CT117-3-2-FWDD Further Web Design and Development

Individual Assignment

Name	Darren Chey Wei Jun
Student ID	TP066934
Intake	APD2F2402SE
Project Title	Snake & Syntax: A Learning Adventure in Snake and Ladders

Table of Contents

1.0 Introduction

This project involves creating a hybrid game web application inspired by the classic board game, Snakes and Ladders. The goal is to build a fun digital experience that combines traditional game mechanics with engaging educational challenges. As players move along a virtual game board, they will roll dice to advance. When they land on specific spots, they will face Python questions. If they answer correctly, they gain an extra move, making the game both interactive and informative.

Objectives

1. Enhance Engagement: Design an interactive platform that that keeps players interested by mixing classic board game fun with new digital features.
2. Promote Learning: Integrate Python programming questions into the game for a fun and educational experience.
3. Ensure Accessibility: Design a user-friendly web app accessible to a larger audience, from younger users to Python learners of all ages.
4. Encourage Replayability: Develop a game structure that motivates players to return for multiple sessions by maintaining a balance between fun and educational challenges.

Scope

- Frontend Interface: A visually engaging and responsive interface using CSS and JavaScript, with dynamic elements rendered through Pug templates to create a consistent look and feel.
- Backend Logic: Server-side programming with Node.js handles game mechanics, user login, and data processing, while a phpMyAdmin database stores game records and user profiles.
- Interactive Gameplay: When players land on designated spaces, a prompt allows them to draw a card with a Python question. Correct answers reward bonus moves.
- **User Authentication and Data Tracking:** Players can log in to view their game stats and track how well they answer questions, making the experience more personal.

- **Logout with Confirmation:** A user-friendly logout feature includes a confirmation prompt and feedback, improving the overall experience.

End-User Specifications

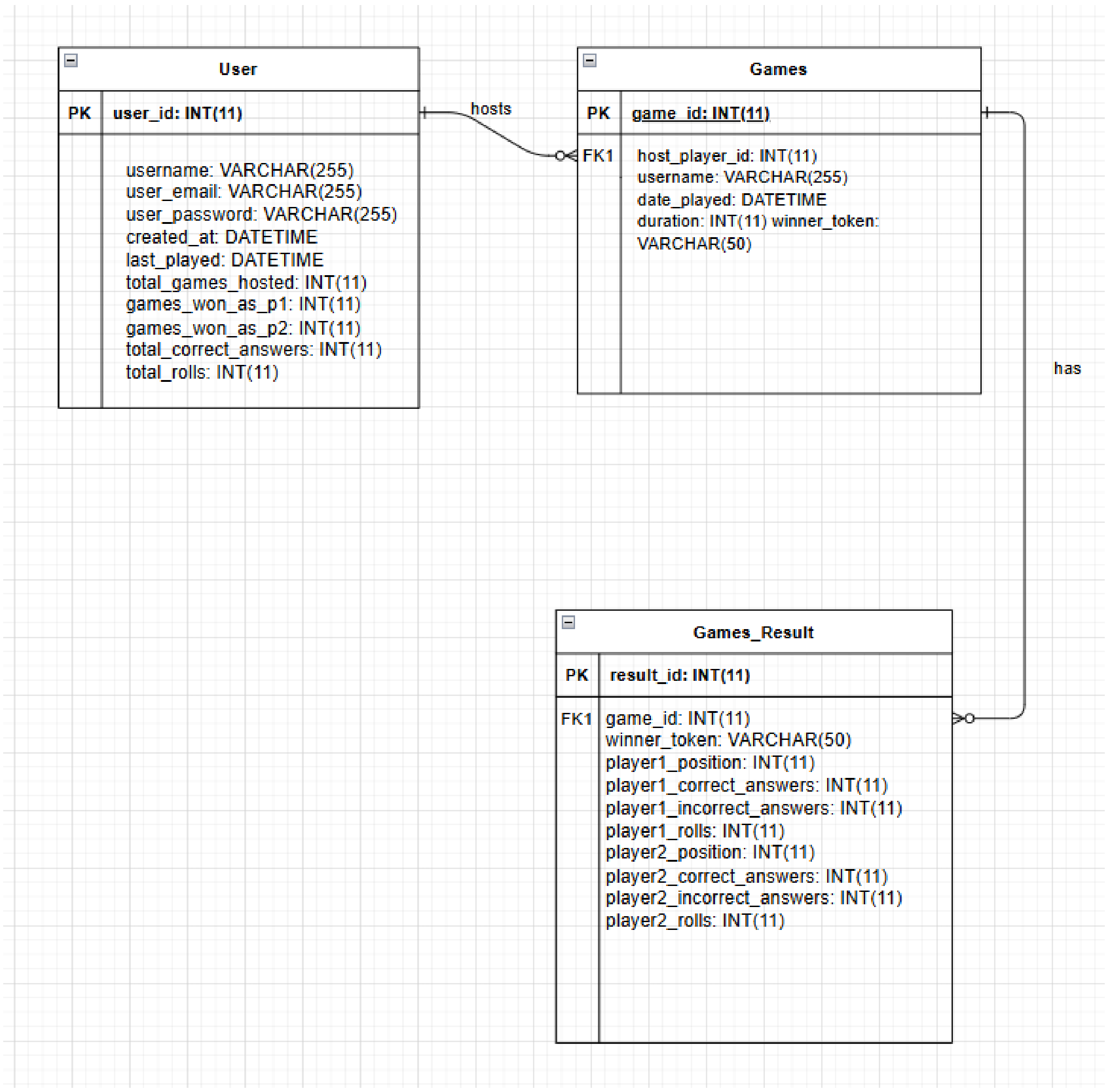
1. Students and Learners: Individuals interested in learning Python through an engaging, interactive game.
2. Educators: Teachers and institutions looking for a fun educational tool in a classroom or training setting.
3. Casual Gamers: Users seeking a traditional online board game experience enhanced with educational content.

Major Features of the Web App:

1. Game Board Navigation: Allows players to roll virtual dice and move along a Snakes and Ladders-style game board.
2. Question-Based Challenges: Prompts players to draw a card and answer a question upon landing on specific squares; correct answers grant extra moves or points.
3. User Authentication and Progress Tracking: Allows users to log in, track their performance, and receive specific feedback based on question accuracy.
4. Player Statistics: Shows performance measures like question correctness, improving the educational value of gameplay.
5. Responsive UI/UX: Optimized for both desktop and mobile devices, providing a consistent and smooth user experience across platforms.

2.0 Design

Entity Relationship Diagram (ERD)


User
PK user_id: INT(11)
username: VARCHAR(255)
user_email: VARCHAR(255)
user_password: VARCHAR(255)
created_at: DATETIME
last_played: DATETIME
total_games_hosted: INT(11)
games_won_as_p1: INT(11)
games_won_as_p2: INT(11)
total_correct_answers: INT(11)
total_rolls: INT(11)
hosts
Games
PK game_id: INT(11)
FK1 host_player_id: INT(11)
username: VARCHAR(255)
date_played: DATETIME
duration: INT(11) winner_token: VARCHAR(50)
has
Games_Result
PK result_id: INT(11)
FK1 game_id: INT(11)
winner_token: VARCHAR(50)
player1_position: INT(11)
player1_correct_answers: INT(11)
player1_incorrect_answers: INT(11)
player1_rolls: INT(11)
player2_position: INT(11)
player2_correct_answers: INT(11)
player2_incorrect_answers: INT(11)
player2_rolls: INT(11)

Users Table

This table holds information about users, including their ID, name, email, password, creation date, last played date, and statistics such as total games hosted, games won as player 1 or player 2, total correct answers, and total dice rolls.

Games Table

This table stores details about games, such as the game ID, the host player's ID, the host's username, the date played, the duration of the game, and who won (represented by a winner token).

Game Results Table

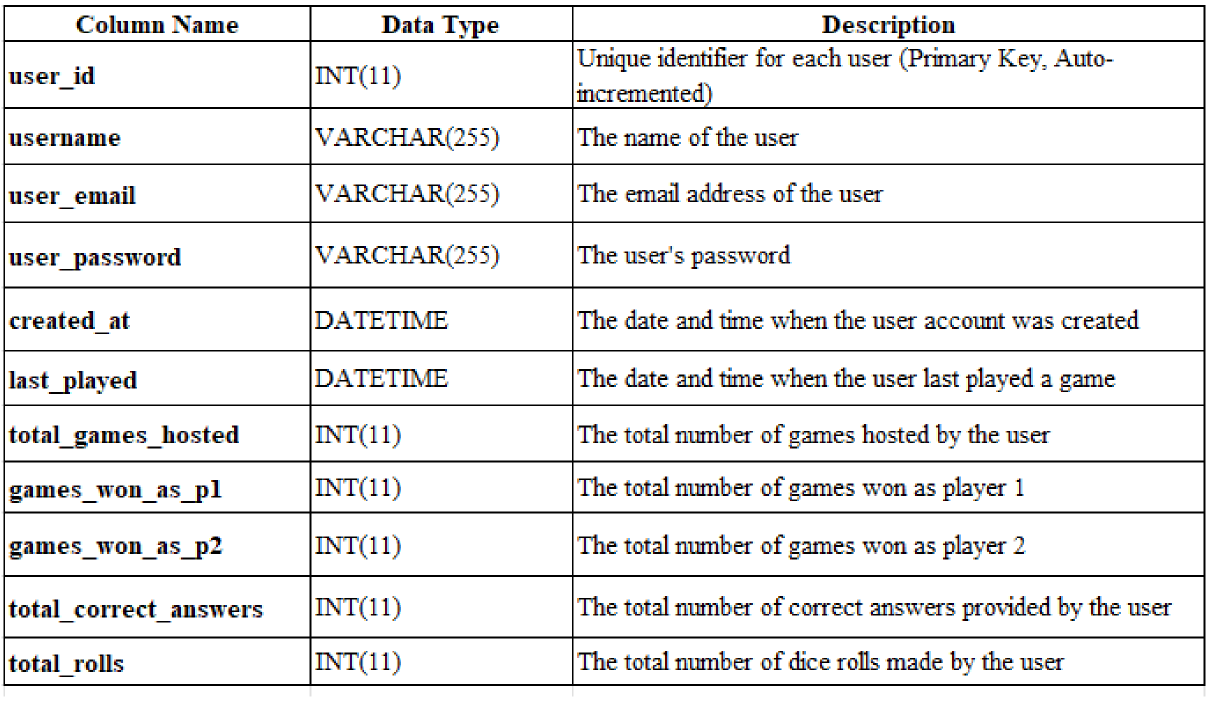
This table records the results of each game, including the game ID, the winner, and performance details for both player 1 and player 2, like their final positions, correct and incorrect answers, and total rolls.

Relationships

1. Users to Games: A user can host multiple games.
2. Games to Game Results: Each game has detailed results for both players.

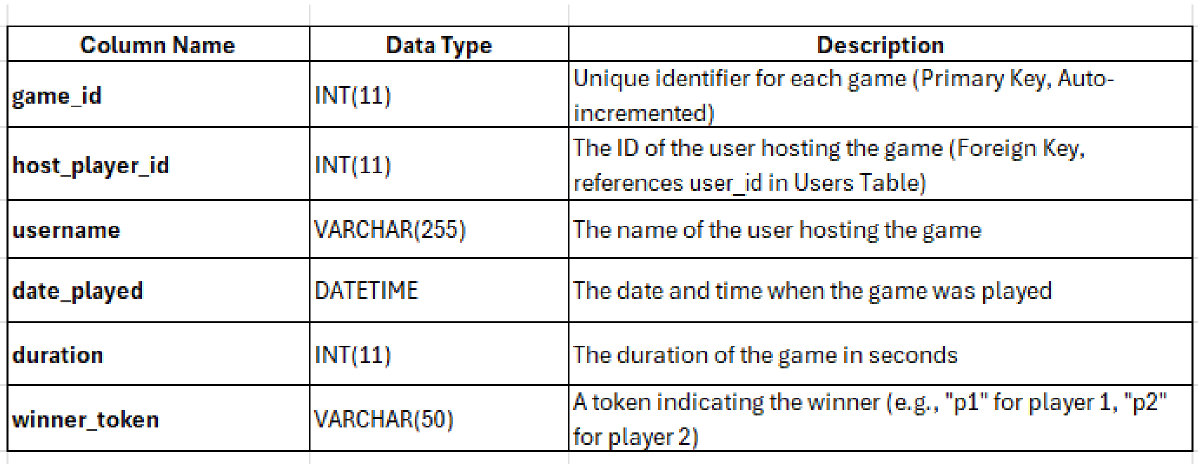
2.1 Data Dictionary

User Table



Column Name	Data Type	Description
user_id	INT(11)	Unique identifier for each user (Primary Key, Auto-incremented)
username	VARCHAR(255)	The name of the user
user_email	VARCHAR(255)	The email address of the user
user_password	VARCHAR(255)	The user's password
created_at	DATETIME	The date and time when the user account was created
last_played	DATETIME	The date and time when the user last played a game
total_games_hosted	INT(11)	The total number of games hosted by the user
games_won_as_p1	INT(11)	The total number of games won as player 1
games_won_as_p2	INT(11)	The total number of games won as player 2
total_correct_answers	INT(11)	The total number of correct answers provided by the user
total_rolls	INT(11)	The total number of dice rolls made by the user

Game Table



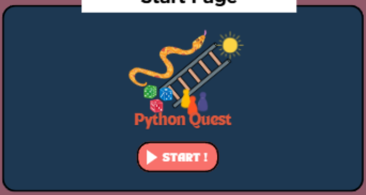
Column Name	Data Type	Description
game_id	INT(11)	Unique identifier for each game (Primary Key, Auto-incremented)
host_player_id	INT(11)	The ID of the user hosting the game (Foreign Key, references user_id in Users Table)
username	VARCHAR(255)	The name of the user hosting the game
date_played	DATETIME	The date and time when the game was played
duration	INT(11)	The duration of the game in seconds
winner_token	VARCHAR(50)	A token indicating the winner (e.g., "p1" for player 1, "p2" for player 2)

Game_results Table

Column Name	Data Type	Description
result_id	INT(11)	Unique identifier for each game result (Primary Key, Auto-incremented)
game_id	INT(11)	The ID of the game (Foreign Key, references game_id in Games Table)
winner_token	VARCHAR(50)	A token indicating the winner (e.g., "p1" for player 1, "p2" for player 2)
player1_position	INT(11)	The final position of player 1 in the game
player1_correct_answers	INT(11)	The total number of correct answers given by player 1
player1_incorrect_answers	INT(11)	The total number of incorrect answers given by player 1
player1_rolls	INT(11)	The total number of dice rolls made by player 1
player2_position	INT(11)	The final position of player 2 in the game
player2_correct_answers	INT(11)	The total number of correct answers given by player 2
player2_incorrect_answers	INT(11)	The total number of incorrect answers given by player 2
player2_rolls	INT(11)	The total number of dice rolls made by player 2

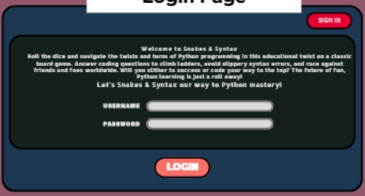
2.2 Storyboard

Start Page



Click the Start button to proceed.

