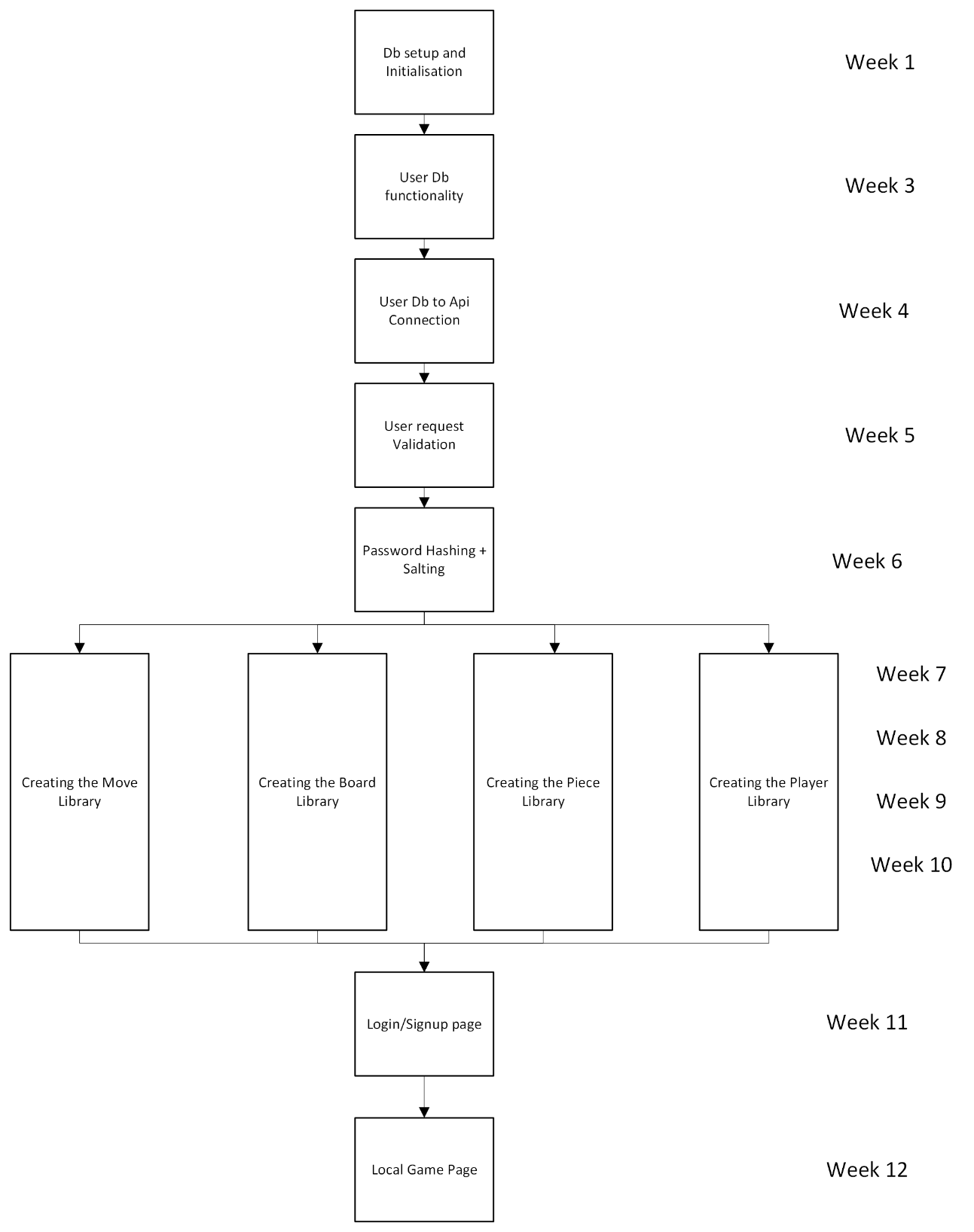
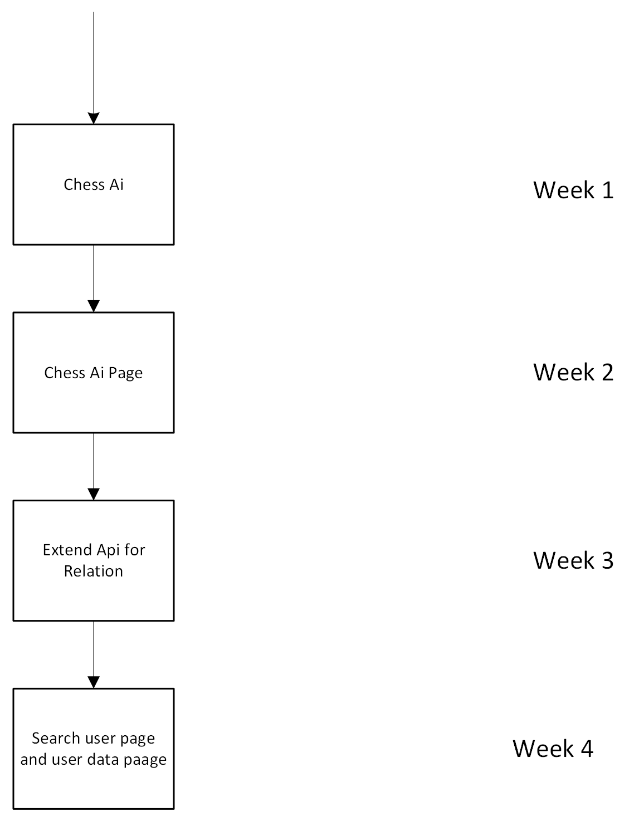
Design

Critical Path Diagram

I believe my project will take 16 weeks to complete considering all of the different features in the app and the time I will spend researching new concepts. This is just a general approximation and in reality, the time for the app to be made may deviate especially for some of the more complex features of the app such as AI. This also does not consider the times where time will be reduced due to good modularity and abstraction allowing code to be reused and not remade.

This is the flow for the components of my app that are in the scope of A-Level below are the parts out of scope. I have separated the two so I can treat the parts outside of scope as an extension if my week plan does not go as expected. Testing will happen at every stage I predicted it will take an extra week.

System Design

Before I can begin with creating my application, I need to formulate a plan of how the program will look and how this will link to each page. This will allow me to quickly create the front end of the website because I will know how I want it to look before I even start.

In the top left of every page there will be the site logo this logo when pressed will bring the user back to the landing page so they can access the different features of the app from there.

When the website is first accessed there will be a main landing page. This page will display chess in the centre of the screen the first time that it is accessed. When the user has logged in using the login page described later this will change to the welcome then the username stored in the web local storage. In the top right of the page there will an animated navigation icon. This will have a cool bubble effect bringing some life to the page. When clicked the nav icon will expand to occupy the screen on this new background the different pages available in the app will be shown. If the user is not logged in the AI GAME option will be greyed out and unavailable.

