertically, or diagonally). It cannot move to a square attacked by an opponent's piece.

## **6. Queen**

It is often regarded as the most powerful piece. The queen combines the powers of both rooks and bishops. It moves diagonally, horizontally, or vertically across any number of squares.



**Figure 1.2 Components of Chess**

### 1.1.6 Applications

The different applications in use are listed as follows

- Education and Academia
- Professional Competitions
- Community Building
- Skill Development
- Entertainment and Recreation

## 1.2 VARIATIONS

Chess, a game with a rich history spanning centuries, has evolved into numerous variations, each introducing unique twists to the classic gameplay. These variations often modify the board size, piece movements, or introduce new pieces, thereby offering diverse challenges and strategies to players. Some well-known variations include:

- Courier Chess
- Tamerlane Chess
- Grand Chess

### **1.2.1 Courier Chess**

Courier Chess, an ancient variant predating modern chess, features a 12x8 board with unique pieces: the courier and chancellor. This medieval European game expands the chess experience by introducing these pieces, combining movements of traditional chess pieces. The courier mirrors the bishop and knight's moves, while the chancellor combines rook and knight movements. This larger board allows complex strategies and diverse movements, offering a captivating variation from standard 8x8 chess. Though less prevalent now, Courier Chess stands as a historically rich and engaging variant, highlighting the game's evolution and complexity throughout history.

### **1.2.2 Tamerlane Chess**

Tamerlane Chess, attributed to the conqueror Tamerlane, is a historical chess variant known for its large board, typically 10x11 or 10x12, featuring additional pieces. This game incorporates exotic and extra pieces such as the camel and elephant, each with unique moves akin to traditional chess pieces. The larger board size, coupled with these new units, amplifies strategic depth and tactical possibilities, offering players an expanded and intricate chess experience. While less widespread than standard chess, Tamerlane Chess showcases the creativity and diversity found in historical variants, enriching the game's legacy and strategic dimensions.

### **1.2.3 Grand Chess**

Grand Chess, an extension of traditional chess, is played on a larger 10x10 board. This variant introduces two new pieces: the Marshal, which combines the moves of a knight and rook, and the Cardinal, having the powers

of a knight and bishop. With more squares and pieces, Grand Chess expands strategic horizons, fostering complex and creative gameplay. The increased board size allows for deeper planning, longer games, and novel strategies, emphasizing positional play and intricate maneuvers. Grand Chess offers enthusiasts a captivating and challenging twist to the classical game, promoting new ideas and enhancing the depth of strategic thinking.



## **CHAPTER 2**

### **LITERATURE SURVEY**

This chapter shows surveys about various papers related to the web board games. The 9x9 grid elevates gameplay by intensifying strategic possibilities, extending match duration, and adding complexity. With each move, players face a broader array of choices, fostering a dynamic and intricate gaming experience. This expanded grid not only prolongs game time but also challenges players to think critically, ensuring a unique and engaging encounter with every move. Related to this we have downloaded lots of papers to gain clear knowledge.

Some of the downloaded papers were mentioned below.

**A user-interface environment solution for an online educational Chess server** (J. Picussa et al., 2008)

This focuses on creating an interactive Chess teaching environment, emphasizing interface design and functionality. The main screen layout prioritizes the game board, accommodating player information, PGN moves, and match scores. Unique features include settings for public matches, notes during games, and post-match options like reporting abuse or saving games for educational purposes. User profiles were redefined, highlighting active learners alongside roles like Managers, Robots, Helpers, Teachers, and regular Users. The interface allows for user grouping, especially beneficial in educational contexts, aiding teachers in organizing student groups efficiently. Overall, the approach combines unique interface design, user profile restructuring, and intuitive access to features, catering to both learners and educators within the Chess teaching environment.

**Design and evaluation of database and API supporting Shogi learners on the Internet.** (R. Miura et al., 2015)

The paper discusses SAKURA, an Internet-based support system for Shogi (Japanese Chess), encompassing game functions, discussion support, and database interfaces for artificial intelligence (AI) integration. The system contains databases for game records and positions, interlinked to provide integrated information like comments, evaluations, and moves. Unlike existing services, SAKURA focuses on comprehensive position databases crucial for post-game discussions, a popular aspect in Shogi. SAKURA's design emphasizes client interfaces for shared and personal boards, chat functions, and tree chart displays for discussions. The database architecture encompasses game records, positions, variations, candidate moves, and user-related information, enabling users to access and contribute to discussions effectively.

**Chess piece movement detection and tracking, a vision system framework for autonomous chess playing robot** (D.A.Christie, T.M.Kusuma and P.Musa, 2017)

The paper outlines the creation of a vision system for an autonomous chess-playing robot, focusing on the vision component. It aims to enable the robot to recognize and track chessboard elements and piece movements accurately. The work is divided into three segments: chessboard detection, chess cell identification, and chess piece movement tracking. Chessboard detection involves contour tracing and approximation to identify the board's structure. Chess cell detection uses line detection via Hough Transformation to identify cell borders, and chess piece movement detection employs stable/unstable frame detection and blob analysis to track piece movements. The method maps blob centroids to specific cells on the board and updates a matrix representing the chessboard state. The system generates Portable Game Notation (PGN) output, a standard language for chess moves. Evaluations indicate satisfactory

performance in chessboard and cell detection, though challenges persist with black piece detection in low-light conditions. The system, designed for normal chess gameplay scenarios, aims to accurately detect piece movements and generate a standard chess notation for communication.

### **Design and Implementation of Intelligent Chinese Chess System Device** (J.Zhang and H.Yin, 2020)

This paper proposes an advanced Chinese chess system employing electromagnet dot matrix for piece movement, cloud-based speech recognition for user commands, and deep learning algorithms for game decision-making. It aims to enable human-computer interaction through voice commands, facilitating real-time chess play on a physical board. The system's hardware involves modules for speech pickup, processing, and movement control, allowing players to interact via voice commands recognized through cloud-based technology. The system analyzes instructions, validates moves, and updates the board accordingly. Additionally, it determines capturing moves and updates positions dynamically. The piece movement involves a path planning unit and electromagnets arranged in a matrix, ensuring precise and controlled movement on the board.