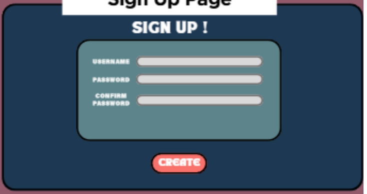
Login Page



Welcome to Snakes & Syntax
Roll the dice and navigate the twists and turns of Python programming in this educational twist on a classic board game. Answer coding questions to climb ladders, avoid slippery syntax errors, and race against friends and foes worldwide. Will you slither to success or code your way to the top? The future of fun, Python learning is just a roll away!
Let's Snakes & Syntax our way to Python mastery!

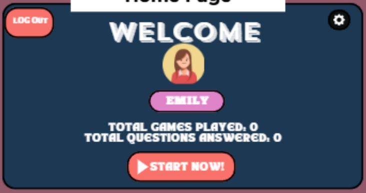
This is the login page. User should enter username and password to a [illegible] es.

Sign Up Page



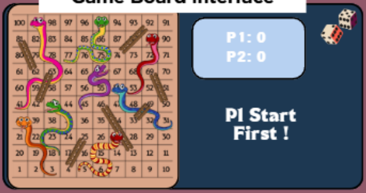
If the user does not have an account, they can create one by entering a username and password.

Home Page



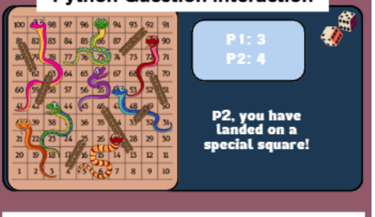
This page shows the user statistics. It Provide total questions answered, correct/incorrect answers, and time spent. It include a "Start Now" button to begin a new game.

Game Board Interface



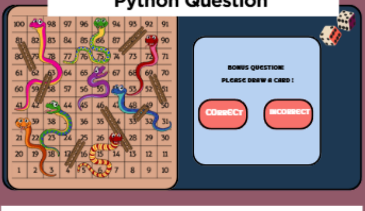
This page is the Game Board Interface. Players will take turns rolling the virtual dice. The system will move token according to dice roll. Players can check for their own position on the box above.

Python Question Interaction



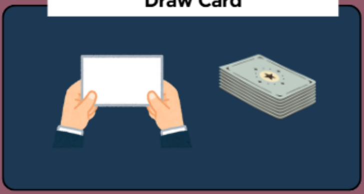
If landing on snake/ladder, player have to answer Python question. If correct, player can have another roll. If incorrect, player will miss the opportunity to roll.

Python Question



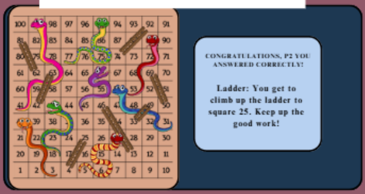
This page show the python question, player must answer it before advance to the next step.

Draw Card



This displays a card for the player to draw, containing a question they must answer.

Answer result

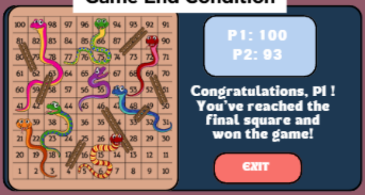


CONGRATULATIONS, P2 YOU ANSWERED CORRECTLY!

Ladder: You get to climb up the ladder to square 25. Keep up the good work!

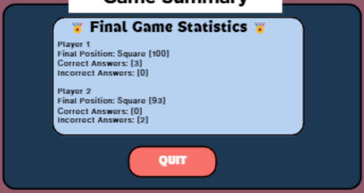
After the player answers, it will prompt a congratulations message for the player. If success, player can move up if not remain at the same place.

Game End Condition


P1: 100
P2: 93
Congratulations, PI ! You've reached the final square and won the game!
EXIT

Game ends when a player reaches the last square. System will declare winner based on position.

Game Summary



Final Game Statistics

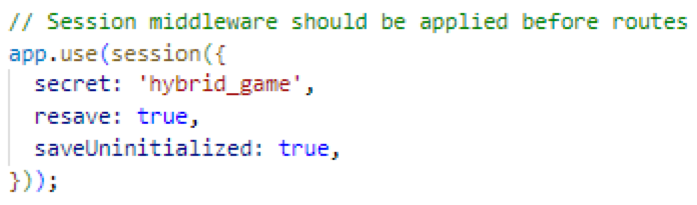
Player 1
Final Position: Square [100]
Correct Answers: [3]
Incorrect Answers: [0]

Player 2
Final Position: Square [93]
Correct Answers: [0]
Incorrect Answers: [2]

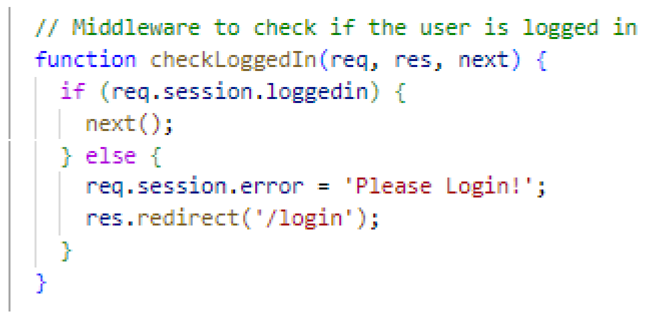
Player can view summary of game performance. It will provide total questions answered, correct/incorrect answers.

3.0 Implementation

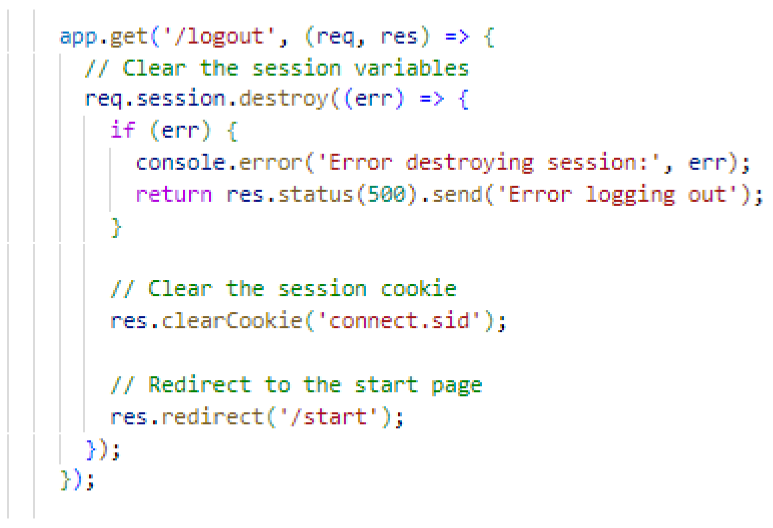
3.1 User Authentication & Session Management



```
// Session middleware should be applied before routes
app.use(session({
  secret: 'hybrid_game',
  resave: true,
  saveUninitialized: true,
}));
```



```
// Middleware to check if the user is logged in
function checkLoggedIn(req, res, next) {
  if (req.session.loggedin) {
    next();
  } else {
    req.session.error = 'Please Login!';
    res.redirect('/login');
  }
}
```

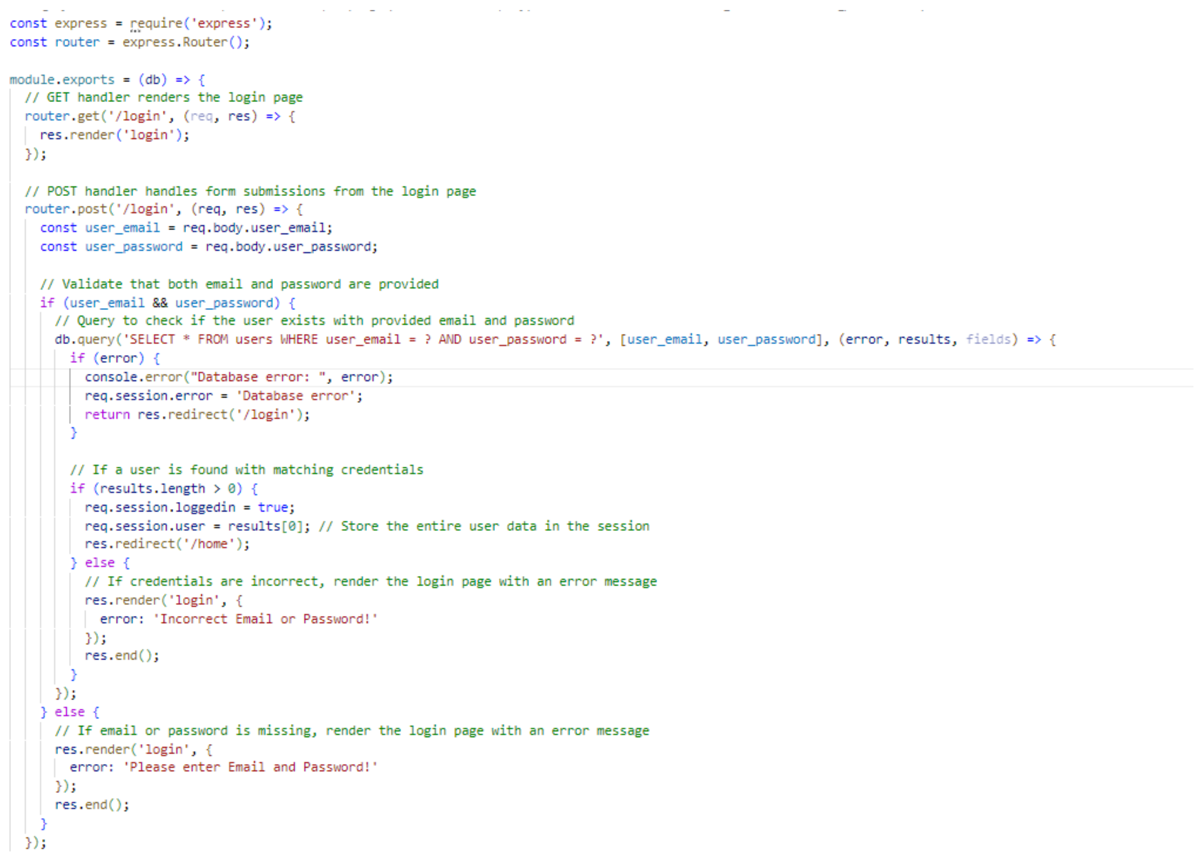


```
app.get('/logout', (req, res) => {
  // Clear the session variables
  req.session.destroy((err) => {
    if (err) {
      console.error('Error destroying session:', err);
      return res.status(500).send('Error logging out');
    }

    // Clear the session cookie
    res.clearCookie('connect.sid');

    // Redirect to the start page
    res.redirect('/start');
  });
});
```

Figure 1: User Authentication & Session

This code manages user sessions and handles login/logout functionality. It uses middleware to manage sessions with a secret key **('hybrid_game')**, ensuring the session is saved and maintained even if it's not initialized right away. The **checkLoggedIn** middleware checks if the user is logged in by verifying the session. If the user is logged in, the request proceeds to the next step. If not, they are redirected to the login page with an error message. The logout functionality is handled by the /logout route, which destroys the user's session, clears the session cookie, and redirects the user to the start page. If there's an error during logout, an error message is shown.

Login.js

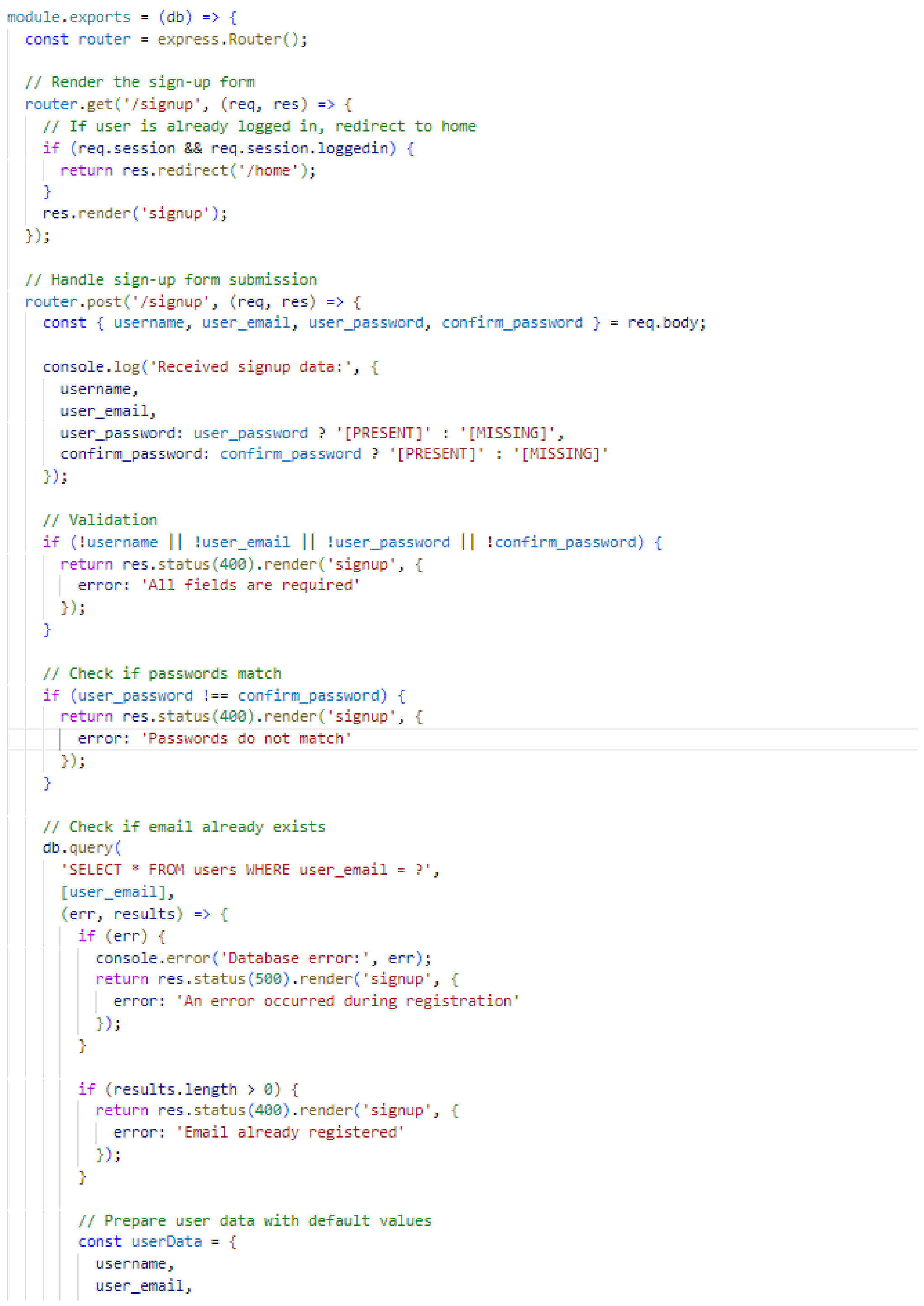


```
const express = require('express');
const router = express.Router();

module.exports = (db) => {
  // GET handler renders the login page
  router.get('/login', (req, res) => {
    res.render('login');
  });

  // POST handler handles form submissions from the login page
  router.post('/login', (req, res) => {
    const user_email = req.body.user_email;
    const user_password = req.body.user_password;

    // Validate that both email and password are provided
    if (user_email && user_password) {
      // Query to check if the user exists with provided email and password
      db.query('SELECT * FROM users WHERE user_email = ? AND user_password = ?', [user_email, user_password], (error, results, fields) => {
        if (error) {
          console.error("Database error: ", error);
          req.session.error = 'Database error';
          return res.redirect('/login');
        }

        // If a user is found with matching credentials
        if (results.length > 0) {
          req.session.loggedin = true;
          req.session.user = results[0]; // Store the entire user data in the session
          res.redirect('/home');
        } else {
          // If credentials are incorrect, render the login page with an error message
          res.render('login', {
            error: 'Incorrect Email or Password!'
          });
          res.end();
        }
      });
    } else {
      // If email or password is missing, render the login page with an error message
      res.render('login', {
        error: 'Please enter Email and Password!'
      });
      res.end();
    }
  });
```