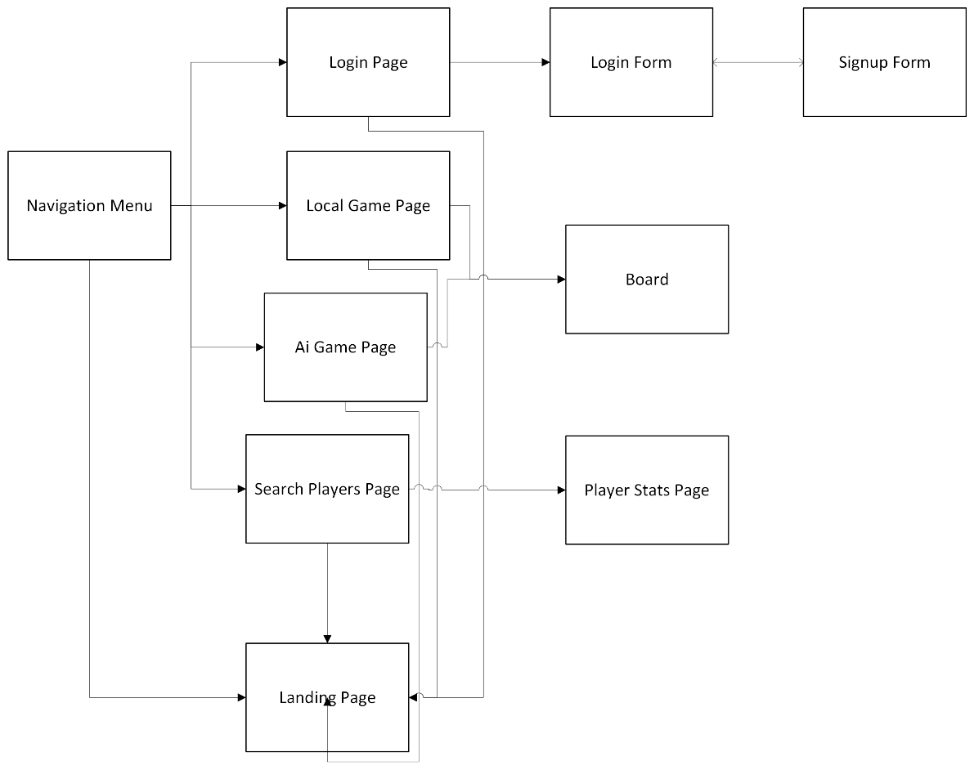
When the Login/signup option is clicked on the navigation menu the user is brought to the login page. On this page there is a signup form and login form. Initially the signup form is displayed when the user leaves a field blank a red message display telling them to complete the form. When the user enters an invalid email, a red message will tell them it is invalid. When the user enters a password that is less than 5 characters a red message is displayed. And finally, if on sign up the username is already taken a message will be displayed telling the user the username is already taken. At the bottom of the signup form there is a login button when pressed an expand animation brings the login form up for the user so the user can login. If an invalid login is entered a red text message tells them, it does not exist. On a successful login or sign in the user will be brought to the landing page with there name now displayed on the screen.

When the local game option is clicked on the navigation menu the user is brought to the local-game page. On this page there is a chess board. When a piece is clicked all of the valid moves for that piece are displayed (invalid moves that would not save a king from check are still shown this is, so you have to think to get out of check). If a square that is not a blue valid square is clicked the selected piece is unselected. When the king is in check it is highlighted red. When the king is in check and the user tries to move to a blue square that is invalid it becomes orange and the piece is then unselected. If there is a checkmate the game ending message fades into the screen and the next click brings the user to the landing page. If there is a stalemate it is the same but with a draw message.

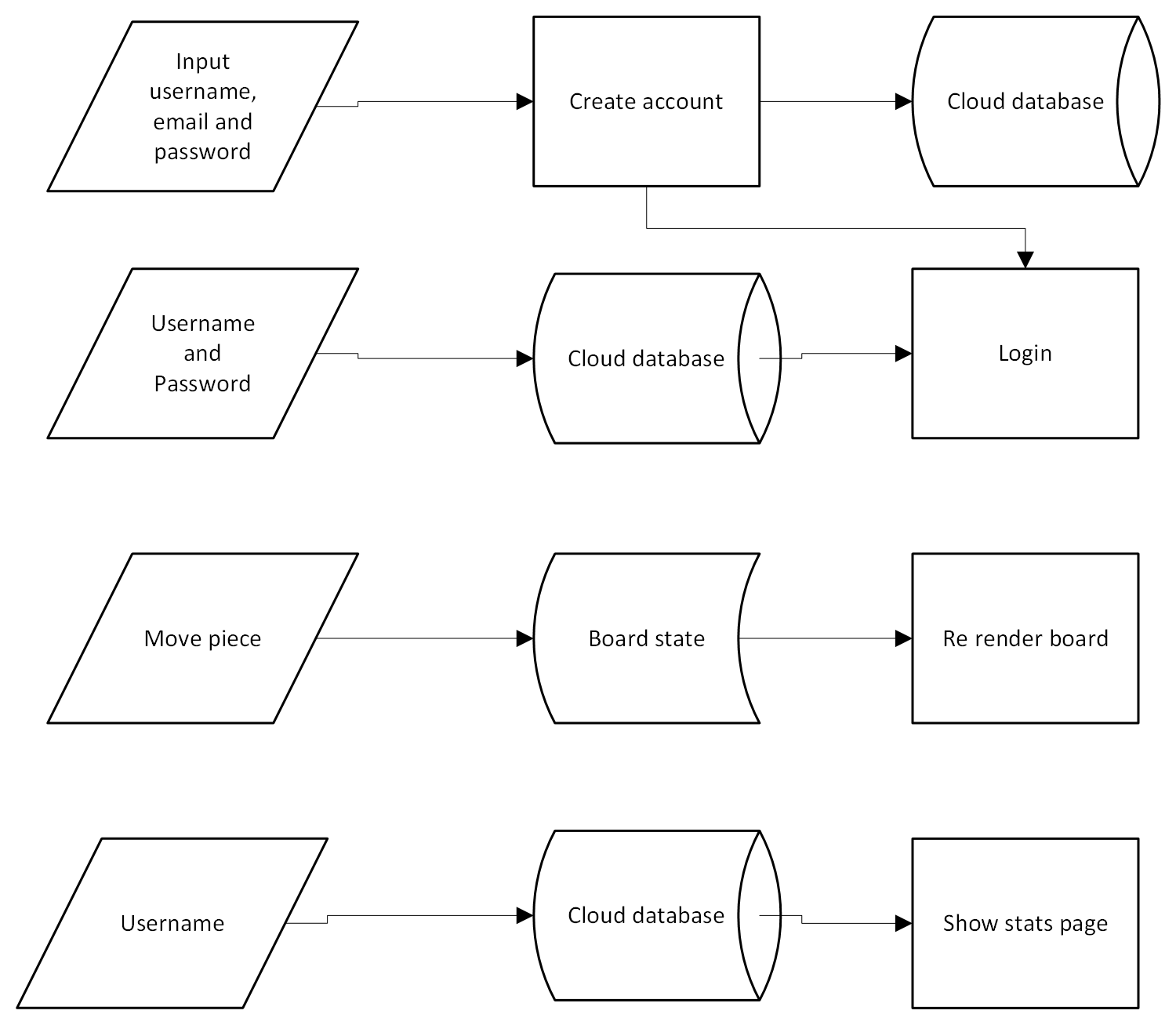
The chess ai page will be exactly the same as the local game page however black pieces will never be moveable and when the user makes a move nothing will be clickable until the ai has moved. The ai will always make a valid move and the board will react accordingly with the same rules as the local game page.

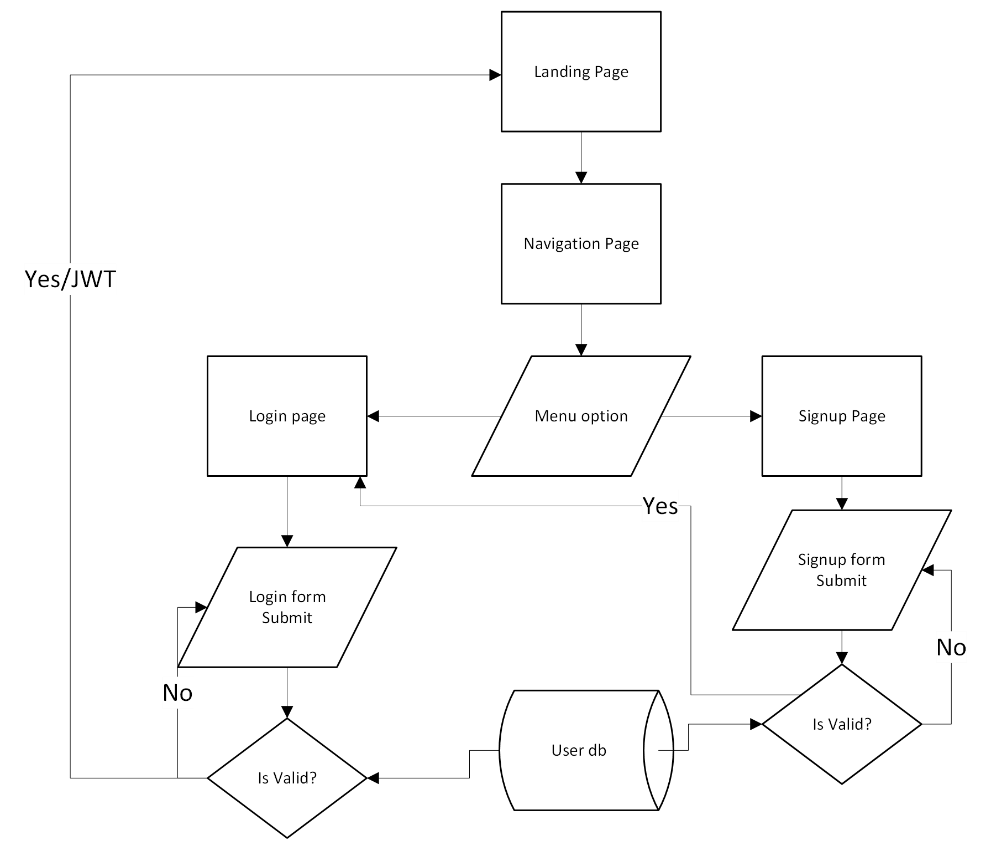
To do- write about search players when it’s done.