## **CHAPTER 3**

### **HARDWARE AND SOFTWARE REQUIREMENTS**

This chapter provides a brief description about the requirements essential for our project.

#### **HARDWARE REQUIREMENT**

Hard disk	:	40 GB
RAM	:	2 GB (minimum)

#### **SOFTWARE REQUIREMENT**

Operating system	:	Windows 7 and above
Database server	:	Mongo db atlas
Library	:	React Js
Javascript runtime environment	:	Node Js

## **CHAPTER 4**

### **PROJECT DESCRIPTION**

This section delineates the advancements and extended capabilities of the 9x9 board game, emphasizing enhancements through additional functionalities to enrich the gaming experience. The framework employs various mechanisms to illustrate and assess these augmentations within the game's dynamics.

#### **4.1 AIM**

The aim of the project is to develop a 9x9 board game, likely based on chess, featuring an interactive interface for players to make moves, manage game states, and display relevant information. This includes implementing game logic, integrating backend communication to store game data, and presenting details like player turns, fallen pieces, and movement history. The goal is to create an engaging and functional gaming experience that encompasses gameplay, data management, and user interaction within the constraints of a 9x9 board structure.

#### **4.2 TECHNOLOGICAL STACK**

The technological stack for our project leverages React.js for dynamic and interactive user interfaces, Node.js for a scalable and efficient server-side environment, and MongoDB for flexible and high-performance data storage. React.js facilitates component-based UI development, Node.js enables non-blocking I/O operations and event-driven architecture, while MongoDB offers a versatile and scalable NoSQL database solution. This combination empowers us to create robust, real-time applications with seamless data flow, responsiveness, and adaptability to evolving project needs.

### **4.2.1 React Js**

React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook. React is a tool for building UI components. React creates a VIRTUAL DOM in memory. Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM. React only changes what needs to be changed. React finds out what changes have been made, and changes only what needs to be changed.

### **4.2.2 Node Js**

Node.js is a free and open-source server environment that works with a variety of operating systems, including Windows, Linux, Unix, Mac OS X, and others. JavaScript is used on the server by Node.js. Asynchronous programming is used in Node.js. Opening a file on the server and sending the content back to the client is a typical activity for a web server. This is the way a file request is handled by Node.js:

- Forwards the work to the file system of the machine.
- All set to take on the next request.
- The content is returned to the client by the server after the file system has opened and read the file.
- Node.js does away with waiting and just moves on to the next request.

Single-threaded, non-blocking, asynchronous programming is run by Node.js and is incredibly memory-efficient.

### **4.2.3 Mongo DB ATLAS**

MongoDB Atlas is indeed a multi-cloud database service offered by the creators of MongoDB. It streamlines the deployment and management of databases while providing the flexibility required to construct robust, high-performing global applications across various cloud platforms. Atlas stands out as an exceptional platform to run MongoDB, a prominent non-relational database. MongoDB's document-based model offers rapid innovation as it seamlessly aligns with the objects within your code. This model's efficiency lies in the direct mapping between documents in the database and the structures within the application's codebase, fostering faster development and easier data manipulation.

### **4.3 SCOPE OF THE PROJECT**

The project's scope encompasses enriching the 9x9 board game by introducing advanced functionalities and improved user interactions. It involves implementing innovative features, refining gameplay, and assessing user experiences to elevate the game's dynamics and engagement. This scope includes rigorous testing and evaluation within a specified framework to ensure seamless integration and enhanced gameplay.

### **4.4 WEB APPLICATION**

This application lets users play a 9x9 board game. First, player 1 makes a move, and then, after validating their movement using possible move conditions, the movement is updated in the database. Then, on Player 2's screen, this movement is displayed. Now, when player 2 makes a movement, that movement is again validated using the predefined set of rules that are encoded in the application program. The same as Player 1, Player 2's move is updated in the database and displayed on Player 1's screen. After both players make a single move, the movement track sheet is updated with values such as serial

number, white player's move, white player's time taken to make the move, black player's time, and black player's time to make the move. Also, each player's turn is displayed on the screen. The application doesn't allow a player to make a move during another player's time. Along with that, it doesn't allow a player to make two moves consecutively.

#### **4.4.1 Phases Involved**

- Frontend
- Backend
- Database

##### **4.4.1.1 Frontend**

- i. The player should create an account to play the game. They create their account using credentials like their email ID, their unique user ID, and their password.
- ii. Next, they will be redirected to the sign-up page, where they can sign in using their user ID and password.
- iii. After successful sign-in, they are redirected to the game page, where they can play.
- iv. Each player takes a turn and plays their game. The white player takes the first turn.
- v. If there are any invalid movements, they are notified on the screen, and the corresponding movement is not recorded or stored.
- vi. After player 2 (black) completes his or her move, both the white and black players' moves are updated on the movement track sheet on the screen. All the movements displayed on the track sheet are retrieved from the database only.
- vii. Along with this, the captured soldiers are pushed to the fallen soldier block.



#### 4.4.1.2 Backend

- i. The '/login' endpoint handles user login attempts by verifying the provided credentials against the MongoDB database. If the user is authenticated successfully, a "Login success" message is sent; otherwise, appropriate error messages are returned.
- ii. The '/signup' endpoint facilitates user registration by checking if the provided userID is already taken. If the userID is available, the user information is inserted into the "credentials" collection in the MongoDB database.
- iii. The code connects to a MongoDB database hosted globally on mongo db atlas. The '/InsertWhiteBlack' and '/Insert' endpoints update records in the "pvsc" and "pieceMovements" collections, respectively, based on the provided values and serial numbers. The '/InsertWhiteBlack' updates the coin movement to another player made by another player. If the black player moves, his move is stored in the database, and on the white player's screen, it is updated from the database. The '/Insert' endpoint receives a string that contains the move made by a player. It stores the piece that was moved, the source position of the piece before movement, and the destination position of the piece. If captured, it also appends the captured notation.
- iv. The '/Update' endpoint updates the "black" field in the "pieceMovements" collection for a specified serial number. The serial starts at 1 increment each time a round is completed to keep track of the number of movements done.