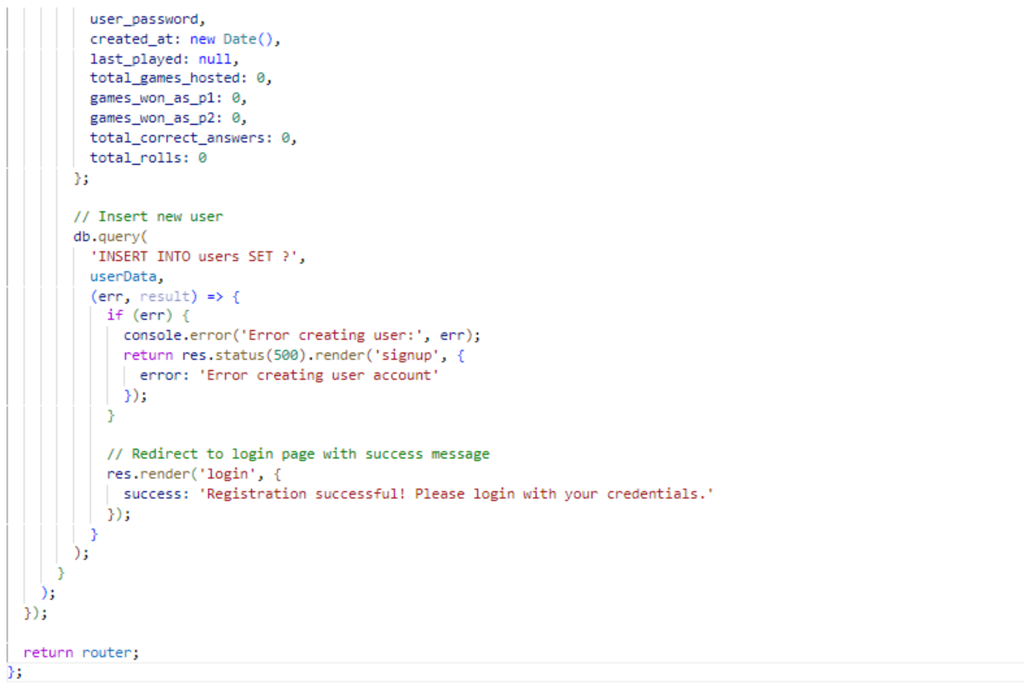
Figure 2: Login.js

This code handles the **login process** for a web application. When a user visits the **/login** page, it displays a form where they can enter their email and password. When the user submits the form, the app checks if both email and password are provided. If they are, it queries the database to find a matching user with the same email and password. If a match is found, the user is logged in, their details are saved in the session, and they are redirected to the **/home** page. If the credentials are incorrect, the login page is shown again with an error message. If there's an issue with the database, an error message is displayed, and the user is redirected back to the login page. The router is returned to be used in the main application to manage the login route.

Signup.js



```
module.exports = (db) => {
  const router = express.Router();

  // Render the sign-up form
  router.get('/signup', (req, res) => {
    // If user is already logged in, redirect to home
    if (req.session && req.session.loggedin) {
      return res.redirect('/home');
    }
    res.render('signup');
  });

  // Handle sign-up form submission
  router.post('/signup', (req, res) => {
    const { username, user_email, user_password, confirm_password } = req.body;

    console.log('Received signup data:', {
      username,
      user_email,
      user_password: user_password ? '[PRESENT]' : '[MISSING]',
      confirm_password: confirm_password ? '[PRESENT]' : '[MISSING]'
    });

    // Validation
    if (!username || !user_email || !user_password || !confirm_password) {
      return res.status(400).render('signup', {
        error: 'All fields are required'
      });
    }

    // Check if passwords match
    if (user_password !== confirm_password) {
      return res.status(400).render('signup', {
        error: 'Passwords do not match'
      });
    }

    // Check if email already exists
    db.query(
      'SELECT * FROM users WHERE user_email = ?',
      [user_email],
      (err, results) => {
        if (err) {
          console.error('Database error:', err);
          return res.status(500).render('signup', {
            error: 'An error occurred during registration'
          });
        }

        if (results.length > 0) {
          return res.status(400).render('signup', {
            error: 'Email already registered'
          });
        }

        // Prepare user data with default values
        const userData = {
          username,
          user_email,
```



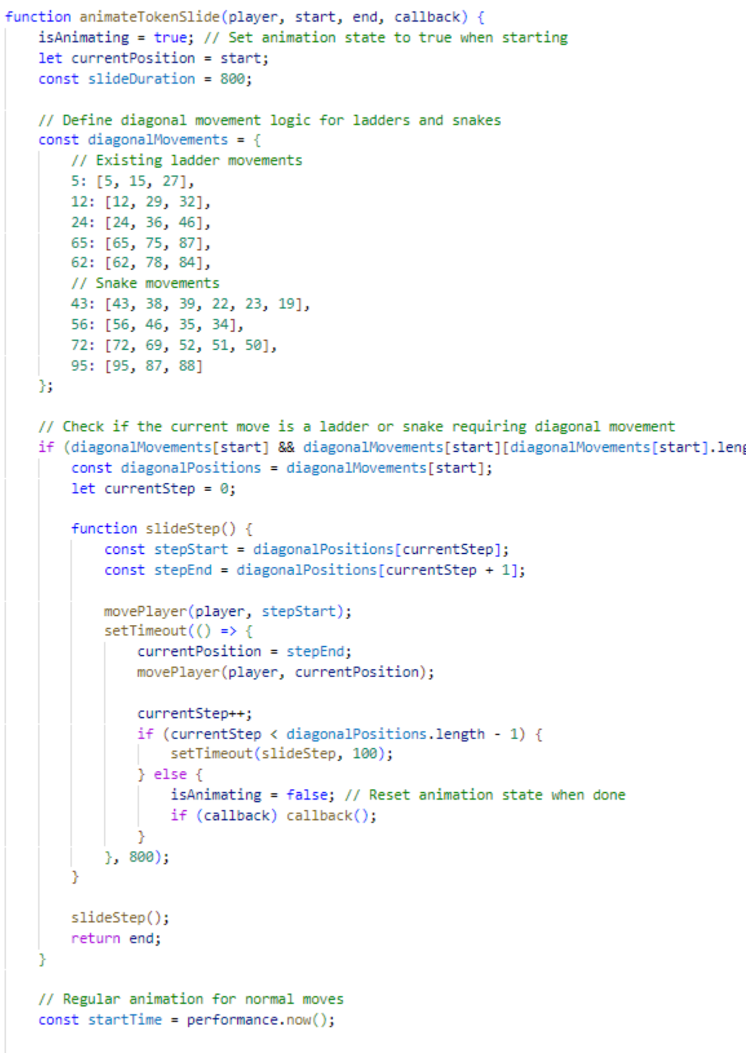
```
                user_password,
                created_at: new Date(),
                last_played: null,
                total_games_hosted: 0,
                games_won_as_p1: 0,
                games_won_as_p2: 0,
                total_correct_answers: 0,
                total_rolls: 0
            };

            // Insert new user
            db.query(
                'INSERT INTO users SET ?',
                userData,
                (err, result) => {
                    if (err) {
                        console.error('Error creating user:', err);
                        return res.status(500).render('signup', {
                            error: 'Error creating user account'
                        });
                    }

                    // Redirect to login page with success message
                    res.render('login', {
                        success: 'Registration successful! Please login with your credentials.'
                    });
                }
            );
        }
    );
  });

  return router;
};
```

Figure 3: SignUp.js

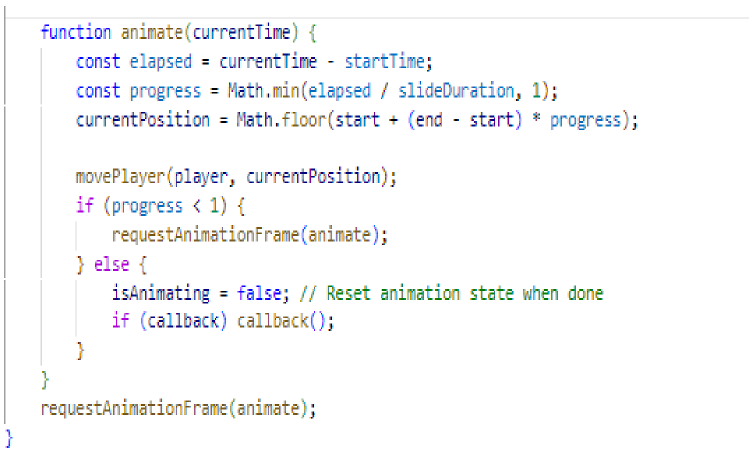
This code manages the user sign-up process for a web application. When a user visits the /signup page, the app checks if the user is already logged in by looking at the session (req.session.loggedin). If the user is logged in, they are redirected to the /home page. If not, it renders the sign-up form so the user can create an account. When the user submits the form, the app checks if all fields are filled in, and if the password and confirm password match. If anything is missing or the passwords do not match, an error message is shown. It also checks if the email is already registered. If the email exists, an error is displayed. If everything is correct, the app saves the user's information in the database and shows a success message, directing the user to the login page. If there are any issues during registration, appropriate error messages are shown.

3.2 Board/Visual Functions

function animateTokenSlide(player, start, end, callback)



```
function animateTokenSlide(player, start, end, callback) {
    isAnimating = true; // Set animation state to true when starting
    let currentPosition = start;
    const slideDuration = 800;

    // Define diagonal movement logic for ladders and snakes
    const diagonalMovements = {
        // Existing ladder movements
        5: [5, 15, 27],
        12: [12, 29, 32],
        24: [24, 36, 46],
        65: [65, 75, 87],
        62: [62, 78, 84],
        // Snake movements
        43: [43, 38, 39, 22, 23, 19],
        56: [56, 46, 35, 34],
        72: [72, 69, 52, 51, 50],
        95: [95, 87, 88]
    };

    // Check if the current move is a ladder or snake requiring diagonal movement
    if (diagonalMovements[start] && diagonalMovements[start][diagonalMovements[start].leng
        const diagonalPositions = diagonalMovements[start];
        let currentStep = 0;

        function slideStep() {
            const stepStart = diagonalPositions[currentStep];
            const stepEnd = diagonalPositions[currentStep + 1];

            movePlayer(player, stepStart);
            setTimeout(() => {
                currentPosition = stepEnd;
                movePlayer(player, currentPosition);

                currentStep++;
                if (currentStep < diagonalPositions.length - 1) {
                    setTimeout(slideStep, 100);
                } else {
                    isAnimating = false; // Reset animation state when done
                    if (callback) callback();
                }
            }, 800);
        }

        slideStep();
        return end;
    }

    // Regular animation for normal moves
    const startTime = performance.now();
```



```
    function animate(currentTime) {
        const elapsed = currentTime - startTime;
        const progress = Math.min(elapsed / slideDuration, 1);
        currentPosition = Math.floor(start + (end - start) * progress);

        movePlayer(player, currentPosition);
        if (progress < 1) {
            requestAnimationFrame(animate);
        } else {
            isAnimating = false; // Reset animation state when done
            if (callback) callback();
        }
    }
    requestAnimationFrame(animate);
}
```

Figure 4: animateTokenSlide

This code defines a function, **animateTokenSlide**, that smoothly moves a player's token across the board. It animates the token from a starting position to an ending position using either straight-line or diagonal movement, depending on whether the move involves a ladder or a snake.

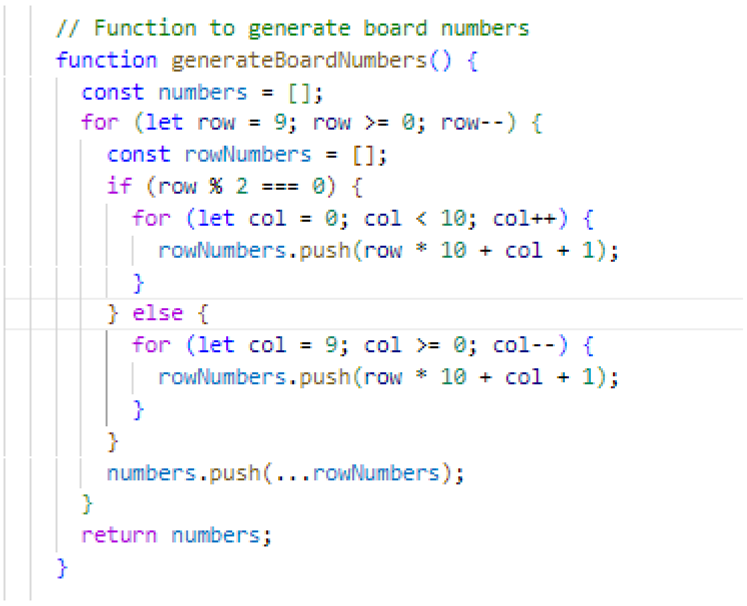
The function first sets **isAnimating** to true, indicating that an animation is in progress and preventing other actions until it completes. It initializes the player's current position and specifies an animation duration of 800 milliseconds for each move.

The **diagonalMovements** object shows specific sequences for ladder and snake moves that require more complex paths. For example, moving from position 5 via a ladder involves stepping through intermediate positions 5, 15, and finally reaching 27. Similarly, moving down from position 43 via a snake involves stopping at positions 38, 39, 22, 23, and ending at 19.