Modular System Design



General System Flow Chart





Interface Design and Rationale

User Interface Design (HCI)

These images display what my pages will look like when they are created. Including all of the different states of each page.

Landing page



To keep with the minimalistic design, I have kept it simple. Chess is displayed in the centre of the screen and a navigation icon is available in the top corner.

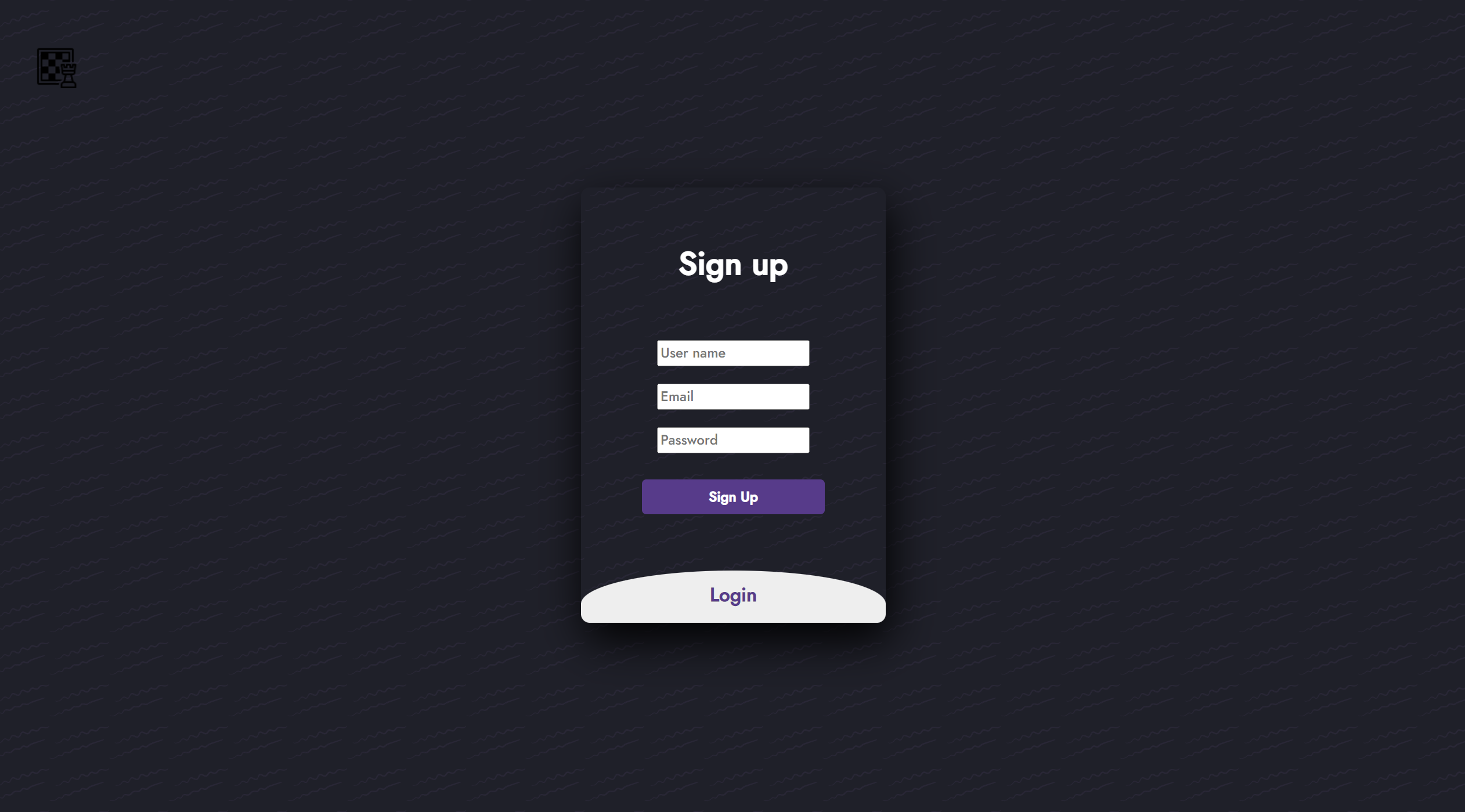


When the user logs in the state of this page changes. The username of the user is displayed along side a welcome message. Otherwise, the page is the same.

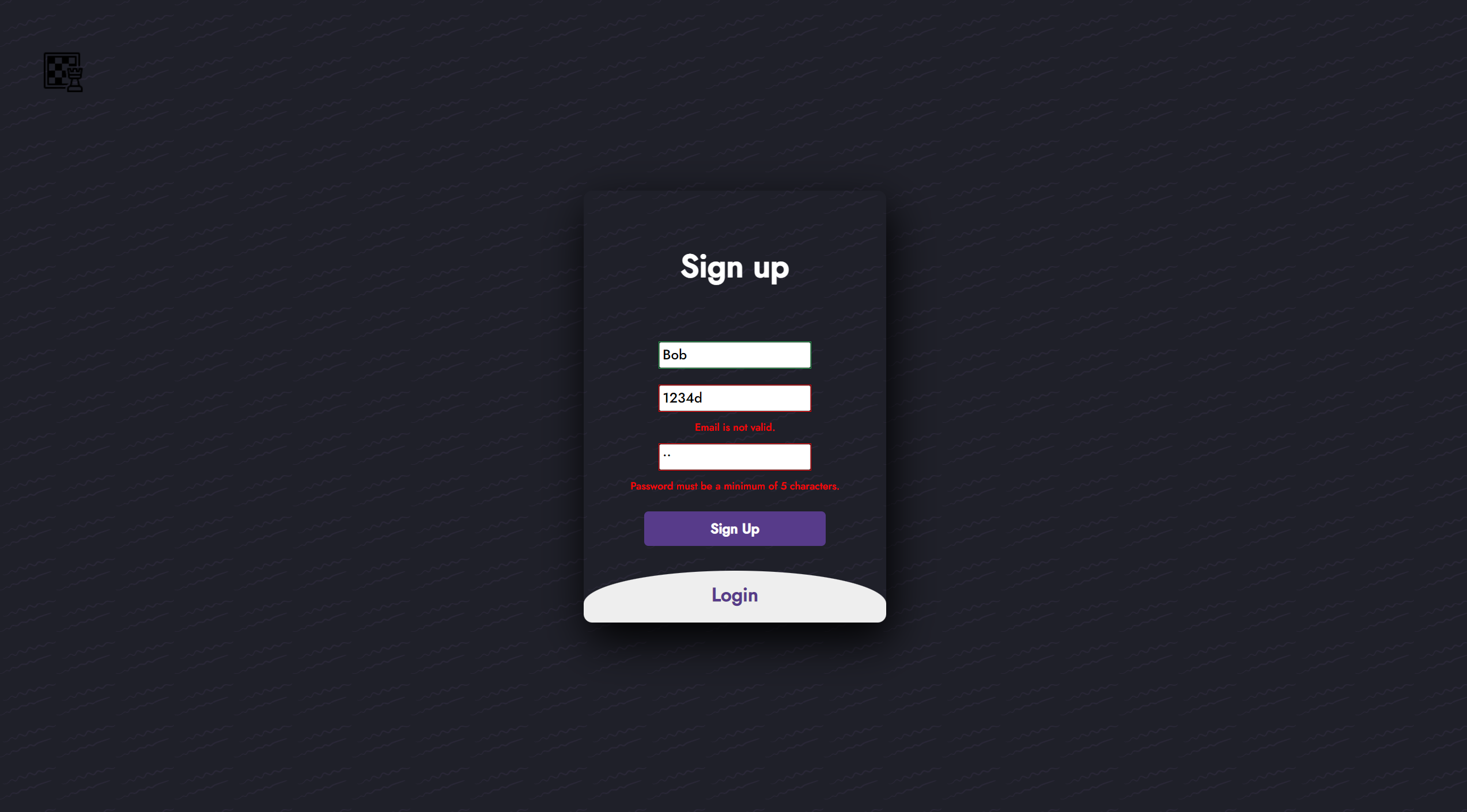


When the navigation icon is clicked the user is shown 4 options. When these options are clicked, they are brought to the respective game. This allows the user to have a simple intuitive way to traverse the site.