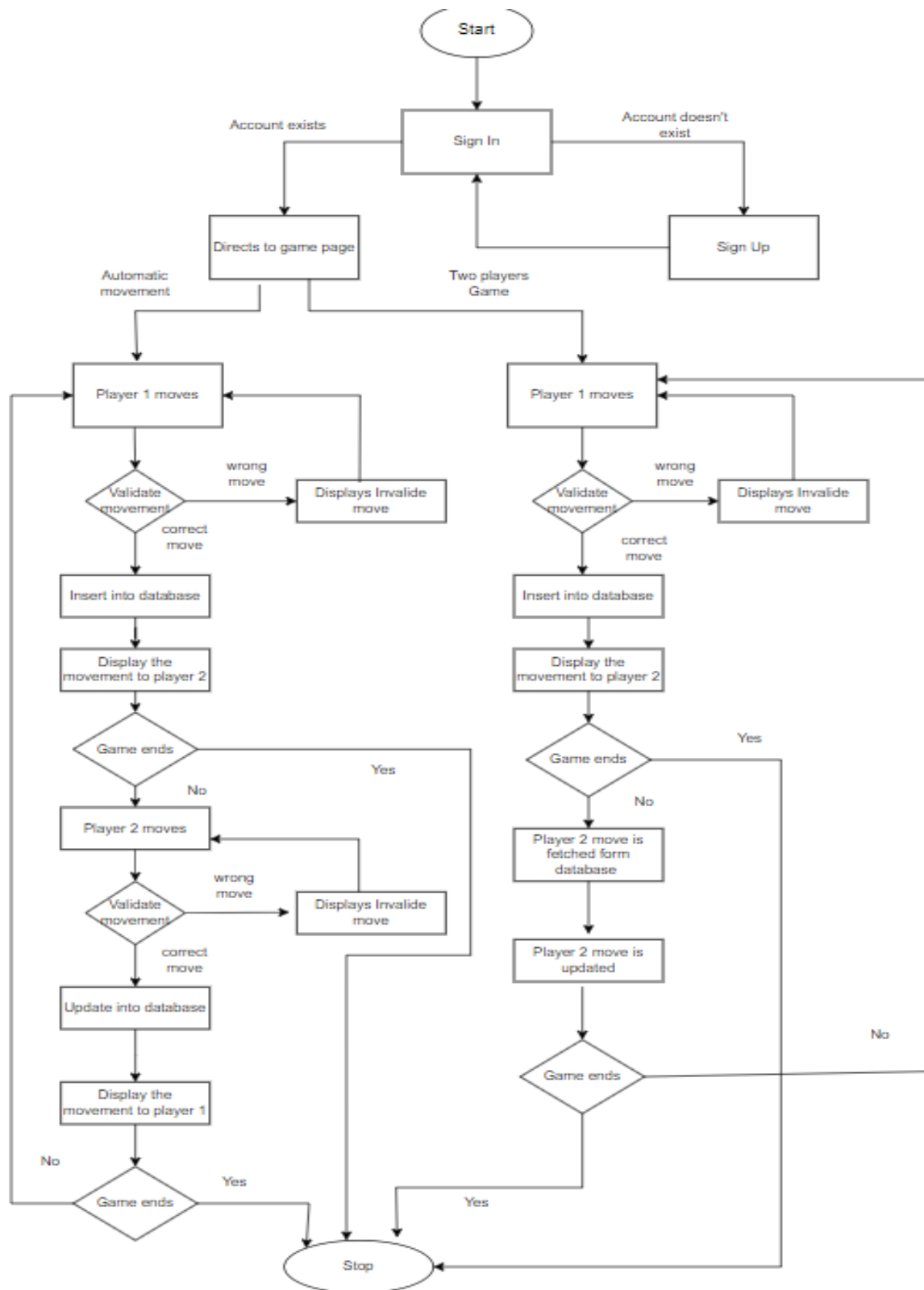
- v. The server is configured to run on port 5000 (app.listen(5000)). Cross-Origin Resource Sharing (CORS) is enabled to allow requests from different origins. Body parsing middleware is used to handle JSON and URL-encoded data.
- vi. The server can be utilized for user authentication, registration, and managing records in the MongoDB database. All backend packages are installed with npm.
- vii. Users can log in, sign up, and update information related to the "pvsc" and "pieceMovements" collections.

#### 4.4.1.3 Database

- i. In the MongoDB Atlas, one collection is used to store the credentials of the user.
- ii. Another cluster is used for storing the piece movements made by each player.