If the current move is a ladder or snake, the function enters a specialized animation mode using **slideStep**, a nested function that moves the player token along each step in the diagonal path. It repeatedly updates the token's position, waiting 800 milliseconds at each step before advancing to the next until it reaches the final position. After finishing, **isAnimating** is reset to **false**, and the **callback** function (if provided) is called.

For moves that not involving ladders or snakes, the function calculates the starting time and uses it to create a smooth animation. This structure makes the function adaptable to different movement types, ensuring visually consistent gameplay.

Function generateBoardNumbers

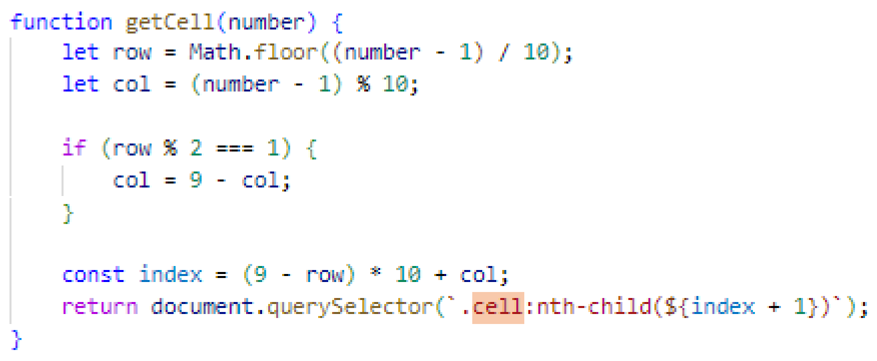


```
// Function to generate board numbers
function generateBoardNumbers() {
  const numbers = [];
  for (let row = 9; row >= 0; row--) {
    const rowNumbers = [];
    if (row % 2 === 0) {
      for (let col = 0; col < 10; col++) {
        rowNumbers.push(row * 10 + col + 1);
      }
    } else {
      for (let col = 9; col >= 0; col--) {
        rowNumbers.push(row * 10 + col + 1);
      }
    }
    numbers.push(...rowNumbers);
  }
  return numbers;
}
```

Figure 5: Function generateBoardNumbers

The **generateBoardNumbers** function creates the numbers for a Snakes and Ladders game board, which consists of 100 cells arranged in a 10x10 grid. It loops through each row of the grid, starting from the bottom row and moving upward. For even-numbered rows, the numbers are assigned from left to right, moving from the first column to the last. For odd-numbered rows, the numbers are assigned from right to left, moving from the last column to the first. The function builds a list of all the numbers in this pattern, ranging from 1 to 100, and returns this list, which represents the cells on the board.

function getCell(number)



```javascript
function getCell(number) {
    let row = Math.floor((number - 1) / 10);
    let col = (number - 1) % 10;

    if (row % 2 === 1) {
        col = 9 - col;
    }

    const index = (9 - row) * 10 + col;
    return document.querySelector(`.cell:nth-child(${index + 1})`);
}
```

Figure 6: function get Cell

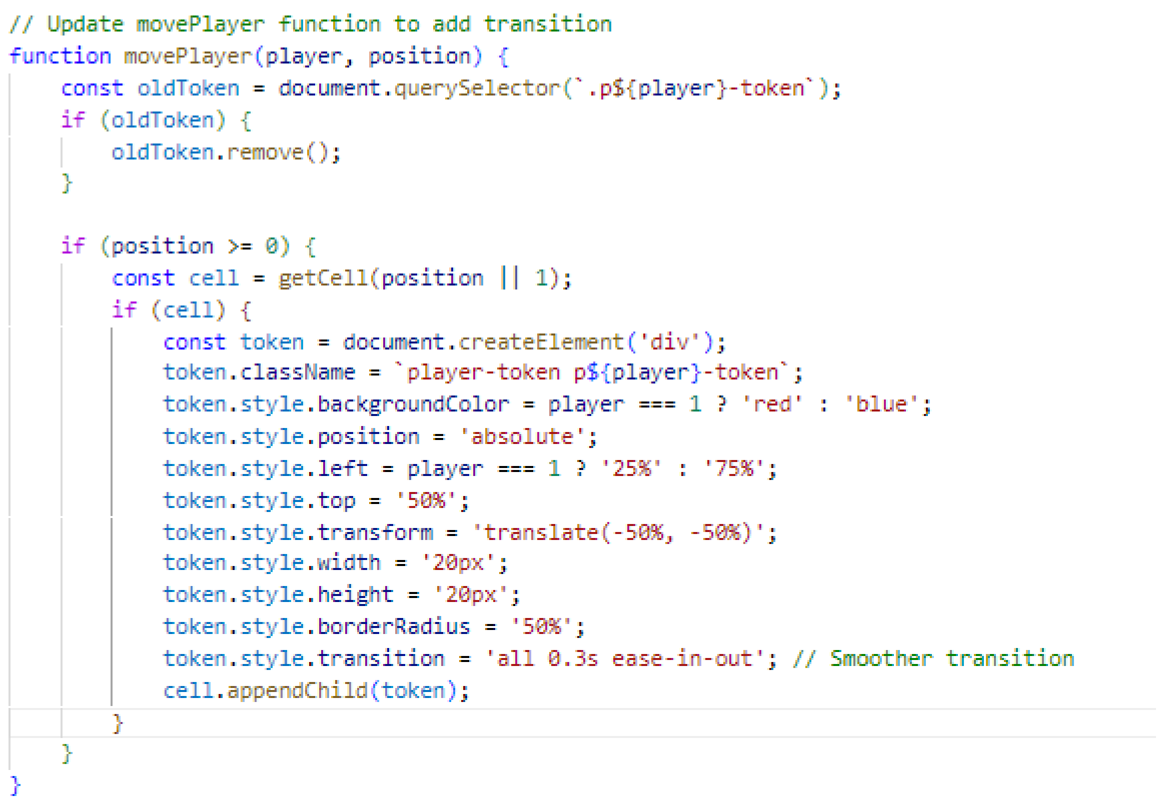
The **getCell** function is used to locate the correct HTML element for a cell on a Snake and Ladders board based on its cell number, considering the board for a zigzag pattern.

First, it **calculates the row and column** for the cell. The **row** is determined by dividing the cell number by 10, and the **column** is the remainder when dividing by 10. It is because every other row on the board is reversed to create a zigzag layout, the function checks if the row number is odd. If it is, it flips the column value by calculating **9 - col** to ensure it points to the correct spot in the reversed row.

After that, it calculates an overall board index based on the row and column, using **9 - row** to start counting rows from the bottom of the board to match the zigzag layout.

Finally, it uses this index to find and return the HTML element for the cell on the board. This function helps the game access any cell on the board, accurately reflecting the direction of rows.

function movePlayer



```
// Update movePlayer function to add transition
function movePlayer(player, position) {
    const oldToken = document.querySelector(`.p${player}-token`);
    if (oldToken) {
        oldToken.remove();
    }

    if (position >= 0) {
        const cell = getCell(position || 1);
        if (cell) {
            const token = document.createElement('div');
            token.className = `player-token p${player}-token`;
            token.style.backgroundColor = player === 1 ? 'red' : 'blue';
            token.style.position = 'absolute';
            token.style.left = player === 1 ? '25%' : '75%';
            token.style.top = '50%';
            token.style.transform = 'translate(-50%, -50%)';
            token.style.width = '20px';
            token.style.height = '20px';
            token.style.borderRadius = '50%';
            token.style.transition = 'all 0.3s ease-in-out'; // Smoother transition
            cell.appendChild(token);
        }
    }
}
```

Figure 7: function movePlayer

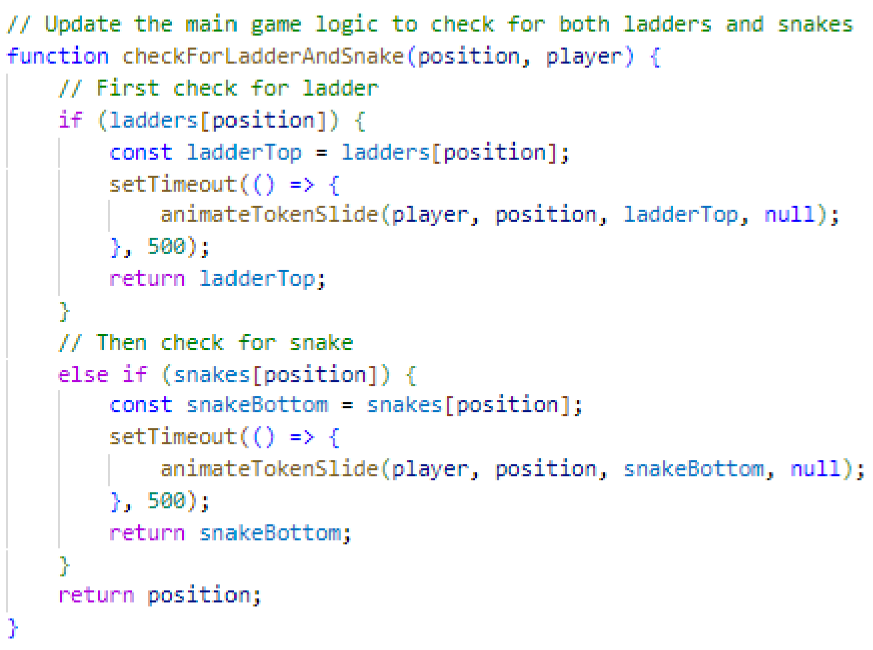
The **movePlayer** function is designed to move a player's token smoothly to a new position on the game board. It first checks if the player already has a token on the board and, if so, removes the old one to avoid duplicates. Then, it determines which cell the token should move to by calling another function which is **getCell**, that finds the HTML element for the specific cell on the board.

Once the cell is found, the function creates a new **div** element that represents the player's token. The token's position is offset slightly depending on the player to help differentiate between the two tokens if they happen to land on the same cell.

To make the movement smooth, a transition effect is added, allowing the token to slide into place with a slight animation. Finally, the token is added to the board in the correct cell, making it visible at the new location.

3.3 Game Logic Functions

Function checkForLadderAndSnake



```
// Update the main game logic to check for both ladders and snakes
function checkForLadderAndSnake(position, player) {
    // First check for ladder
    if (ladders[position]) {
        const ladderTop = ladders[position];
        setTimeout(() => {
            animateTokenSlide(player, position, ladderTop, null);
        }, 500);
        return ladderTop;
    }
    // Then check for snake
    else if (snakes[position]) {
        const snakeBottom = snakes[position];
        setTimeout(() => {
            animateTokenSlide(player, position, snakeBottom, null);
        }, 500);
        return snakeBottom;
    }
    return position;
}
```

Figure 8 function checkForLadderAndSnake

The **checkForLadderAndSnake** function checks whether a player's current position is at the base of a ladder or the head of a snake and handles the movement accordingly.

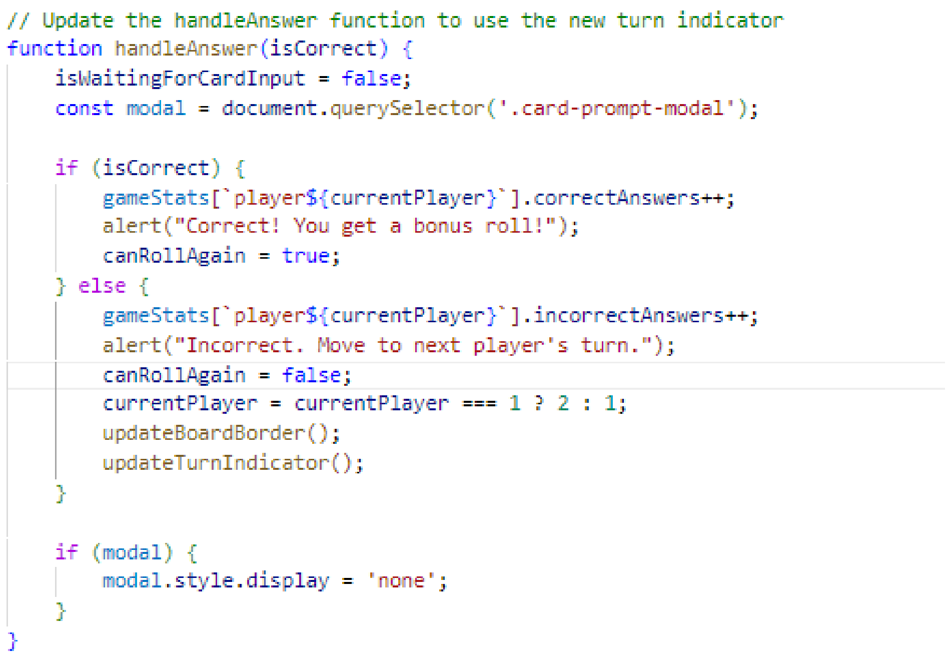
First, it checks if the player is on a ladder by looking up the current position in the **ladders** object. If the position match to the base of a ladder, it finds the top of the ladder and animates the player's token moving up to that position. The animation is delayed by 500 milliseconds to make the transition more noticeable. Once the animation is complete, the player's new position is returned as the top of the ladder.

If there's no ladder at the current position, the function checks for a snake instead. If the player is on the head of a snake, it retrieves the bottom of the snake and animates the player's token sliding down to that position. Again, the player's new position is returned as the bottom of the snake.

If no ladder or snake is found at the position, the function simply returns the current position, meaning the player stays in the same spot.