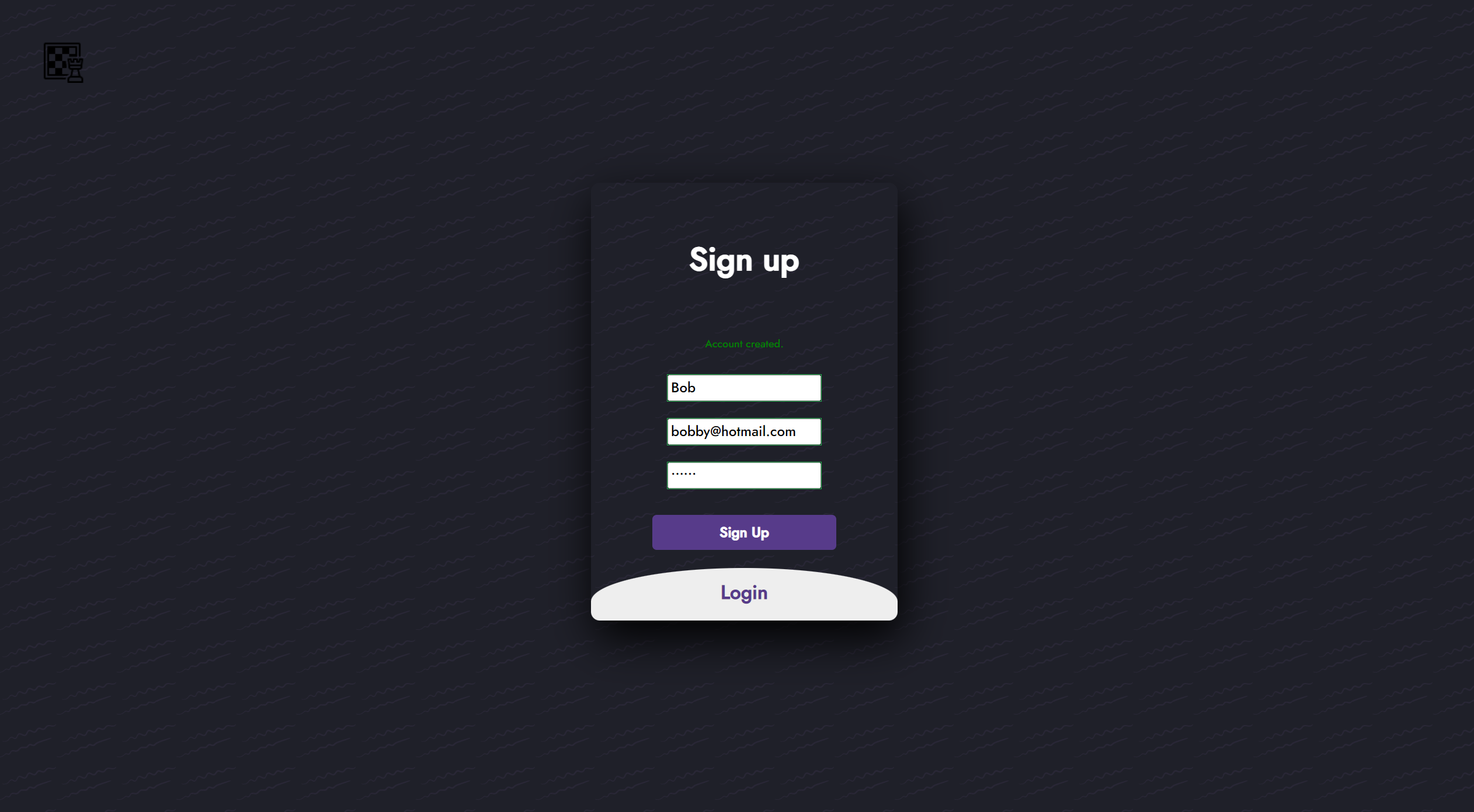
Login page



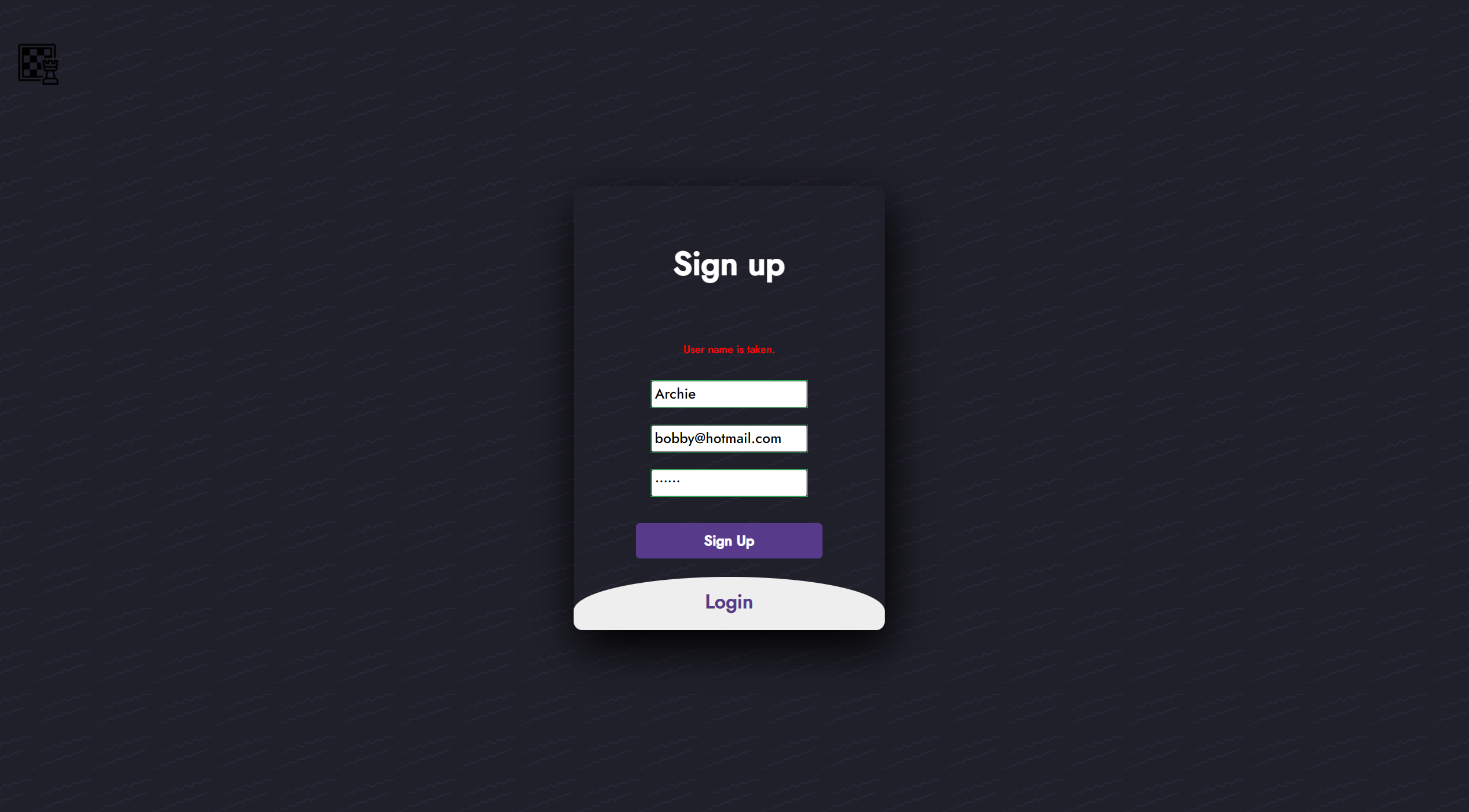
When the login page is first accessed the user is displayed a nice standard sign up form. The landing page button is in the top left and the button to change into a login page is at the bottom.