**Figure 4.1 Piece Movements Database Structure**



**Figure 4.2 Flow graph of the application**

#### 4.4.2 Application Pseudo Code

##### Frontend Game Page:

constructor:

squares = initialiseChessBoard()

whiteFallenSoldiers = []

blackFallenSoldiers = []

player = 1

sourceSelection = -1

status = "

turn = 'white'

val = "

place = ["a9", "b9", ..., "i1"]

coins = ["R", "N", "B", ..., "R"]

counter = 0

sno = 1

tableData = []

prevTime = null

currentTime = new Date()

currenttimeDifference = null

prevtimeDifference = null

updateTime:

currentTime = new Date()

prevTime = this.state.currentTime

currenttimeDifference = prevTime ? abs(currentTime - prevTime) / 1000 :

null

prevtimeDifference = this.state.currenttimeDifference

setState:

prevTime

currentTime

currenttimeDifference

prevtimeDifference

handleClick(i):

squares = copy of this.state.squares

if sourceSelection is -1:

if not squares[i] or squares[i].player is not this.state.player:

setState: status = "Wrong selection. Choose player " + this.state.player  
+ " pieces."

if squares[i]:

squares[i].style.backgroundColor = ""

else:

squares[i].style.backgroundColor = "RGB(111,143,114)"

```

        setState:

            status = "Choose destination for the selected piece"

            sourceSelection = i

        return

    updateTime()

    squares[sourceSelection].style.backgroundColor = ""

    if squares[i] and squares[i].player is this.state.player:

        setState:

            status = "Wrong selection. Choose valid source and destination
again."

            sourceSelection = -1

    else:

        whiteFallenSoldiers = []

        blackFallenSoldiers = []

        isDestEnemyOccupied = Boolean(squares[i])

        isMovePossible = squares[sourceSelection]. isMovePossible(
            sourceSelection, i, isDestEnemyOccupied)

        coinsCopy = copy of this.state.coins

        coinsCopy[i] = this.state.coins[sourceSelection]

        coinsCopy[sourceSelection] = ""

        setState:

```

```

    val = coins[sourceSelection] + place[sourceSelection] + (
        'X' + place[i] if squares[i] else place[i] )

    coins = coinsCopy

    if isMovePossible:

        if squares[i]:

            if squares[i].player is 1:

                whiteFallenSoldiers.push(squares[i])

            else:

                blackFallenSoldiers.push(squares[i])

        squares[i] = squares[sourceSelection]

        squares[sourceSelection] = null

        isCheckMe = isCheckForPlayer(squares, this.state.player)

        if isCheckMe:

            setState:

                status = "Wrong selection. Choose valid source and destination
again. Now you have a check!"

                sourceSelection = -1

        else:

            player = 2 if this.state.player is 1 else 1

            turn = 'black' if this.state.turn is 'white' else 'white'

            setState:

```

```

        sourceSelection = -1

        squares

        whiteFallenSoldiers

        blackFallenSoldiers

        player

        status = "

        turn

    else:

        setState:

            status = "Wrong selection. Choose valid source and destination
again."

            sourceSelection = -1

componentDidUpdate(prevProps, prevState):

    if this.state.val is not prevState.val:

        if this.state.counter is 0:

            axios.post("http://localhost:5000/Insert", this.state)

                .then(result => console.log(result))

                .catch(error => console.log(error))

            setState:

                counter = 1

        else:

```



```

    axios.post("http://localhost:5000/Update", this.state)

    .then(result =>

        if result.data and result.data.data:

            updatedData = (

                result.data.data.map(item => ({...item, prevtimeDifference:
this.state.prevtimeDifference,                                currenttimeDifference:
this.state.currenttimeDifference})))

            if Array.isArray(result.data.data)

            else [{

                ...result.data.data,

                prevtimeDifference: this.state.prevtimeDifference,

                currenttimeDifference: this.state.currenttimeDifference

            }]

        )

        setState:

            tableData = [...prevState.tableData, ...updatedData]

            counter = 0

            sno = prevState.sno + 1

        else:

            console.error("Received data is not an array:", result.data)

    )

```

```
.catch(error => console.log(error))
```

renderTable:

```
if not Array.isArray(this.state.tableData):
```

```
    console.error("tableData is not an array:", this.state.tableData)
```

```
    return null
```

```
return (
```

```
    table with tableData
```

```
)
```

getKingPosition(squares, player):

```
    reduce squares to find and return the position of the player's king
```

isCheckForPlayer(squares, player):

```
    opponent = 2 if player is 1 else 1
```

```
    playersKingPosition = getKingPosition(squares, player)
```

```
    canPieceKillPlayersKing = (piece, i) =>
    piece.isMovePossible(playersKingPosition, i, squares)
```

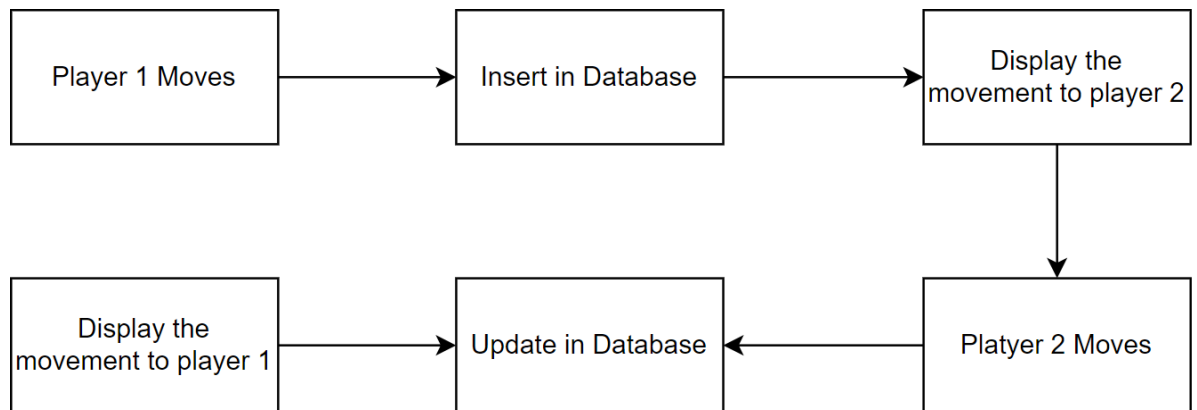
```
    reduce squares to check if any piece can kill the player's king
```

render:

```
    return a div containing the game components
```

## 4.5 CODE SIMULATOR

The code is tested on the local host. The frontend runs on <http://localhost:3000>, the server runs on <http://localhost:5000>, and the database runs in the Atlas cloud.