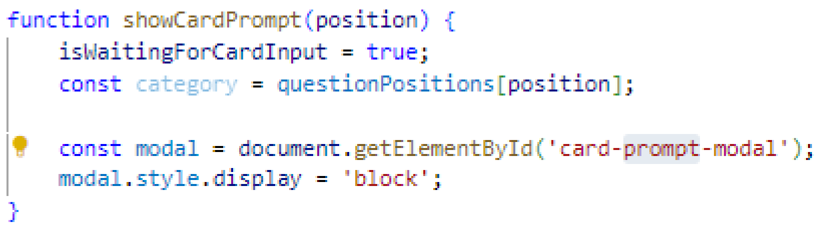
3.4 Question/Card System Functions

Function handleAnswer and Function showCardPrompt



```
// Update the handleAnswer function to use the new turn indicator
function handleAnswer(isCorrect) {
    isWaitingForCardInput = false;
    const modal = document.querySelector('.card-prompt-modal');

    if (isCorrect) {
        gameStats[`player${currentPlayer}`].correctAnswers++;
        alert("Correct! You get a bonus roll!");
        canRollAgain = true;
    } else {
        gameStats[`player${currentPlayer}`].incorrectAnswers++;
        alert("Incorrect. Move to next player's turn.");
        canRollAgain = false;
        currentPlayer = currentPlayer === 1 ? 2 : 1;
        updateBoardBorder();
        updateTurnIndicator();
    }

    if (modal) {
        modal.style.display = 'none';
    }
}
```

Figure 9: Function handleAnswer



```
function showCardPrompt(position) {
    isWaitingForCardInput = true;
    const category = questionPositions[position];

    const modal = document.getElementById('card-prompt-modal');
    modal.style.display = 'block';
}
```

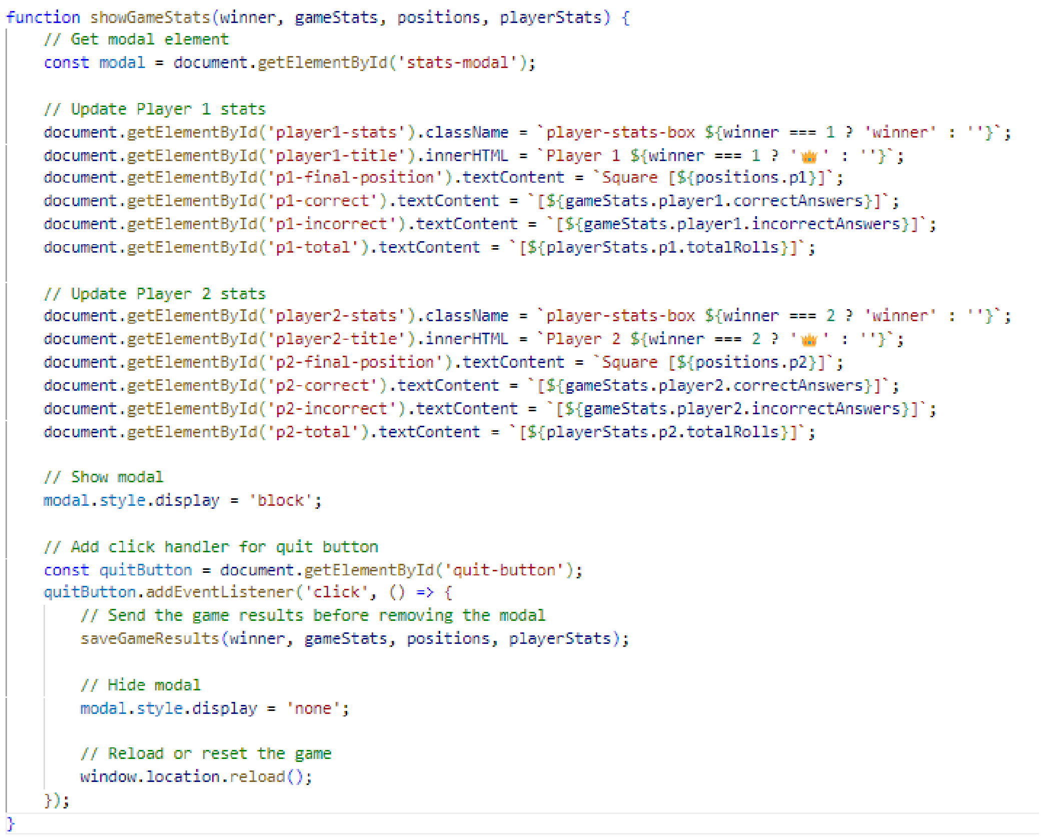
Figure 10: Function showCardPrompt

The **handleAnswer** function is used to process the player's response to a question. If the player answers correctly, their correct answer count increases, and they get a bonus roll. If the answer is incorrect, their incorrect answer count increases, and the turn switches to the other player. It also updates the game's board and turn indicators. The question modal is hidden after the answer is handled.

The **showCardPrompt** function is triggered when a player lands on a special spot on the board that requires answering a question. It shows a modal (popup) asking the player to answer, and it sets up the game to wait for the player's input. The specific question is fetched based on the player's position on the board. The modal is displayed, prompting the player to answer.

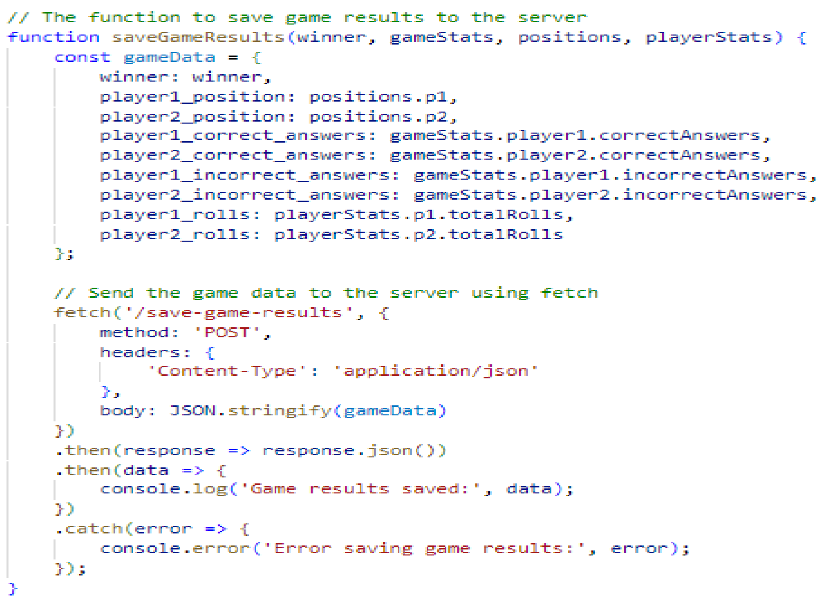
3.5 End Game Functions

Function showGameStats and saveGameResults



```javascript
function showGameStats(winner, gameStats, positions, playerStats) {
    // Get modal element
    const modal = document.getElementById('stats-modal');

    // Update Player 1 stats
    document.getElementById('player1-stats').className = `player-stats-box ${winner === 1 ? 'winner' : ''}`;
    document.getElementById('player1-title').innerHTML = `Player 1 ${winner === 1 ? '👑' : ''}`;
    document.getElementById('p1-final-position').textContent = `Square [${positions.p1}]`;
    document.getElementById('p1-correct').textContent = `[${gameStats.player1.correctAnswers}]`;
    document.getElementById('p1-incorrect').textContent = `[${gameStats.player1.incorrectAnswers}]`;
    document.getElementById('p1-total').textContent = `[${playerStats.p1.totalRolls}]`;

    // Update Player 2 stats
    document.getElementById('player2-stats').className = `player-stats-box ${winner === 2 ? 'winner' : ''}`;
    document.getElementById('player2-title').innerHTML = `Player 2 ${winner === 2 ? '👑' : ''}`;
    document.getElementById('p2-final-position').textContent = `Square [${positions.p2}]`;
    document.getElementById('p2-correct').textContent = `[${gameStats.player2.correctAnswers}]`;
    document.getElementById('p2-incorrect').textContent = `[${gameStats.player2.incorrectAnswers}]`;
    document.getElementById('p2-total').textContent = `[${playerStats.p2.totalRolls}]`;

    // Show modal
    modal.style.display = 'block';

    // Add click handler for quit button
    const quitButton = document.getElementById('quit-button');
    quitButton.addEventListener('click', () => {
        // Send the game results before removing the modal
        saveGameResults(winner, gameStats, positions, playerStats);

        // Hide modal
        modal.style.display = 'none';

        // Reload or reset the game
        window.location.reload();
    });
}
```

Figure 11: Function showGameStats



```javascript
// The function to save game results to the server
function saveGameResults(winner, gameStats, positions, playerStats) {
    const gameData = {
        winner: winner,
        player1_position: positions.p1,
        player2_position: positions.p2,
        player1_correct_answers: gameStats.player1.correctAnswers,
        player2_correct_answers: gameStats.player2.correctAnswers,
        player1_incorrect_answers: gameStats.player1.incorrectAnswers,
        player2_incorrect_answers: gameStats.player2.incorrectAnswers,
        player1_rolls: playerStats.p1.totalRolls,
        player2_rolls: playerStats.p2.totalRolls
    };

    // Send the game data to the server using fetch
    fetch('/save-game-results', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(gameData)
    })
    .then(response => response.json())
    .then(data => {
        console.log('Game results saved:', data);
    })
    .catch(error => {
        console.error('Error saving game results:', error);
    });
}
```

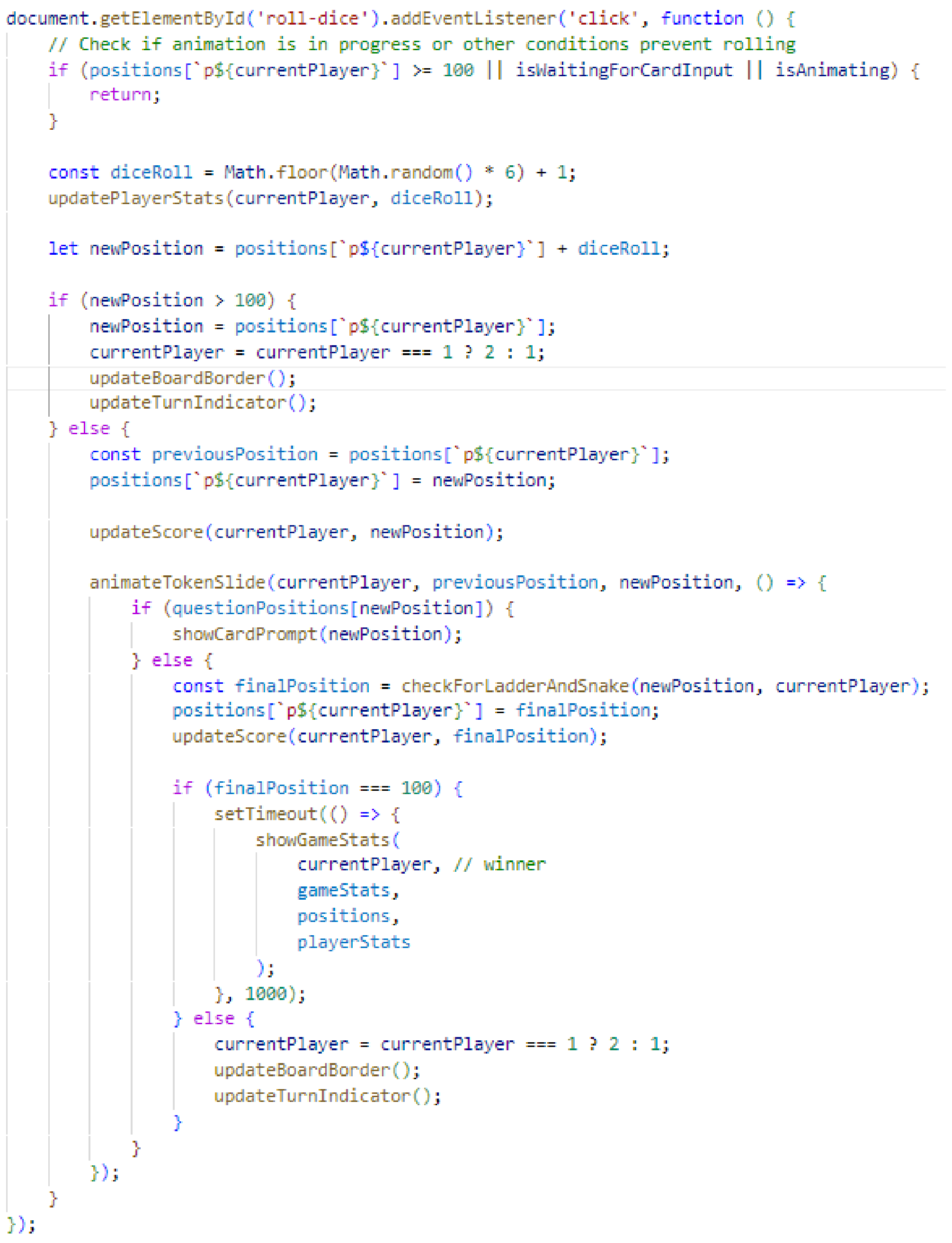
Figure 12: Function saveGameResults

The code above are **showGameStats** and **saveGameResults**. The **showGameStats** function is for displaying the final game results in a modal. It updates the statistics for both players, showing their final positions, the number of correct and incorrect answers, and their total rolls. Moreover, it highlights the winner by adding an emoji. When the player clicks the "quit" button, the function saves the game results using **saveGameResults**, hides the modal, and reloads the game to start over.

On the other hand, the **saveGameResults** function sends the collected game data, including the winner, players' positions, and stats, to the server. It constructs a **gameData** object and sends it via a POST request using the fetch API. This allows the server to store the results for future reference. The response from the server is logged to the console, indicating whether the save operation was successful or not.

3.6 Main Event Handlers

Dice Roll Event Handler



```
document.getElementById('roll-dice').addEventListener('click', function () {
    // Check if animation is in progress or other conditions prevent rolling
    if (positions[`p${currentPlayer}`] >= 100 || isWaitingForCardInput || isAnimating) {
        return;
    }

    const diceRoll = Math.floor(Math.random() * 6) + 1;
    updatePlayerStats(currentPlayer, diceRoll);

    let newPosition = positions[`p${currentPlayer}`] + diceRoll;

    if (newPosition > 100) {
        newPosition = positions[`p${currentPlayer}`];
        currentPlayer = currentPlayer === 1 ? 2 : 1;
        updateBoardBorder();
        updateTurnIndicator();
    } else {
        const previousPosition = positions[`p${currentPlayer}`];
        positions[`p${currentPlayer}`] = newPosition;

        updateScore(currentPlayer, newPosition);

        animateTokenSlide(currentPlayer, previousPosition, newPosition, () => {
            if (questionPositions[newPosition]) {
                showCardPrompt(newPosition);
            } else {
                const finalPosition = checkForLadderAndSnake(newPosition, currentPlayer);
                positions[`p${currentPlayer}`] = finalPosition;
                updateScore(currentPlayer, finalPosition);

                if (finalPosition === 100) {
                    setTimeout(() => {
                        showGameStats(
                            currentPlayer, // winner
                            gameStats,
                            positions,
                            playerStats
                        );
                    }, 1000);
                } else {
                    currentPlayer = currentPlayer === 1 ? 2 : 1;
                    updateBoardBorder();
                    updateTurnIndicator();
                }
            }
        });
    }
});
```