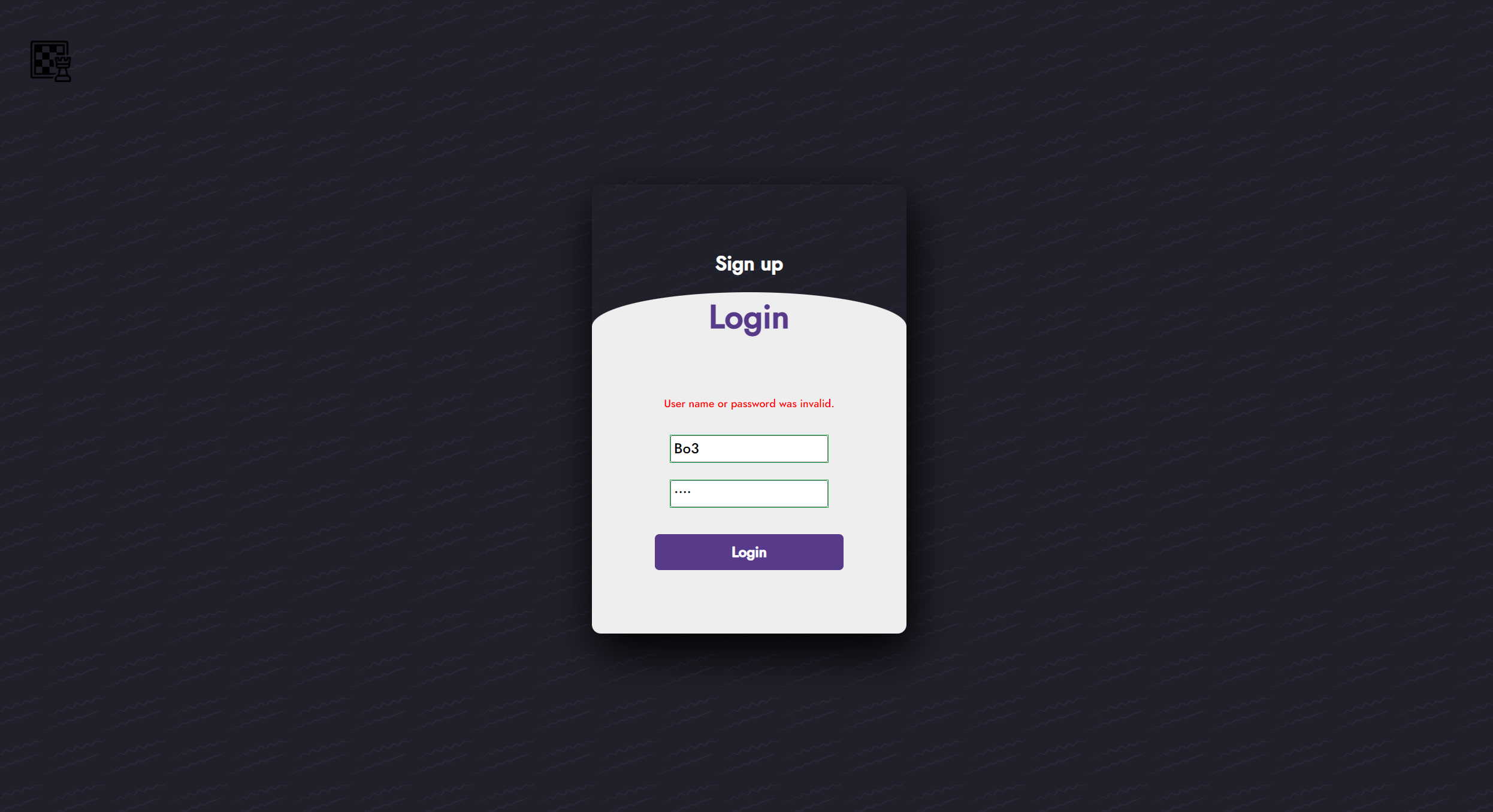
When invalid signup information is entered the user is given an appropriate error message under the field.



When an account is successfully created an account creation message is shown in green. Now the user must login to access their new account.

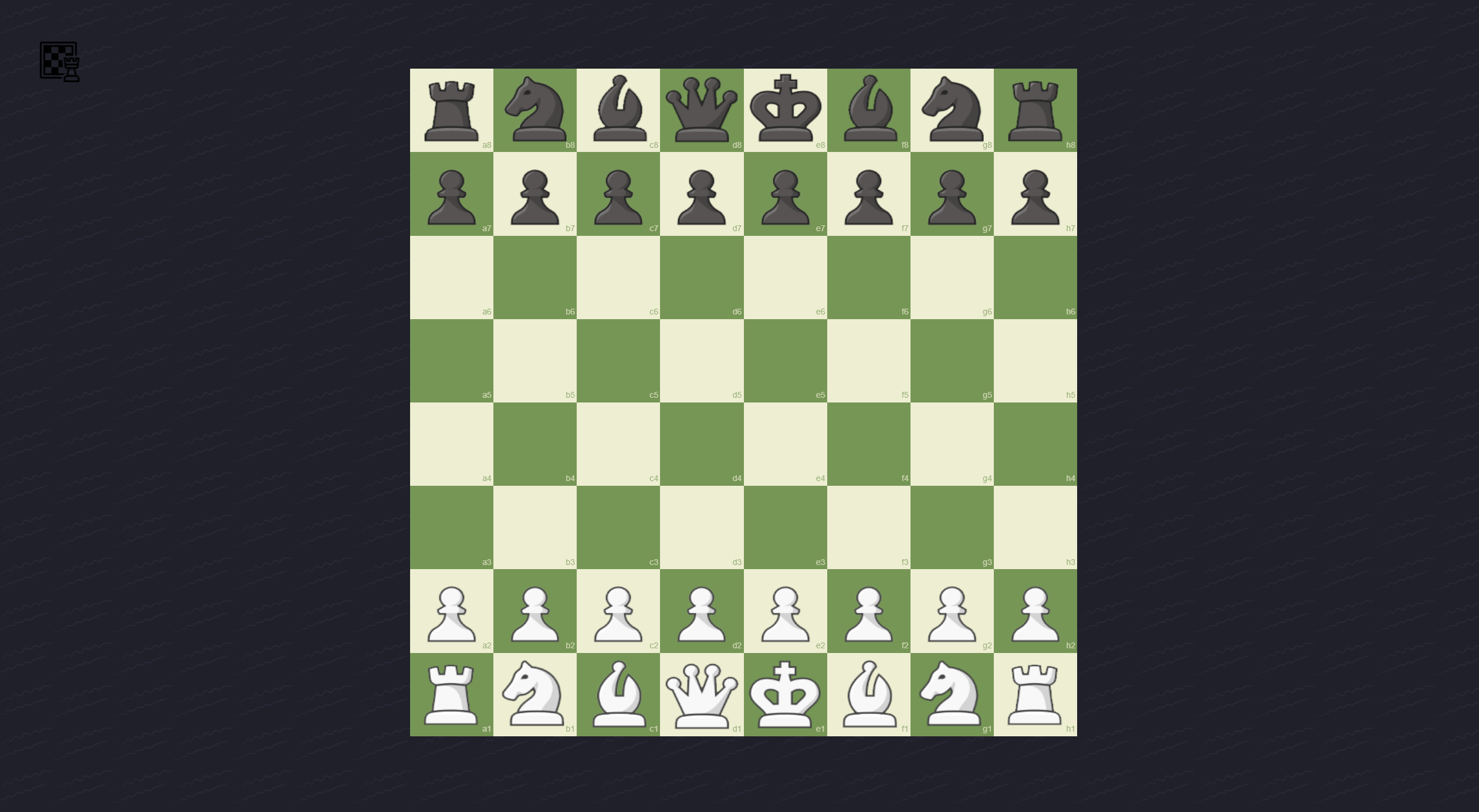


If an account creation is attempted and the username is already taken the user is informed.