**Figure 4.3 Player Movement Flow Diagram**

When the frontend code runs with the help of Axios, we connect to the backend. In the backend, using MongoDB's connect URL, we connect to the specified account. The frontend consists of ReactJs, CSS, and Bootstrap, and the backend consists of JavaScript code.

## CHAPTER 5

### RESULTS

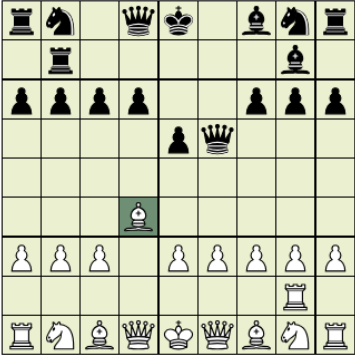
This chapter specifies the resultant application page, such as the two players playing the game, the movement track sheet, the fallen soldiers block, and each player's time taken.

#### 5.1 SIMULATIONS


The below image is the game application interface page, which will be used by the players to play the game.

---

### 9x9 Board Game





**Turn**




Bd4Xe6

Choose destination for the selected piece

### Fallen Soldiers Block



### Movement Track Sheet

Sno	White	Black
1	Pd3d5	Pe7e6
2	Pd5Xe6	Pf7Xe6
3	Bb2d4	Bc9f6
4	Pc3Xf6	Qf9Xf6

**Figure 5.1 Game Application Page**

The below is the output displayed from the database for the movement track sheet.

## Movement Track Sheet

Sno	White	Black
1	Pd3d5	Pe7e6
2	Pd5Xe6	Pf7Xe6
3	Pe3e5	Pg7g6
4	Pe5Xd6	Bh8f6
5	Pe7Xd6	Nh9g7
6	Pd6d5	Pg6Xb1
7	Pf5Xe6	Pb3b5
8	e5f6	Pc3c4
9	g6g5	Bb2d4

**Figure 5.2 Movement Track Sheet**

The following is the output displayed in the fallen soldiers block and the player turn block.

## Fallen Soldiers Block



**Figure 5.3 Fallen Soldiers Block**

## Turn



Be7Xd8

**Figure 5.4 Player turn block**

## **CHAPTER 6**

### **CONCLUSIONS AND FUTURE ENHANCEMENTS**

This chapter concludes about the advantages of 9x9 and the future enhancement of our project.

#### **6.1 Conclusions**

While the conventional 8x8 grid in the game offers numerous advantages, the transition to a 9x9 grid introduces a new dimension of complexity. The expanded layout not only enables extended strategies for players but also diversifies opening and middle-game scenarios. The 9x9 grid serves as a valuable learning and teaching tool, fostering innovation and exploration within the game. This alteration in size prompts players to reevaluate traditional approaches, encouraging a deeper understanding of strategic nuances. Overall, the shift to a 9x9 grid enriches the gaming experience, providing a platform for enhanced skill development and a more nuanced exploration of the game's intricacies.

#### **6.2 Future Enhancements**

The upcoming enhancements for our project involve integrating a hardware board to detect movements made on the board itself. This hardware will interact with the application to validate these movements, ensuring accuracy and reliability. Once validated, the movements will be stored directly in the database, updating the game state seamlessly. This integration will provide real-time tracking and synchronization between physical board actions and the digital application. It aims to enhance user experience by bridging the gap between physical gameplay and digital data storage, allowing for a seamless and efficient interaction between the two mediums.

## APPENDICES

### **game.js**

```
import React from 'react';

import axios from 'axios';

import '../index.css';

import Board from './board.js';

import King from '../pieces/king'

import FallenSoldierBlock from './fallen-soldiers-block.js';

import initialiseChessBoard from './helpers/board-initialiser.js';

export default class Game extends React.Component {

  constructor() {

    super();

    this.state = {

      squares: initialiseChessBoard(),

      whiteFallenSoldiers: [],

      blackFallenSoldiers: [],

      player: 1,

      sourceSelection: -1,

      status: "",

      turn: 'white',
```



```

    val:",

    place:["a9","b9","c9","d9","e9","f9","g9","h9","i9","a8","b8","c8","d8","e8","f8",
    "g8","h8","i8","a7","b7","c7","d7","e7","f7","g7","h7","i7","a6","b6","c6","d6",
    "e6","f6","g6","h6","i6","a5","b5","c5","d5","e5","f5","g5","h5","i5","a4","b4",
    "c4","d4","e4","f4","g4","h4","i4","a3","b3","c3","d3","e3","f3","g3","h3","i3",
    "a2","b2","c2","d2","e2","f2","g2","h2","i2","a1","b1","c1","d1","e1","f1","g1",
    "h1","i1"],

    coins:["R","N","B","Q","K","Q","B","N","R","","R","","","","","B","","P","P",
    "P","P","P","P","P","P","P","","","","","","","","","","","","","","","","",
    "","","","","","","","P","P","P","P","P","P","P","P","P","P","B","","","","","R",
    "","","R","N","B","Q","K","Q","B","N","R"],

    counter:0,

    sno:1,

    tableData : [],

    prevTime: null,

    currentTime: new Date(),

    currenttimeDifference: null,

    prevtimeDifference: null

  }

}

updateTime() {

  const currentTime = new Date();

  const prevTime = this.state.currentTime;

```

```

    const currenttimeDifference = prevTime ? Math.abs(currentTime - prevTime)
/ 1000 : null;

    const prevtimeDifference = this.state.currenttimeDifference

    this.setState({

        prevTime,

        currentTime,

        currenttimeDifference,

        prevtimeDifference

    });

}

handleClick(i) {

    const squares = [...this.state.squares];

    if (this.state.sourceSelection === -1) {

        if (!squares[i] || squares[i].player !== this.state.player) {

            this.setState({ status: "Wrong selection. Choose player " + this.state.player
+ " pieces." });

            if (squares[i]) {

                squares[i].style = { ...squares[i].style, backgroundColor: "" };

            }

        }

        else {