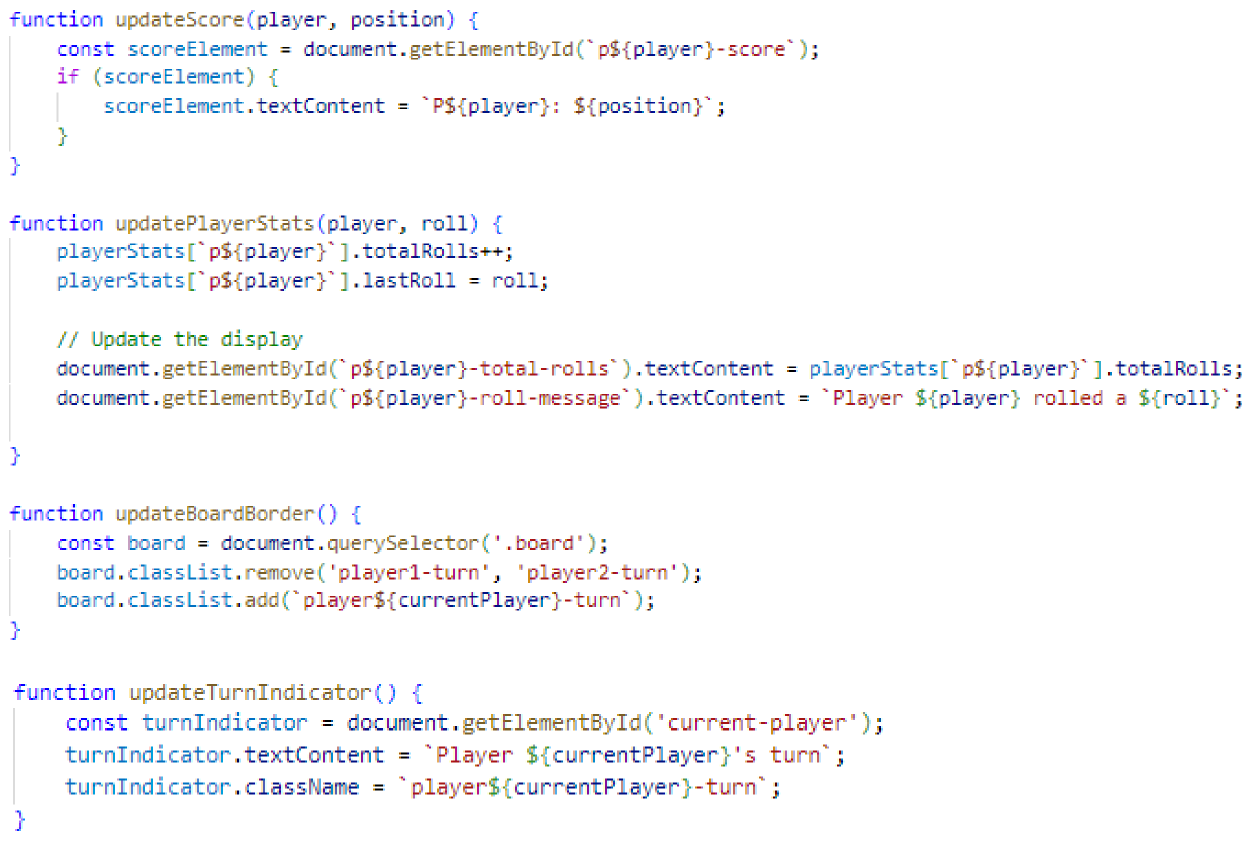
Figure 13: Dice Roll Event Handler

The code listens for a click event on the "roll-dice" button and initiates the rolling process for the game. First, it checks if certain conditions prevent the dice roll: if the current player has already reached position 100, if the game is waiting for card input, or if an animation is in progress. If none of these conditions are met, a dice roll is simulated by generating a random number between 1 and 6.

The new position for the player is then calculated by adding the dice roll to their current position. If this new position exceeds 100, the player's position remains the same, and the turn switches to the other player. Otherwise, the player's position is updated, and the score is adjusted accordingly.

The **animateTokenSlide** function is called to animate the player's token moving to the new position. After the animation completes, it checks if the new position has a question card. If yes, the **showCardPrompt** function is triggered to display the card. If there is no card, it checks if the new position corresponds to a ladder or snake using **checkForLadderAndSnake** and updates the position accordingly. If the final position is 100, the game results are displayed, marking the player as the winner. Otherwise, the turn switches to the other player, and the board is updated to reflect the new turn.

3.7 Game State Management Functions



```
function updateScore(player, position) {
    const scoreElement = document.getElementById(`p${player}-score`);
    if (scoreElement) {
        scoreElement.textContent = `P${player}: ${position}`;
    }
}

function updatePlayerStats(player, roll) {
    playerStats[`p${player}`].totalRolls++;
    playerStats[`p${player}`].lastRoll = roll;

    // Update the display
    document.getElementById(`p${player}-total-rolls`).textContent = playerStats[`p${player}`].totalRolls;
    document.getElementById(`p${player}-roll-message`).textContent = `Player ${player} rolled a ${roll}`;

}

function updateBoardBorder() {
    const board = document.querySelector('.board');
    board.classList.remove('player1-turn', 'player2-turn');
    board.classList.add(`player${currentPlayer}-turn`);
}

function updateTurnIndicator() {
    const turnIndicator = document.getElementById('current-player');
    turnIndicator.textContent = `Player ${currentPlayer}'s turn`;
    turnIndicator.className = `player${currentPlayer}-turn`;
}
```

Figure 14: Game State Management Functions

updateTurnIndicator: This function updates the turn indicator on the screen to reflect which player's turn it is. It sets the text content to show "Player 1's turn" or "Player 2's turn" based on the **currentPlayer**. It also updates the CSS class to reflect the current player's turn.

updateScore: This function updates the player's score in the UI. It targets an element with the **ID p{player}-score, where {player}** is either 1 or 2. It then updates the text content to show the player's current position.

updatePlayerStats: This function updates the player's total rolls and the last roll they made. It increments the total rolls for the specified player and updates the displayed stats, including the total rolls and a message indicating the roll message.

updateBoardBorder: This function highlights the current player's side of the board by updating the border styles, making it clear who is taking their turn.

4.0 User Guidance

4.1 Start page

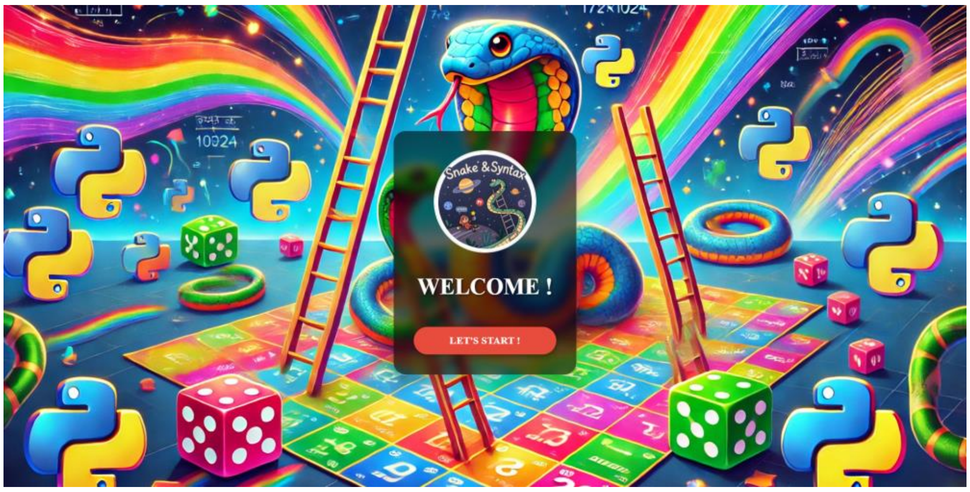
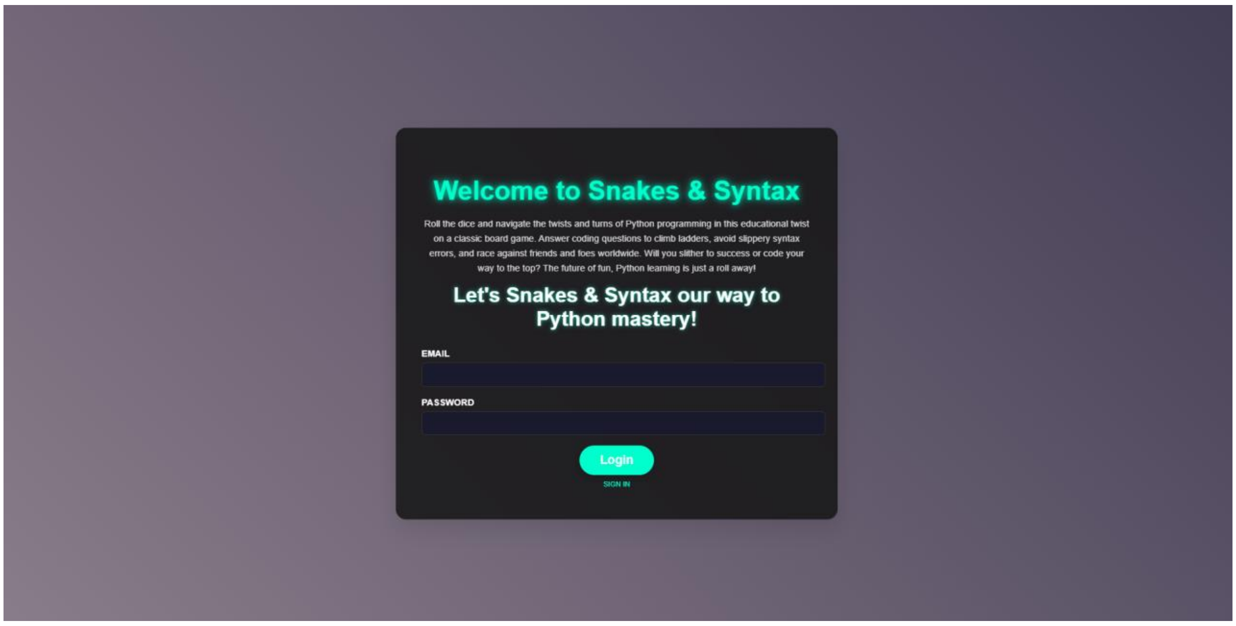


Figure 15: Start page

This is the landing page of your web application, a welcoming user with a simple design. It includes a large logo alongside a heading that says "WELCOME!". Below the heading, there is a button-like link with the text "LET'S START!" that directs users to the login page where they can begin their journey.

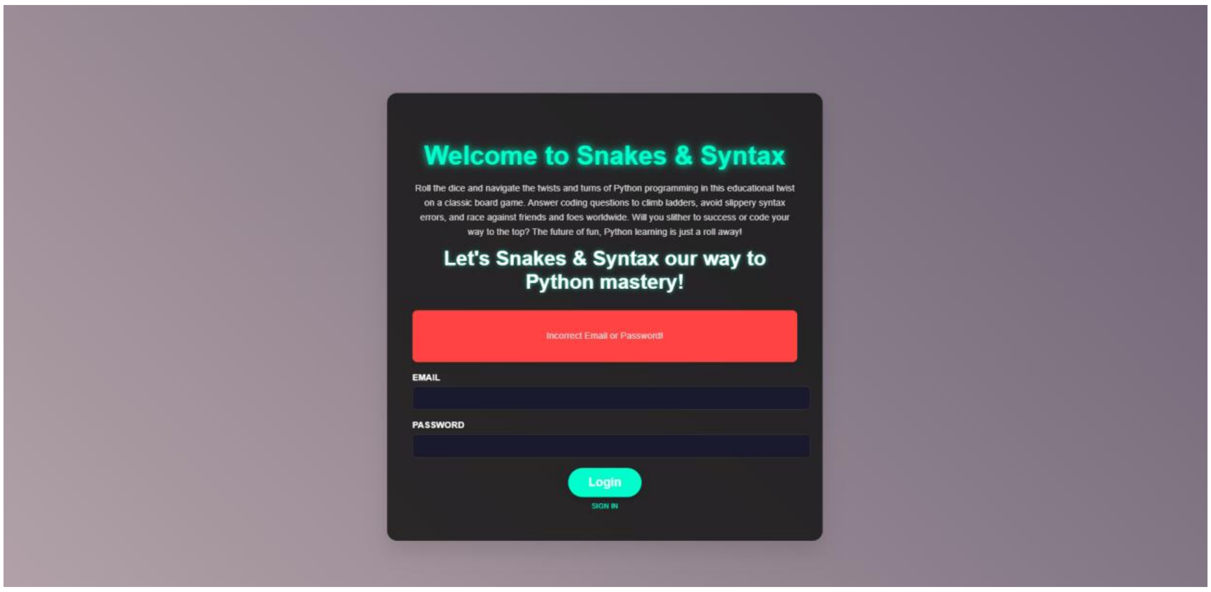
4.2 Login Page



Welcome to Snakes & Syntax

Roll the dice and navigate the twists and turns of Python programming in this educational twist on a classic board game. Answer coding questions to climb ladders, avoid slippery syntax errors, and race against friends and foes worldwide. Will you slither to success or code your way to the top? The future of fun, Python learning is just a roll away!

Let's Snakes & Syntax our way to Python mastery!



Welcome to Snakes & Syntax

Roll the dice and navigate the twists and turns of Python programming in this educational twist on a classic board game. Answer coding questions to climb ladders, avoid slippery syntax errors, and race against friends and foes worldwide. Will you slither to success or code your way to the top? The future of fun, Python learning is just a roll away!

Let's Snakes & Syntax our way to Python mastery!

Incorrect Email or Password!

EMAIL

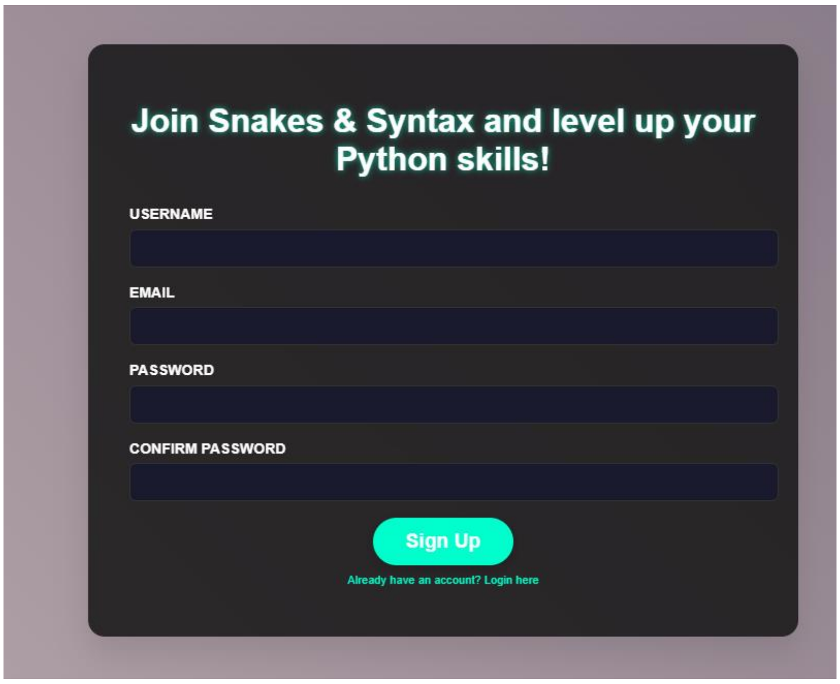
PASSWORD

Login

SIGN IN

This is the login page for the "Snakes & Syntax" game. The page includes fields for the user to enter their email and password. If there is an error, an error message is shown. When the user submits the form, it will proceed to the home page. There is also a link for users to sign up if they don't have an account yet.