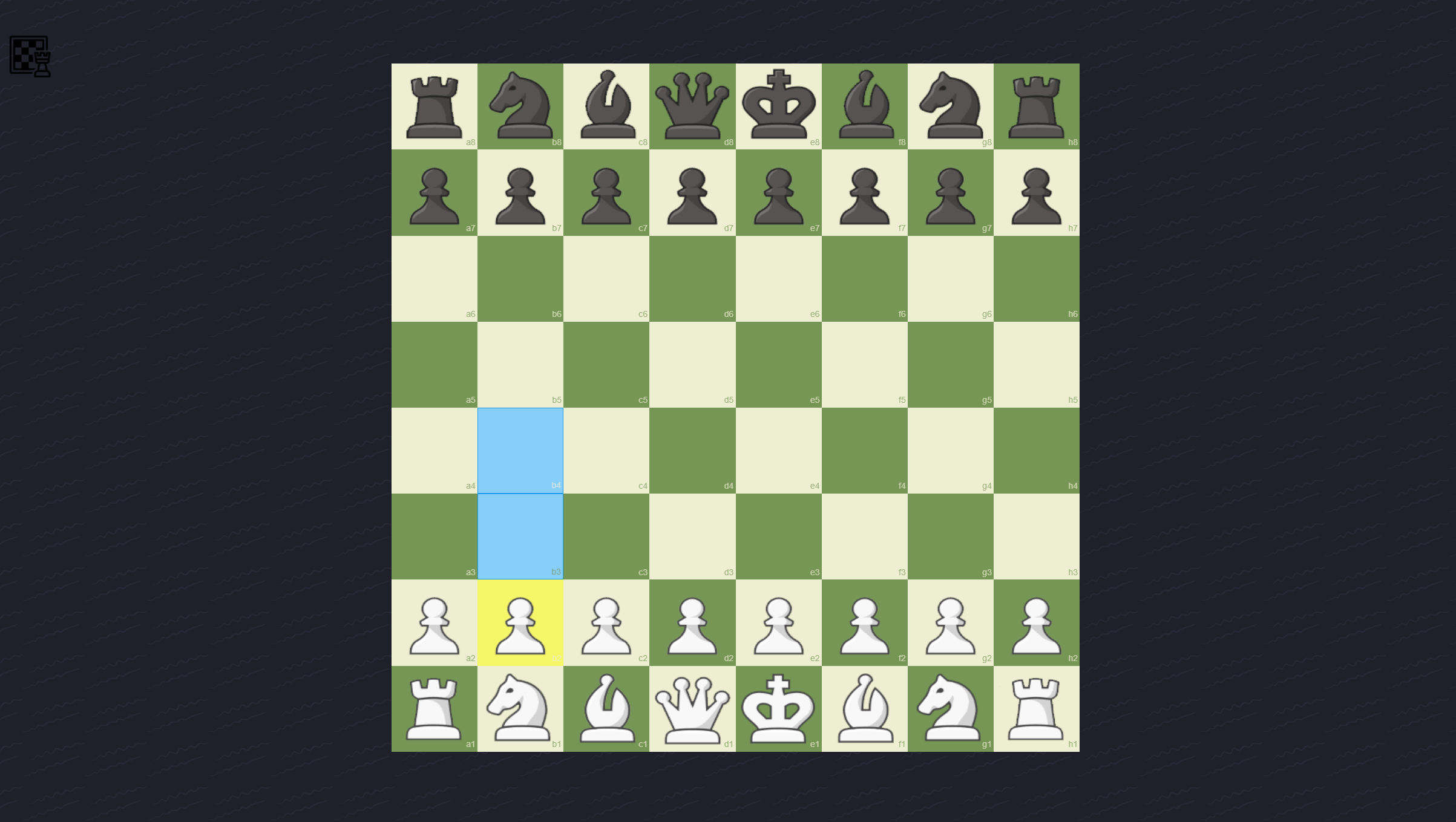


When a user transitions to the login form the signup button is at the top. When a user tries to log in and fails they are informed with an appropriate error message.