```

```

        squares[i].style = { ...squares[i].style, backgroundColor:
"RGB(111,143,114)" };

        this.setState({

            status: "Choose destination for the selected piece",

            sourceSelection: i,

        })

    }

    return

}

this.updateTime();

squares[this.state.sourceSelection].style = {
...squares[this.state.sourceSelection].style, backgroundColor: "" };

if (squares[i] && squares[i].player === this.state.player) {

    this.setState({

        status: "Wrong selection. Choose valid source and destination again.",

        sourceSelection: -1,

    });

}

else {

    const whiteFallenSoldiers = [];

    const blackFallenSoldiers = [];

```

```

const isDestEnemyOccupied = Boolean(squares[i]);

const isMovePossible =
squares[this.state.sourceSelection].isMovePossible(this.state.sourceSelection, i,
isDestEnemyOccupied);

let coinsCopy = {...this.state.coins};

coinsCopy[i] = this.state.coins[parseInt(this.state.sourceSelection)]; //new
src

coinsCopy[parseInt(this.state.sourceSelection)] = ""; //prev src null

if(squares[i] !== null)
{
    this.setState({
        val :
this.state.coins[parseInt(this.state.sourceSelection)]+this.state.place[parseInt(thi
s.state.sourceSelection)]+"X"+this.state.place[parseInt(i)],

        coins :coinsCopy
    })
}
else{
    this.setState({
        val :
this.state.coins[parseInt(this.state.sourceSelection)]+this.state.place[parseInt(thi
s.state.sourceSelection)]+this.state.place[parseInt(i)],

        coins :coinsCopy
    })
}

```

```

    })

    }

    if (isMovePossible) {

        if (squares[i] !== null) {

            if (squares[i].player === 1) {

                whiteFallenSoldiers.push(squares[i]);

            }

            else {

                blackFallenSoldiers.push(squares[i]);

            }

        }

        squares[i] = squares[this.state.sourceSelection];

        squares[this.state.sourceSelection] = null;

        const isCheckMe = this.isCheckForPlayer(squares, this.state.player)

        if (isCheckMe) {

            this.setState(oldState => ({

                status: "Wrong selection. Choose valid source and destination again.
Now you have a check!",

                sourceSelection: -1,

            })))

        } else {

```

```

let player = this.state.player === 1 ? 2 : 1;

let turn = this.state.turn === 'white' ? 'black' : 'white';


this.setState(oldState => ({

  sourceSelection: -1,

  squares,

  whiteFallenSoldiers: [...oldState.whiteFallenSoldiers,
...whiteFallenSoldiers],

  blackFallenSoldiers: [...oldState.blackFallenSoldiers,
...blackFallenSoldiers],

  player,

  status: "",

  turn

})));

}

}

else {

  this.setState({

    status: "Wrong selection. Choose valid source and destination again.",

    sourceSelection: -1,

  });

```

```

    }

    }

}

componentDidUpdate(prevProps, prevState) {

    console.log(this.state.val)

    if (this.state.val !== prevState.val) {

        if (this.state.counter === 0) {

            axios.post("http://localhost:5000/Insert", this.state)

                .then(result => {

                    console.log(result);

                })

                .catch(error => {

                    console.log(error);

                });

            this.setState({

                counter: 1

            });

        }

        else {

            axios.post("http://localhost:5000/Update", this.state)

```

```

.then(result => {

  console.log(result);

  if (result.data && result.data.data) {

    const updatedData = Array.isArray(result.data.data)

? result.data.data.map(item => ({

  ...item,

  prevtimeDifference: this.state.prevtimeDifference,

  currenttimeDifference: this.state.currenttimeDifference

  )))

: [

  {

    ...result.data.data,

    prevtimeDifference: this.state.prevtimeDifference,

    currenttimeDifference: this.state.currenttimeDifference

  }

];

this.setState(prevState => ({

  tableData: [

    ...prevState.tableData,

    ...updatedData

```



```

    ],

    counter: 0,

    sno: prevState.sno + 1,

  }));

  } else {

    console.error("Received data is not an array:", result.data);

  }

})

.catch(error => {

  console.log(error);

});

}

}

}

renderTable() {

  if (!Array.isArray(this.state.tableData)) {

    console.error("tableData is not an array:", this.state.tableData);

    return null; // or handle it accordingly

  }

  return (

```

```
<tableclassName=' table table-light table-bordered table-  
hover'style={{ width : "70% " , textAlign : "center" }}>
```

```
<thead>
```

```
<tr>
```

```
<th>Sno</th>
```

```
<th>White</th>
```

```
<th>White Time</th>
```

```
<th>Black</th>
```

```
<th>Black Time</th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
{ this.state.tableData.map(row => (
```

```
<tr key={row.sno}>
```

```
<td>{row.sno}</td>
```

```
<td>{row.white}</td>
```

```
<td>{row.prevertimeDifference}</td>
```

```
<td>{row.black}</td>
```

```
<td>{row.currenttimeDifference}</td>
```

```
</tr>
```

```
)))
```

```

    </tbody>

</table>

);

}

getKingPosition(squares, player) {

return squares.reduce((acc, curr, i) =>

acc || //King may be only one, if we had found it, returned his position

((curr //current square mustn't be a null

&& (curr.getPlayer() === player)) //we are looking for a special king

&& (curr instanceof King)

&& i), // returned position if all conditions are completed

null)

}

isCheckForPlayer(squares, player) {

const opponent = player === 1 ? 2 : 1

const playersKingPosition = this.getKingPosition(squares, player)

const canPieceKillPlayersKing = (piece, i) =>
piece.isMovePossible(playersKingPosition, i, squares)

return squares.reduce((acc, curr, idx) =>

acc ||

(curr &&