4.3 Sign up page



Join Snakes & Syntax and level up your Python skills!

USERNAME

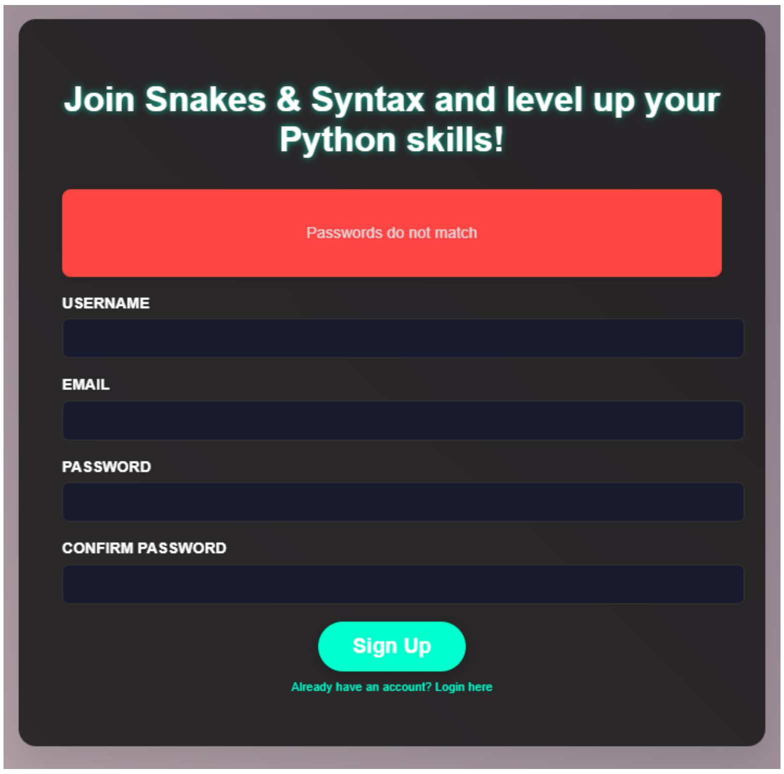
EMAIL

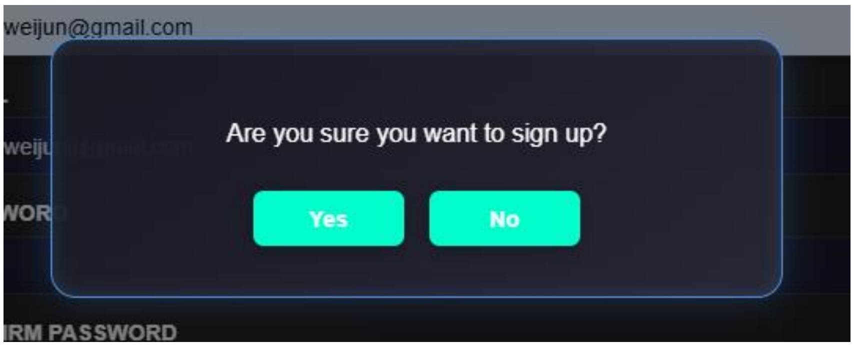
PASSWORD

CONFIRM PASSWORD

Sign Up

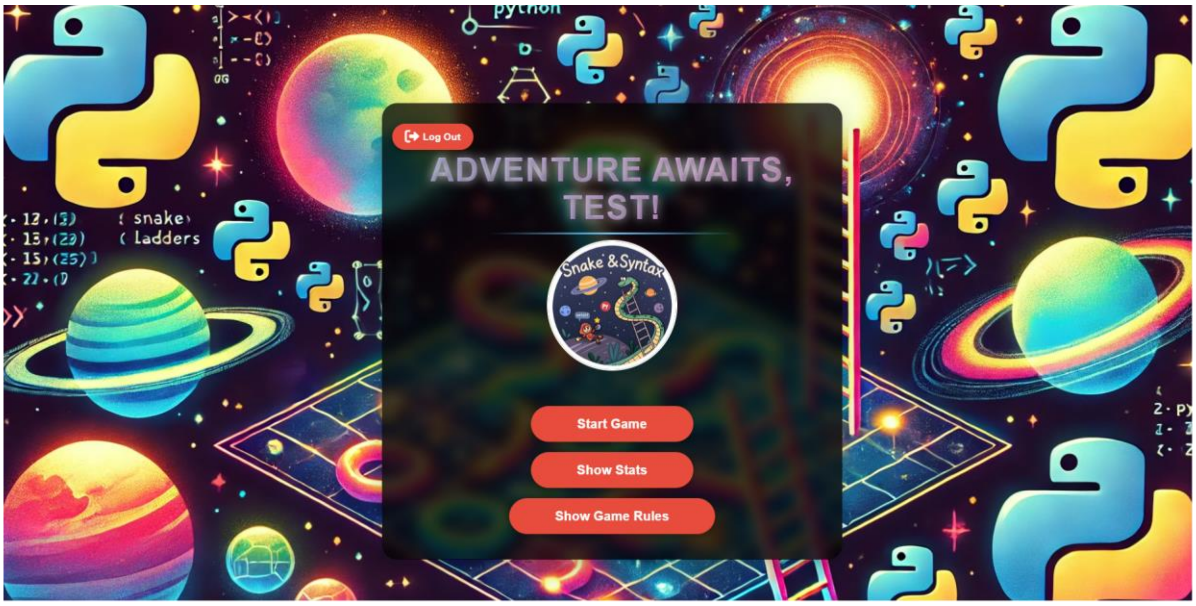
Already have an account? Login here

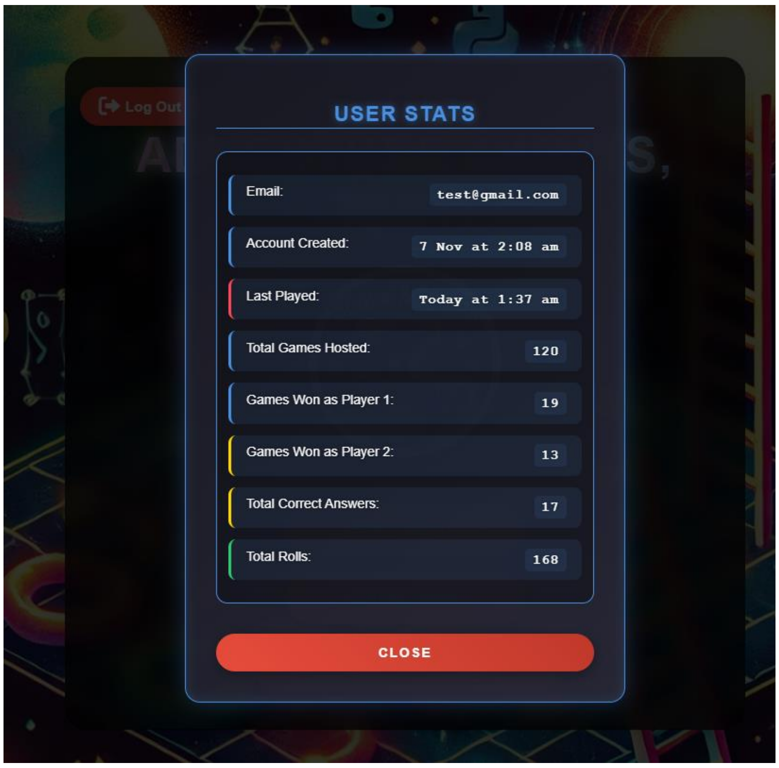

Join Snakes & Syntax and level up your Python skills!
Passwords do not match
USERNAME
EMAIL
PASSWORD
CONFIRM PASSWORD
Sign Up
Already have an account? Login here



This is the sign-up page. It allows new users to create an account. If there is an error, an error message is displayed at the top. Before the form is submitted, a confirmation modal appears asking the user if they are sure they want to sign up.

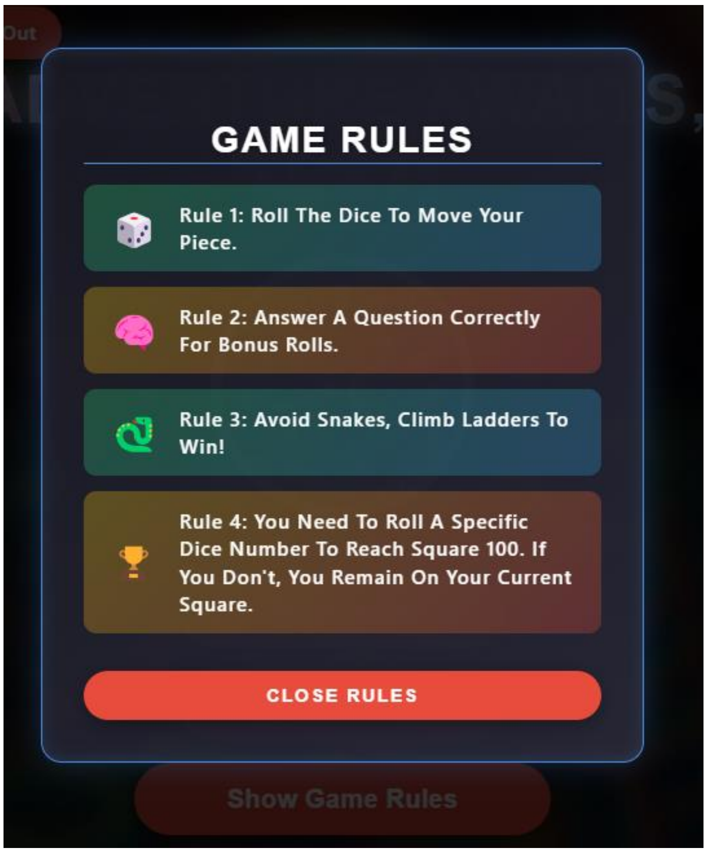
4.4 Home page





USER STATS

Email:	test@gmail.com
Account Created:	7 Nov at 2:08 am
Last Played:	Today at 1:37 am
Total Games Hosted:	120
Games Won as Player 1:	19
Games Won as Player 2:	13
Total Correct Answers:	17
Total Rolls:	168



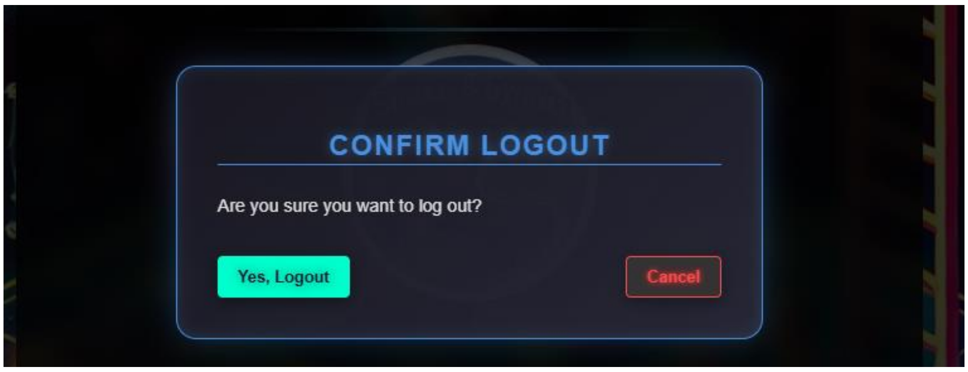
GAME RULES

Rule 1: Roll The Dice To Move Your Piece.

Rule 2: Answer A Question Correctly For Bonus Rolls.

Rule 3: Avoid Snakes, Climb Ladders To Win!

Rule 4: You Need To Roll A Specific Dice Number To Reach Square 100. If You Don't, You Remain On Your Current Square.

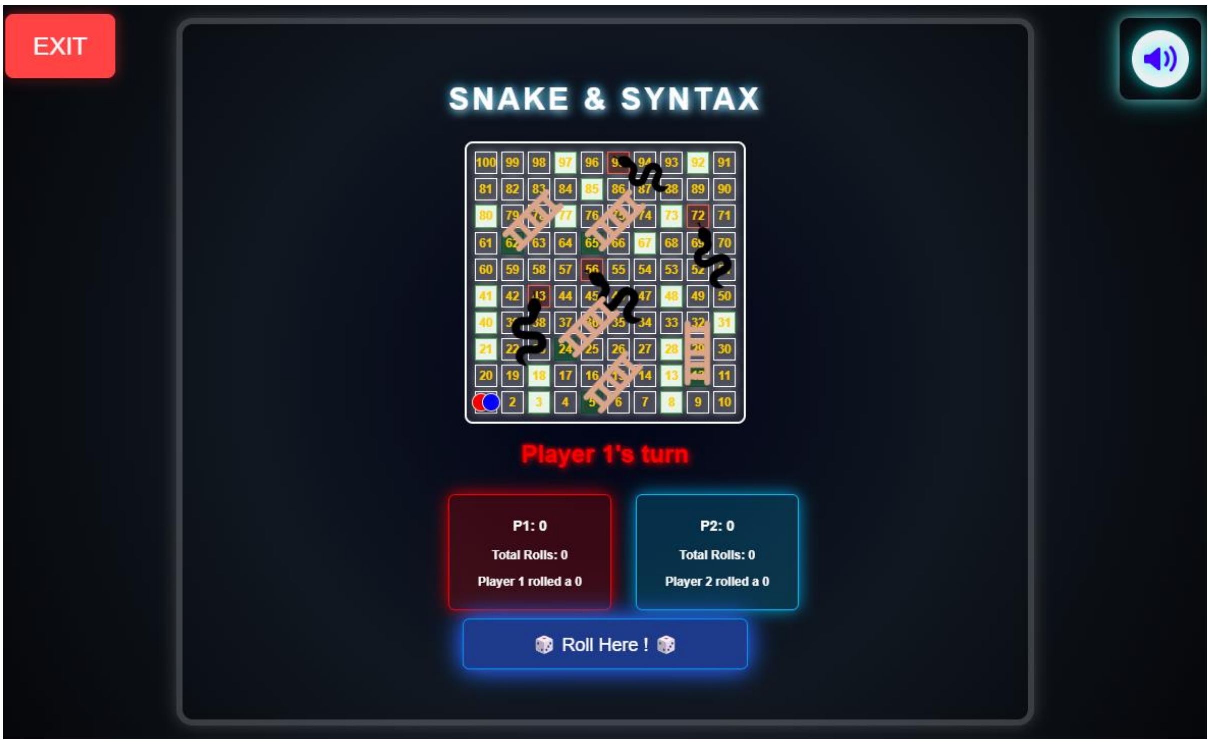


CONFIRM LOGOUT

Are you sure you want to log out?

