# Turn-Based Game Implementation

## 1. Project Setup

### 1.1. Create a GitHub Repository

- Repository Name: Use your full name and VIT registration number.
- Directory Structure:
  - /server: Server-side code.
  - /client: Client-side code.
  - /common: Shared code (e.g., game state management).

### 1.2. Initialize the Project:

```
mkdir turn-based-game
cd turn-based-game
mkdir server client common
```

## 2. Server-Side Implementation

### 2.1. Choose a Backend Language

- Language: Node.js
- Initialize Node.js Project:

```
cd server
npm init -y
npm install ws
```

```

2.2. Implement Game Logic

- Game State & Logic (common/game.js):

```javascript
class Game {
    constructor() {
        this.grid = Array.from({ length: 5 }, () => Array(5).fill(null));
        this.players = { A: [], B: [] };
        this.currentPlayer = 'A';
    }

    placeCharacter(player, character, position) {
        const [x, y] = position;
        if (this.grid[x][y] === null) {
            this.grid[x][y] = { player, character };
            this.players[player].push({ character, position });
        }
    }

    moveCharacter(player, character, move) {
        // Implement movement logic based on character type
        // Validate move, update grid, and check for captures
        // Return the updated game state
    }

    getGameState() {
```
```

```
        return this.grid;
    }
}

module.exports = Game;
```

- WebSocket Server (server/index.js):

```javascript
const WebSocket = require('ws');
const Game = require('../common/game');

const wss = new WebSocket.Server({ port: 8080 });
const game = new Game();

wss.on('connection', (ws) => {
    ws.send(JSON.stringify({ type: 'init', state: game.getGameState() }));

    ws.on('message', (message) => {
        const { player, command } = JSON.parse(message);
        const [character, move] = command.split(':');

        const result = game.moveCharacter(player, character, move);

        if (result.valid) {
            wss.clients.forEach(client => {
                client.send(JSON.stringify({ type: 'update', state: game.getGameState() }));
```

```
            });
        } else {
            ws.send(JSON.stringify({ type: 'invalid', message: result.message }));
        }
    });
});
```

**3. Client-Side Implementation**

3.1. Setup Frontend

```
cd ../client
npm init -y
npm install --save websocket
```

3.2. HTML Structure (client/index.html):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Turn-based Game</title>
    <style>
        /* Add basic styles for the 5x5 grid */
```

```
        #game-board {
            display: grid;
            grid-template-columns: repeat(5, 50px);
            grid-template-rows: repeat(5, 50px);
            gap: 5px;
        }

        .cell {
            width: 50px;
            height: 50px;
            border: 1px solid black;
            text-align: center;
            line-height: 50px;
        }
    </style>
</head>
<body>
    <h1>Turn-based Game</h1>
    <div id="game-board"></div>
    <div id="controls"></div>
    <div id="status"></div>

    <script src="index.js"></script>
</body>
</html>
```

3.3. WebSocket Communication (client/index.js):

```javascript
const ws = new WebSocket('ws://localhost:8080');

ws.onmessage = (event) => {
    const message = JSON.parse(event.data);

    if (message.type === 'init' || message.type === 'update') {
        renderGameBoard(message.state);
    } else if (message.type === 'invalid') {
        alert(message.message);
    }
};

function renderGameBoard(state) {
    const board = document.getElementById('game-board');
    board.innerHTML = '';

    for (let i = 0; i < state.length; i++) {
        for (let j = 0; j < state[i].length; j++) {
            const cell = document.createElement('div');
            cell.className = 'cell';
            if (state[i][j]) {
                cell.textContent = `${state[i][j].player}-${state[i][j].character}`;
            }
            board.appendChild(cell);
        }
```

```
    }
}
```

**4. Handle Game Rules and Flow**

4.1. Implement Movement and Combat Logic

- Game Logic Update (server/game.js):

```javascript
moveCharacter(player, character, move) {
    // Implement the specific logic based on the character type
    // Example: Moving a Pawn
    // Validate the move and update the grid
    // If the move is valid, update the game state and return true
    // Else, return false with an error message
}
```

4.2. Validate Moves

- Client-Side Validation:

```javascript
function isValidMove(character, move) {
    // Implement client-side validation based on character type
    return true; // Return true or false
}
```

**5. Test and Deploy**

5.1. Testing

- Unit Testing: Write tests for game logic.

- Integration Testing: Test WebSocket communication.

## 6. Conclusion

This document outlines the development of a turn-based game played on a 5x5 grid using a server-client architecture. The server manages game state and validates moves, while the client displays the game and handles player input. WebSocket communication enables real-time updates between server and clients. Further enhancements could include additional characters or AI opponents.