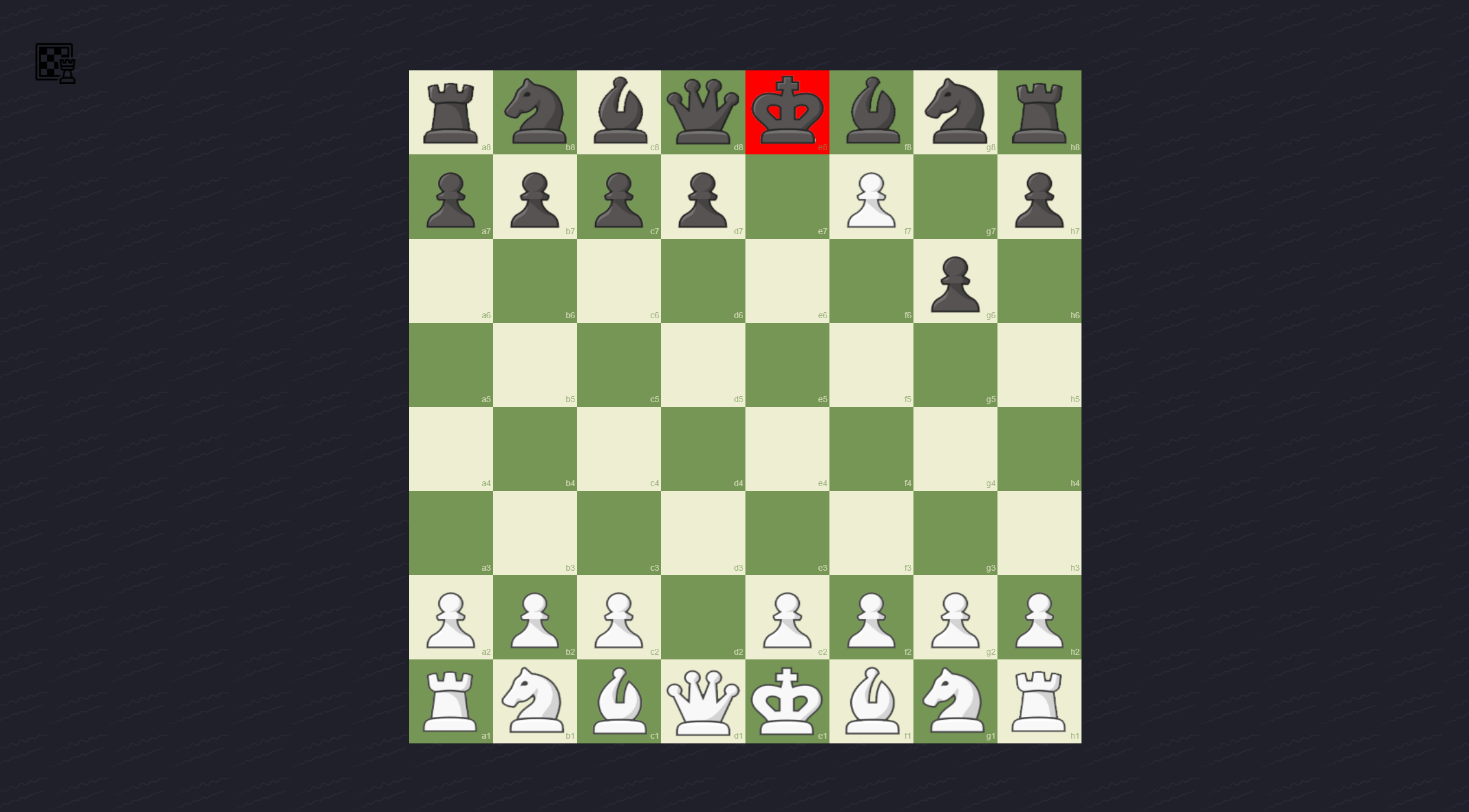
Local game page



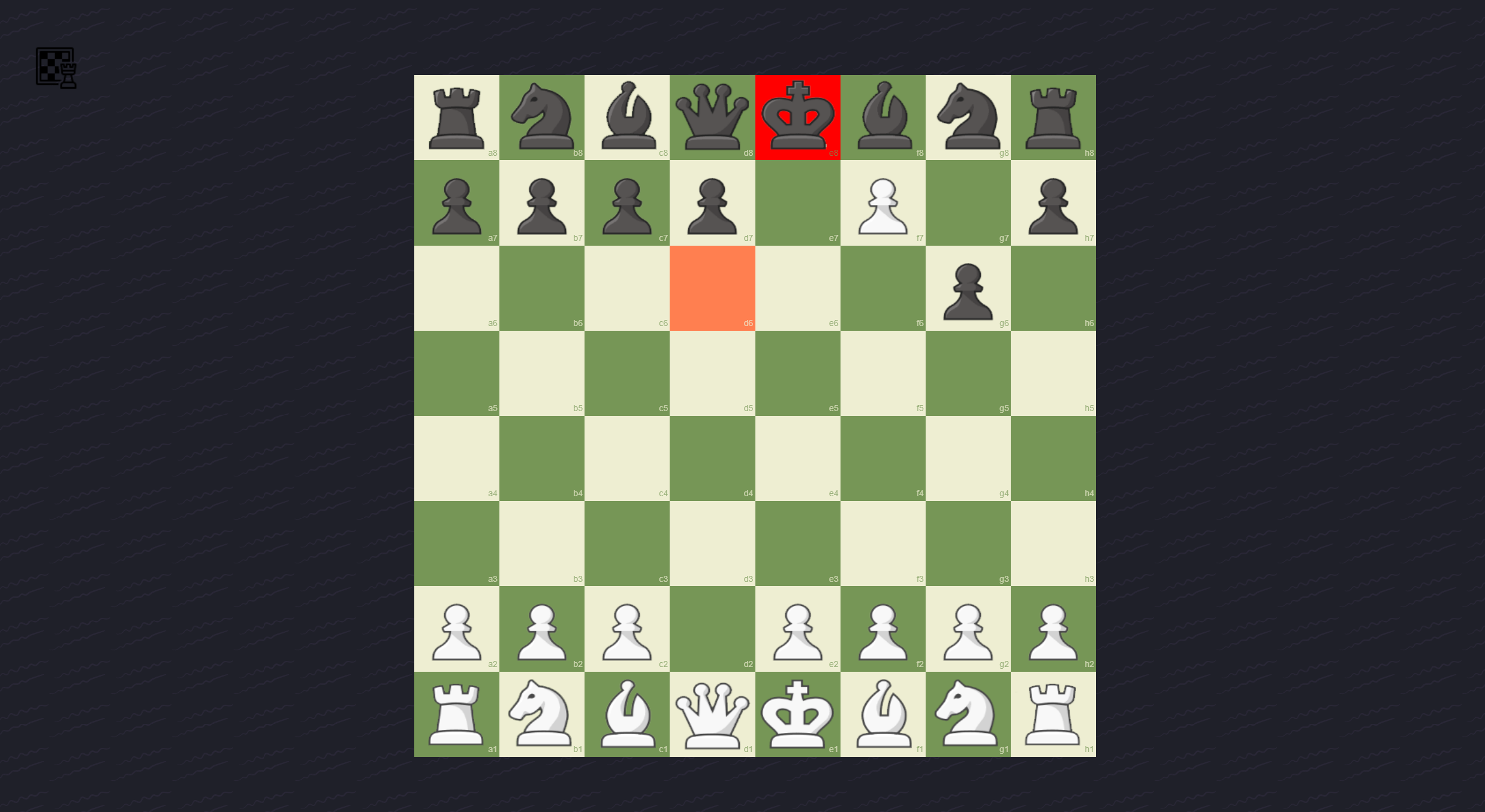
The local game page again like all the other pages is minimal. The user is provided a standard board to play.



When a piece is selected all of the valid moves for that piece on that turn are displayed. The user can move it to any of these squares assuming that it doesn’t cause their own king to be checked.



When a king is in check it is highlighted red. This allows the user to react and not get confused why all of his moves are invalid.



When a move is attempted that will put a friendly king in check the square is highlighted orange and is not played.

Algorithms

Signup

Pseudocode

CLASS UserSignupModel:

string Username with getter and setter

has web attributes:

Required: errorMessage="Please complete field"

MaxLength: 30

string Email with getter and setter

has web attributes:

Required: errorMessage="Please complete field"

EmailAddress: errorMessage="Email is not valid."

string Password with getter and setter

has web attributes:

Required: errorMessage="Please complete field"

MinLength: 5, errorMessage="Password must be a minimum of 5 characters"

PAGE CODE SignupForm:

// service that abstracts out http requests into english

UserService \_userService from dependency injection

UserSignupModel \_userSignupModel = new

bool \_signupFailed = false

bool \_signupSuccess = false

PROCEDURE OnValidSignUp():

HttpResponseMessage response = \_userService

.Create(new UserModel from UserSignupModel)

IF not reponse.IsSuccess:

\_signupFailed = true;

\_signupSuccess = false;

ELSE:

\_signupFailed = false;

\_signupSuccess = true;

Coded Algorithm

using System.ComponentModel.DataAnnotations;

using ChessWebApp.UI.Models;

using Microsoft.AspNetCore.Components;

public sealed class UserSignupModel

{

    [Required(ErrorMessage = "Please complete field.")]

    [MaxLength(30)]

    public string Username { get; set; } = default!;

    [Required(ErrorMessage = "Please complete field.")]

    [EmailAddress(ErrorMessage = "Email is not valid.")]

    public string Email { get; set; } = default!;

    [Required(ErrorMessage = "Please complete field.")]

    [MinLength(5, ErrorMessage = "Password must be a minimum of 5 characters.")]

    public string Password { get; set; } = default!;

}

public sealed partial class SignupForm : ComponentBase

{

    [Inject]

    private IUserService \_userService;

    private readonly UserSignupModel \_userSignupModel = new();

    private bool \_signupFailed;

    private bool \_signupSuccess;

    private async void OnValidSignupSubmit()

    {

        HttpResponseMessage response = \_userService.Create(new UserModel

        {

            Username = \_userSignupModel.Username,

            Email = \_userSignupModel.Email,

            Password = \_userSignupModel.Password

        });

        if (!response.IsSuccessStatusCode)

        {

            \_signupFailed = true;

            \_signupSuccess = false;

            return;

        }

        \_signupFailed = false;

        \_signupSuccess = true;

    }

}

This algorithm relies on built in validation from blazor. Using this allows us to very easily show correction messages to the user using the EditForm component. As each individual letter is set by the form the models attribute will be called and inform the user real time if they have made a mistake. When submitting a valid form the http abstraction I will have made will attempt to create a new user and if this fails (primarily due to the user already existing) the failed attribute will be set true and the user will see a message telling him an account already has that username. Otherwise, signup success will be true and the user will be given a success message.

Login

Pseudocode

CLASS UserLoginModel:

    string Username

        has web attributes:

            Required: errorMessage="Please complete field."

    string Password

        has web attributes:

            Required: errorMessage="Please complete field."