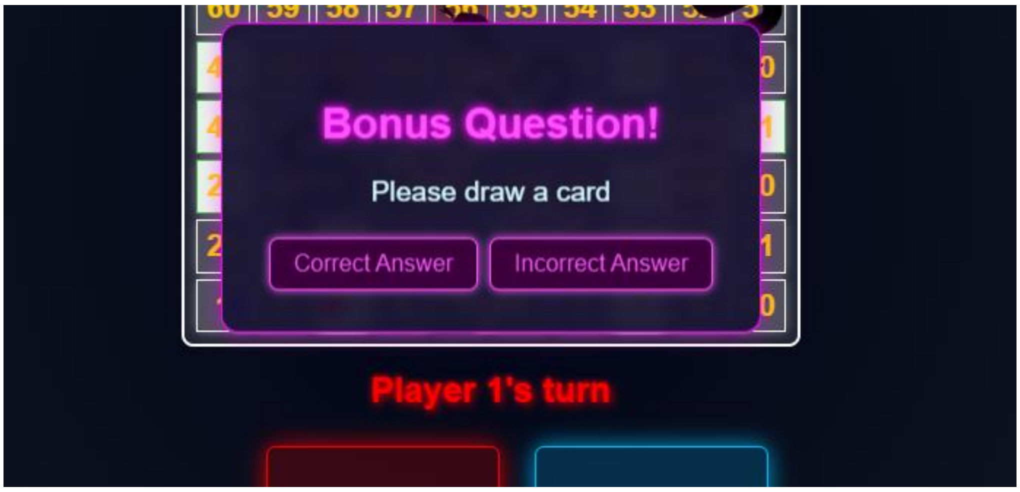
This is the home page for the "Snakes & Syntax" game, where users can access key features after logging in. The page includes a "Start Game" button to begin a new game, along with buttons to view user stats and game rules. There is also a logout button that triggers a confirmation process before logging the user out.

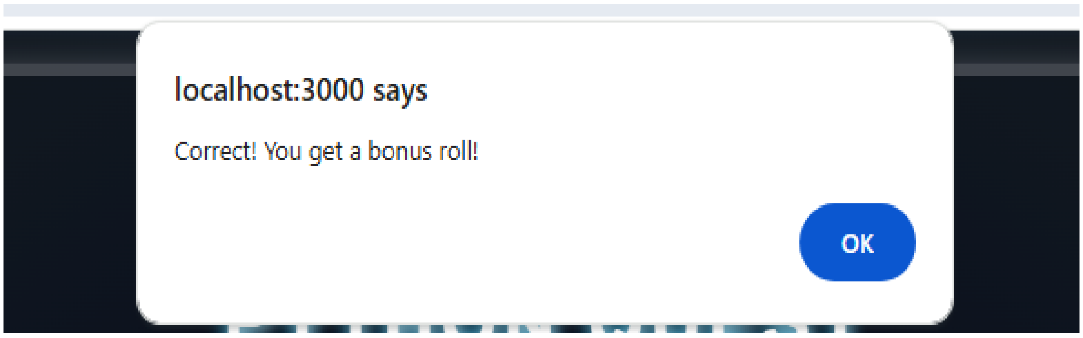
4.5 Gameboard Page



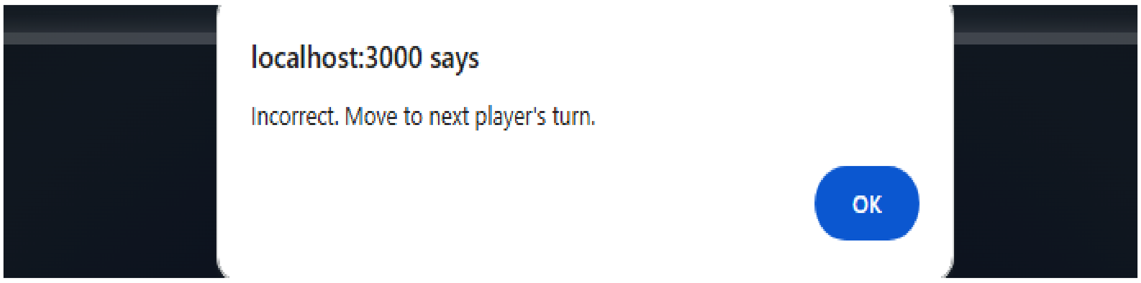
This is a layout for the "Snake & Syntax" game where users can interact with the gameboard, roll dice, view player stats, and exit the game. The game also includes background music to make the experience more enjoyable. The gameboard consists of several squares, where each number represents a position. Some squares contain ladders or snakes. Ladders help you climb up to a higher square. Snakes send you sliding back to a lower square. To move, click the "Roll Here!" button. This will roll, and your token will move based on the result. After each roll, the game will automatically update to show your new position on the board.

4.6 Bonus question modal




localhost:3000 says

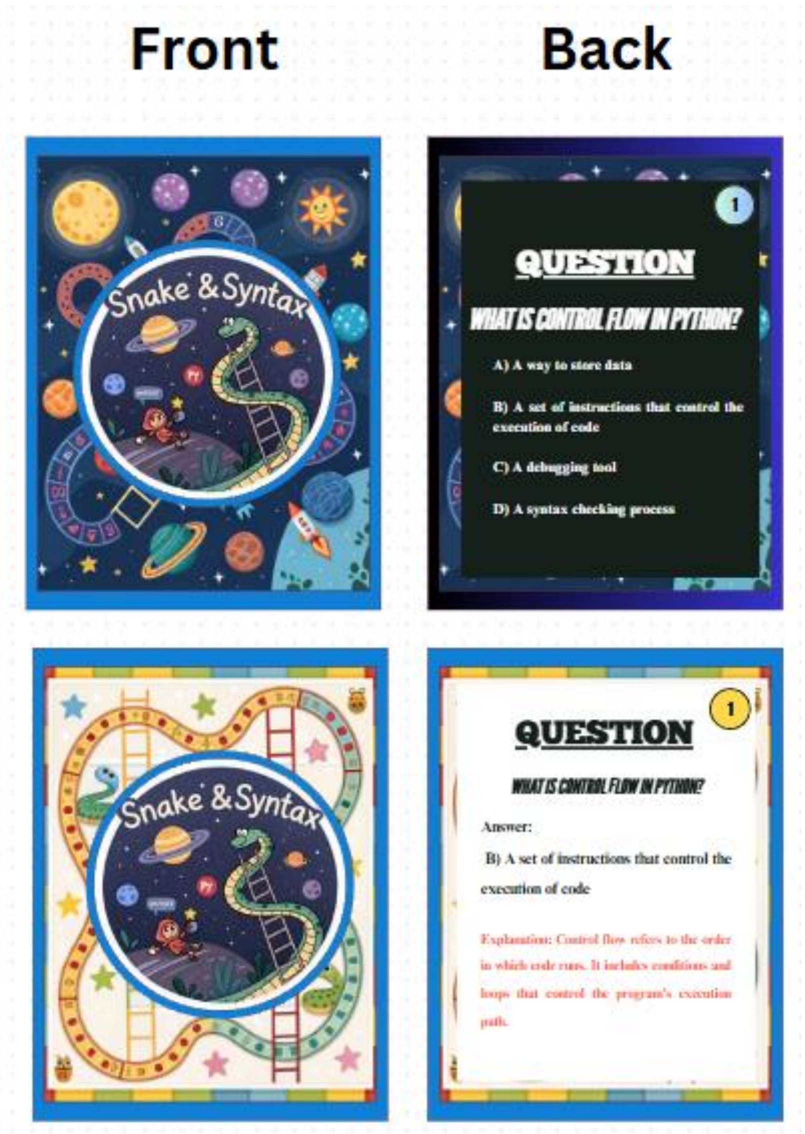
Correct! You get a bonus roll!


localhost:3000 says

Incorrect. Move to next player's turn.

When you will land on a special square that triggers a Bonus Question. A modal will pop up, prompting you to answer a question by clicking the "Correct Answer" or "Incorrect Answer" buttons. Correct answers could reward you with a bonus roll.

4.7 Physical Card



1

QUESTION

WHAT IS CONTROL FLOW IN PYTHON?

A) A way to store data

B) A set of instructions that control the execution of code

C) A debugging tool

D) A syntax checking process

1

QUESTION

WHAT IS CONTROL FLOW IN PYTHON?

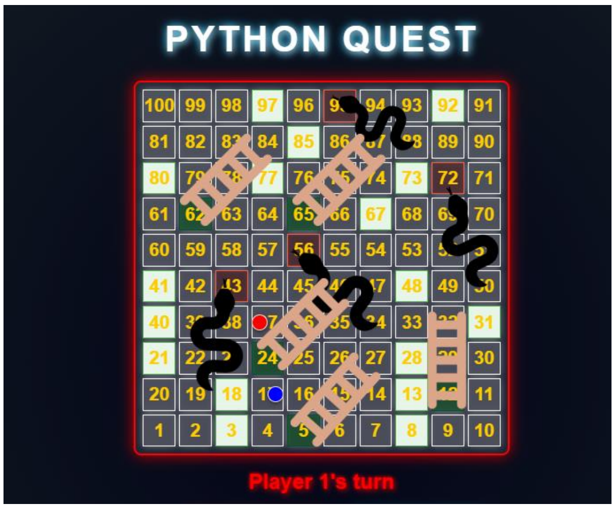
Answer:

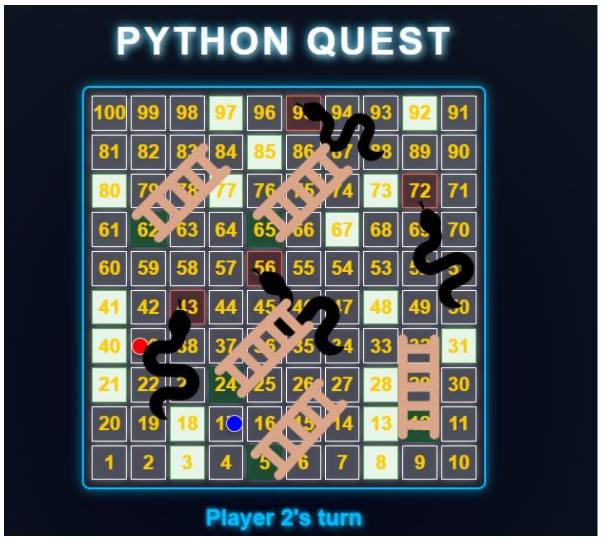
B) A set of instructions that control the execution of code

Explanation: Control flow refers to the order in which code runs. It includes conditions and loops that control the program's execution path.

The game includes two physical card decks: one for questions and another for answers with explanations. When players land on a special box, they must draw a question card and answer.

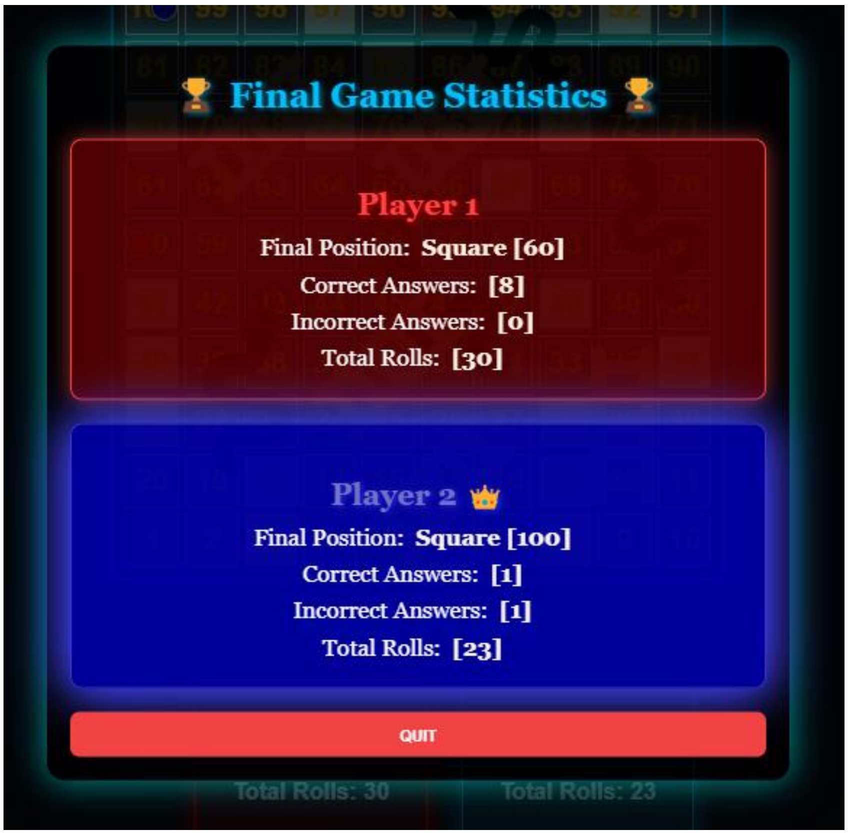
4.8 Player Turn





In the game, the player turn is clearly indicated in the section, where it shows whose turn it is, such as "Player 1's turn." The game switch between Player 1 and Player 2, allowing each player to roll the dice and move their piece based on the dice roll.

4.9 Final Game Statistics



🏆 Final Game Statistics 🏆

Player 1

Final Position: Square [60]

Correct Answers: [8]

Incorrect Answers: [0]

Total Rolls: [30]

Player 2 👑

Final Position: Square [100]

Correct Answers: [1]

Incorrect Answers: [1]

Total Rolls: [23]

When a player reaches cell 100, the game concludes, and a Final Game Statistics modal is displayed to summarize the results. This Stats Modal includes details for both players, such as:

- The winner of the game.
- Each player’s final position on the board.
- The total number of correct and incorrect answers each player provided throughout the game.
- The number of dice rolls for each player, showing how many attempts it took to reach the end.

Lastly, a Quit button allows players to return to the home page.

5.0 Conclusions

The "Snakes & Syntax" game merges classic board game with interactive Python-related questions, creating a fun and educational experience for users. Through this project, I have integrated key features such as a dynamic board display, turn-based logic, score tracking, and engaging prompts for bonus questions. By implementing an intuitive UI and utilizing various JavaScript functionalities, the game offers a smooth, user-friendly experience where players can enhance their Python knowledge while competing in a familiar Snakes and Ladders format.

Potential Future Enhancements:

- **More Question Variety**: Adding more Python questions with different difficulty levels or topics will let players tailor the game to their skill level and interests.
- **Scoreboard and Achievements**: A persistent scoreboard and achievements system would allow players to track their progress and review past performance.
- **Online Multiplayer Mode**: Adding real-time multiplayer options would let players compete with friends remotely, making the game more interactive and fun.

These enhancements can expand the functionality, accessibility, and educational value of "Snakes & Syntax," creating a more engaging experience for all types of players.

5.1 References

W3Schools.com. (n.d.). https://www.w3schools.com/css/css_list.asp

Flaticon. (n.d.). *Free Icons and Stickers - Millions of resources to download*. https://www.flaticon.com/

MDN. (2024, July 25). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/CSS

ChatGPT. (n.d.). https://chatgpt.com/