```

```

    (curr.getPlayer() === opponent) &&
    canPieceKillPlayersKing(curr, idx)
    && true),
    false)
}

render() {
    return (
        <div>
            <div className="game">
                <h2 style={{ textAlign: "left" }}>9x9 Board Game</h2>
                <div className="game-board">
                    <Board
                        squares={this.state.squares}
                        onClick={i => this.handleClick(i)}
                    />
                </div>
                <div className="game-info">
                    <h3>Turn</h3>
                    <div id="player-turn-box" style={{ backgroundColor: this.state.turn }}>
                        </div>

```

```

<div>

<div >{this.state.val}</div>

</div>

<div className="game-status">{this.state.status}</div>

<div className="fallen-soldier-block">

  <p className='display-6'>Fallen Soldiers Block</p>

  {<FallenSoldierBlock

    whiteFallenSoldiers={this.state.whiteFallenSoldiers}

    blackFallenSoldiers={this.state.blackFallenSoldiers}

  />

  }

</div>

</div>

<div className='container-fluid'>board.js

  <p className="display-6">Movement Track Sheet</p>

  {this.renderTable()}

</div>

</div>

</div>

);

```

```
}  
  
}
```

### **board-initialiser.js**

```
import Bishop from '../pieces/bishop.js';  
  
import King from '../pieces/king.js';  
  
import Knight from '../pieces/knight.js';  
  
import Pawn from '../pieces/pawn.js';  
  
import Queen from '../pieces/queen.js';  
  
import Rook from '../pieces/rook.js';  
  
export default function initialiseChessBoard() {  
  
  const squares = Array(81).fill(null);  
  
  for (let i = 18; i < 27; i++) {  
  
    squares[i] = new Pawn(2);  
  
    squares[i + 36] = new Pawn(1);  
  
  }  
  
  squares[0] = new Rook(2);  
  
  squares[8] = new Rook(2);  
  
  squares[80] = new Rook(1);  
  
  squares[72] = new Rook(1);  
  
}
```

```
squares[1] = new Knight(2);  
  
squares[7] = new Knight(2);  
  
squares[79] = new Knight(1);  
  
squares[73] = new Knight(1);  
  
squares[2] = new Bishop(2);  
  
squares[6] = new Bishop(2);  
  
squares[74] = new Bishop(1);  
  
squares[78] = new Bishop(1);  
  
squares[10] = new Rook(2);  
  
squares[70] = new Rook(1);  
  
squares[16] = new Bishop(2);  
  
squares[64] = new Bishop(1);  
  
squares[3] = new Queen(2);  
  
squares[5] = new Queen(2);  
  
squares[77] = new Queen(1);  
  
squares[75] = new Queen(1);  
  
squares[4] = new King(2);  
  
squares[76] = new King(1);  
  
return squares;  
  
}
```

### **piece.js**

```
export default class Piece {  
  
  constructor(player, iconUrl) {  
  
    this.player = player;  
  
    this.style = { backgroundImage: "url('" + iconUrl + "')" };  
  
  }  
  
  getPlayer() {  
  
    return this.player  
  
  }  
  
}
```

### **board.js**

```
import React from 'react';  
  
import './index.css';  
  
import Square from './square.js';  
  
export default class Board extends React.Component {  
  
  renderSquare(i, squareShade) {  
  
    return <Square  
  
      key={i}  
  
      keyVal={i}
```



```

    style={this.props.squares[i] ? this.props.squares[i].style : null}

    shade={squareShade}

    onClick={() => this.props.onClick(i)}

  />

}

render() {

  const board = [];

  for (let i = 0; i < 9; i++) {

    const squareRows = [];

    for (let j = 0; j < 9; j++) {

      const squareShade = "light-square"

      squareRows.push(this.renderSquare((i * 9) + j, squareShade));

    }

    board.push(<div className="board-row" key={i}>{squareRows}</div>)

  }

  return (

    <div>

      {board}

    </div>

  );

```

```
}
```

```
}
```

### **fallen-soldiers-block.js**

```
import React from 'react';
```

```
import './index.css';
```

```
import Square from './square.js';
```

```
export default class FallenSoldierBlock extends React.Component {
```

```
  renderSquare(square, i, squareShade) {
```

```
    return <Square
```

```
      key={i}
```

```
      keyVal={i}
```

```
      piece={square}
```

```
      style={square.style}
```

```
    />
```

```
  }
```

```
  render() {
```

```
    return (
```

```
      <div>
```

```
        <div className="board-row">{this.props.whiteFallenSoldiers.map((ws,  
index) =>
```

```

        this.renderSquare(ws, index)

    )}</div>

    <div className="board-row">{this.props.blackFallenSoldiers.map((bs,
index) =>

        this.renderSquare(bs, index)

    )}</div>

</div>

);

}

}

```

### **square.js**

```

import React from 'react';

import './index.css';

export default function Square(props) {

    return (

        <button className={"square " + props.shade}

            onClick={props.onClick}

            style={props.style}

            key={props.keyVal}

        >

```

```
</button>

);

}
```

## **index.js**

```
const express = require('express')

const bodyParser = require('body-parser')

const cors = require('cors')

const app = express()

app.use(cors())

app.use(express.json())

app.use(bodyParser.urlencoded({
  extended : true
}))

app.post('/login', function (req, res) {

  let userID = req.body.userID

  let password = req.body.password

  const MongoClient = require("mongodb").MongoClient

  const url = 'mongodb://localhost:27017/'

  MongoClient.connect(url)
```

```

.then(

function (db)

{

var dbo = db.db('movements')

var query = {userID: userID}

dbo.collection("credentials").findOne(query)

.then(function (result) {

    if(result){

        if(result.password === password){

            res.send("<h1>Login success</h1>")

        }

        else{

            res.send("<h1>Invalid credentials</h1>")

        }

    }

    else{

        res.send("<h1>User doesn't exist</h1>")

    }

    db.close();

})

```

```

        .catch(function (err) {

            console.log(err);

            res.send("<h1>Login failure</h1>")

        })

    }).

    catch(function (err) {

        console.log(err)

    })

})

app.post('/signup', function(req,res){

    let uname = req.body.uname

    let userID = req.body.userID

    let email = req.body.email

    let password = req.body.password

    const MongoClient = require('mongodb').MongoClient

    const url = 'mongodb://localhost:27017'

    MongoClient.connect(url)

    .then(

        function(db){

            var dbo = db.db('movements')

```

```

var query = {userID:userID}

var in_query =
{uname:uname,userID:userID,email:email,password:password}

dbo.collection("credentials").findOne(query)

.then(function(result){

    if(result){

        res.send("<h1>User ID_taken</h1>")

        db.close()

    }

    else{

        dbo.collection("credentials").insertOne(in_query)

        .then(function(result){

            res.send("<h1>success</h1>")

            db.close()

        })

        .catch(function(err){

            console.log(err)

            res.send("<h1>Signup Failure</h1>")

        })

    }

})

```

```

        .catch(function(err){

            console.log(err)

        })

    }

)

.catch(function(err){

    console.log(err)

})

})

app.post('/InsertWhiteBlack' , function(req,res){

    let val = req.body.val

    let sno = req.body.sno

    const MongoClient = require("mongodb").MongoClient

    const url = 'mongodb://localhost:27017'

    MongoClient.connect(url)

    .then(

        function(db)

        {

            var dbo = db.db('movements')

            var newval = { $set: { white: val } };

```



```

var query = { sno: sno };

dbo.collection("pvsc").updateOne(query,newval)

.then(function(){

    console.log(val + " white updated");

    dbo.collection('pvsc').findOne(query)

    .then(function(result) {

        console.log(result);

        res.json({ message: 'updated', data: result });

    })

    .catch(function(err) {

        console.log(err);

        res.status(500).json({ message: 'Error fetching updated data' });

    });

})

.catch(function(err){

    console.log(err)

})

})

.catch(function(err){

    console.log(err)

```

```

    })

  })

app.post('/Insert' , function(req,res){

  let val = req.body.val

  let sno = req.body.sno

  const MongoClient = require("mongodb").MongoClient

  const url = 'mongodb://localhost:27017'

  MongoClient.connect(url)

  .then(

    function(db)

    {

      var dbo = db.db('movements')

      var query = {sno:sno , white : val , black : ""}

      dbo.collection("pieceMovements").insertOne(query)

      .then(function(){

        console.log(val + " white updated");

        res.send("white updated")

      })

      .catch(function(err){

```

```

        console.log(err)

    })

})

.catch(function(err){

    console.log(err)

})

})

app.post('/Update', function(req, res) {

    let val = req.body.val;

    let sno = req.body.sno;


    const MongoClient = require("mongodb").MongoClient;

    const url = 'mongodb://localhost:27017';

    MongoClient.connect(url)

        .then(function(db) {

            var dbo = db.db('movements');

            var query = { sno: sno };

            var newval = { $set: { black: val } };

```

```

    dbo.collection("pieceMovements").updateOne(query, newval)

    .then(function() {

        console.log(val + " black updated");

        // Fetch the updated document after the update

        dbo.collection('pieceMovements').findOne(query)

            .then(function(result) {

                console.log(result);

                res.json({ message: 'black updated', data: result });

            })

            .catch(function(err) {

                console.log(err);

                res.status(500).json({ message: 'Error fetching updated data'

            });

        });

    })

    .catch(function(err) {

        console.log(err);

        res.status(500).json({ message: 'Error updating black' });

    });

})

.catch(function(err) {

```

```
    console.log(err);

    res.status(500).json({ message: 'Error connecting to the database' });

  });

});

app.listen(5000,function(){

  console.log("Server is running on port 5000")

})
```

## REFERENCES

- [1] J.Picussa et.al , “A user-interface environment solution for an online educational Chess server”. At 2008 Second International Conference on Research Challenges in Information Science, page no 3-6, 2008
- [2] R.Muira et al (2015), “Design and evaluation of database and API supporting Shogi learners on the Internet”. At 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) , June 2015, pages 2-3, 2015
- [3] D.A.Christie, T.M.Kusuma and P.Musa , “Chess piece movement detection and tracking, a vision system framework for autonomous chess playing robot”. In 2017 Second International Conference on Informatics and Computing (ICIC), pages 2-4 , 2017
- [4] J.Zhang and H.Yin, “Design and Implementation of Intelligent Chinese Chess System Device”. In 2020 International Conference on Culture-oriented Science & Technology (ICCST) , pages 1-5 , 2020
- [5] Nino U. Pilueta<sup>1</sup>, Honeylet D. Grimald, Moises F. Jardimiano<sup>1</sup> and Manuel Garcia, “Chessbot: A Voice-Controlled Chess Board with Self- Moving Pieces”. In proceedings of AIP conference, 2021, Page no.3-4.
- [6] P.Karia, V.Jain, M.Shah and S.Rane “Digitization of Chess Board and Prediction of Next Move”. In 2022 IEEE 7th International conference for Convergence in Technology (I2CT), pages 1-3, Mumbai,India, 2022.
- [7] N.L.T.Tra, P.T.Cong and N.D.Anh , “Design A Chess Movement Algorithm and Detect the Movement by Images Classification Using Support Vector Machine Classifier”. In 2018 4th International Conference on Green Technology and Sustainable Development (GTSD), pages 1-4.

- [8] D.Hubner, A.Schall and M.Tangermann,“Two Player Online Brain-Controlled Chess”, 2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC) , pages 1-3 , Germany , Berlin.
- [9] P. K. Rath, N. Mahapatro, P. Nath and R. Dash, “Autonomous Chess Playing Robot”. In 2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), pages 5-6, New Delhi, India, 2019.
- [10] Y. Ban and Y. Zhao, “Algorithm Research on Online Game of Chinese Chess” . At 2012 International Conference on Computer Science and Service System , pages 2-3, Nanjing, China.