PAGE CODE LoginForm:

    UserService injected from dependency injection

    NavManager injected from dependency injection

    LocalStorage injected from dependency injection

    UserLoginModel \_userLoginModel = new //for use on page from

    bool \_loginFailed = false

    PROCEDURE OnValidLoginSubmit():

        HttpResponseMessage response = UserService.Login(

            UserModel from UserLoginModel

        )

        IF not response.IsSuccess:

            \_loginFailed = true

        ELSE:

            string jwt = response.Content

            LocalStorage.Add(jwt)

            LocalStorage.Add(\_userLoginModel.Username)

            NavManager.NavTo("/")

Coded Algorithm

using ChessWebApp.UI.Models;

using Microsoft.AspNetCore.Components;

using Newtonsoft.Json;

namespace ChessWebApp.UI.Components;

public sealed partial class LoginForm : ComponentBase

{

    [Inject]

    private IUserService \_userService;

    [Inject]

    private ILocalStorageService \_browserStorage

    [Inject]

    private NavigationManager \_navManager;

    private readonly UserLoginModel \_userLoginModel = new();

    private bool \_loginFailed;

    private async Task OnValidLoginSubmitAsync()

    {

        HttpResponseMessage response = await \_userService.LoginAsync(new UserModel

        {

            Username = \_userLoginModel.Username,

            Password = \_userLoginModel.Password

        });

        if (!response.IsSuccessStatusCode)

        {

            \_loginFailed = true;

            return;

        }

        string stringJwt = await response.Content.ReadAsStringAsync();

        await \_browserStorage.SetItemAsync("token", stringJwt);

        await \_browserStorage.SetItemAsync("username", \_userLoginModel.Username);

        \_navManager.NavigateTo("/");

    }

}

Again in this algorithm we are using blazor’s validation. We are also using a local storage service this service allows us to keep the json web token for later when the user need to use requests that require authentication. This algorithm is very similar to the signup algorithm and works the same way however when this one is finished the user is navigated back to the landing page.

Minimax Ai Algorithm:

CLASS MiniMax:

    BoardEvaluator \_boardEvaluator;

    CONSTUCTOR(BoardEvaluator boardEvaluator):

        \_boardEvaluator = boardEvaluator

    Move Execute(Board board, int depth):

        Move bestMove;

        int highestSeenValue = int.Min

        int lowestSeenValue = int.Max

        for move in board.CurrentPlayer.LegalMoves:

            MoveTransition transition = board.CurrentPlayer.MakeMove(move)

            if transition.MoveStatus not equal Done:

                continue

            if board.CurrentPlayer.Alliance is White:

                currentValue = Min(transition.Board, depth - 1)

            else:

                currentValue = Max(transition.Board, depth - 1)

            if White and currentValue >= highestSeenValue:

                highestSeenValue = currentValue

                bestMove = move

            else if white and currentValue <= lowestSeenValue:

                lowestSeenValue = currentValue

                bestMove = move

        return bestMove

    int Min(Board board, int depth):

        if depth = 0 or Game is over:

            return \_boardEvaluator.Evaluate(board, depth)

        int lowestSeenValue = int.Max

        for move in board.CurrentPlayer.LegalMoves:

            MoveTransition transition = board.CurrentPlayer.MakeMove(move)

            if transition.MoveStatus is Done:

                continue

            int currentValue = Max(transition.Board, depth-1)

            if currentValue <= lowestSeenValue:

                lowestSeenValue = currentValue

        return lowestSeenValue

    int Max(Board board, int depth):

        if depth = 0 or Game is over:

            return \_boardEvaluator.Evaluate(board, depth)

        int highestSeenValue = int.Min

        for move in board.CurrentPlayer.LegalMoves:

            MoveTransition transition = board.CurrentPlayer.MakeMove(move)

            if transition.MoveStatus is Done:

                continue

            int currentValue = Min(transition.Board, depth-1)

            if currentValue <= highestSeenValue:

                highestSeenValue = currentValue

        return lowestSeenValue

Coded Algorithm

using System.Diagnostics;

using ChessWebApp.ChessEngine.BoardLib;

using ChessWebApp.ChessEngine.MoveLib;

namespace ChessWebApp.ChessEngine.AiLib;

public class MiniMax

{

    private readonly IBoardEvaluator \_boardEvaluator;

    public MiniMax(IBoardEvaluator boardEvaluator)

    {

        \_boardEvaluator = boardEvaluator;

    }

    public Move Execute(Board board, int depth)

    {

        Move bestMove = new EmptyMove();

        int highestSeenValue = int.MinValue;

        int lowestSeenValue = int.MaxValue;

        foreach (Move move in board.CurrentPlayer.LegalMoves)

        {

            MoveTransition transition = board.CurrentPlayer.MakeMove(move);

            if (transition.MoveStatus != MoveStatus.Done)

            {

                continue;

            }

            int currentValue = board.CurrentPlayer.Alliance == Alliance.White

                ? Min(transition.Board, depth - 1)

                : Max(transition.Board, depth - 1);

            switch (board.CurrentPlayer.Alliance)

            {

                case Alliance.White when currentValue >= highestSeenValue:

                    highestSeenValue = currentValue;

                    bestMove = move;

                    break;

                case Alliance.Black when currentValue <= lowestSeenValue:

                    lowestSeenValue = currentValue;

                    bestMove = move;

                    break;

            }

        }

        return bestMove;

    }

    public int Min(Board board, int depth)

    {

        if (depth == 0 || IsGameOver())

        {

            return \_boardEvaluator.Evaluate(board, depth);

        }

        int lowestSeenValue = int.MaxValue;

        foreach (Move move in board.CurrentPlayer.LegalMoves)

        {

            MoveTransition transition = board.CurrentPlayer.MakeMove(move);

            if (transition.MoveStatus != MoveStatus.Done)

            {

                continue;

            }

            int currentValue = Max(transition.Board, depth - 1);

            if (currentValue <= lowestSeenValue)

            {

                lowestSeenValue = currentValue;

            }

        }

        return lowestSeenValue;

    }

    public int Max(Board board, int depth)

    {

        if (depth == 0 || IsGameOver())

        {

            return \_boardEvaluator.Evaluate(board, depth);

        }

        int highestSeenValue = int.MinValue;

        foreach (Move move in board.CurrentPlayer.LegalMoves)

        {

            MoveTransition transition = board.CurrentPlayer.MakeMove(move);

            if (transition.MoveStatus != MoveStatus.Done)

            {

                continue;

            }

            int currentValue = Min(transition.Board, depth - 1);

            if (currentValue >= highestSeenValue)

            {

                highestSeenValue = currentValue;

            }

        }

        return highestSeenValue;

    }

}



This algorithm is a co-recursive 3 function algorithm that has its roots in game theory and graph theory. Above is the logical path the algorithm takes. In our algorithm we assume the best move for the white player is the smallest number and the best number for the black player is the largest number. This algorithm generates a graph like the image above however in the example the players only have 2 possible moves per turn in chess this would be greater. The reason we use a graph depth and do not just use ever possible move combination is because it is estimated there are 10111 possible moves in chess for reference there is believed to be 1082 atoms in the universe if we used every combination to find the greatest move the program would run for a ridiculous amount of time.

The algorithm evaluates every leaf node using the IBoardEvaluator this interface holds a evaluate function that calculates the worth of a move. I have made it an interface so we can easily implement multiple different evaluation tactics if we wanted. The moves where the black player wins are assigned infinity (int.Max). The moves where white wins are assigned negative infinity (int.Min). In the example at level 3 the smallest node in level 4 will be chosen. In level 2 the max from the level 3 nodes is chosen. This repeats periodically until the root node is reached. This will be the best move returned.

**Board Evaluator Algorithm:**

const CHECKMATEBONUS = 10000

const CHECKBONUS = 50;

const DEPTHBONUS = 100;

int Evaluate(Board board, int depth):

    return ScorePlayer(board, board.WhitePlayer(), depth)

        - Score(board, board.BlackPlayer(), depth)

int ScorePlayer(Board board, Player player, int depth):

    return PieceValue(player) +

        Mobility(player) +

        Check(player, depth) +

        Checkmate(player, depth)

int Checkmate(Player player, int depth):

    if player.Opponent().IsInCheckmate():

        return CHECKMATEBONUS \* Depth(depth)

    else return 0

int Depth(int depth):

    if depth = 0:

        return 1

    else:

        DEPTHBONUS \* depth

int Mobility(Player player):

    return player.LegalMoves().Count()

int PieceValue(Player player):

    int pieceValueScore = 0

    foreach piece in player.ActivePieces():

        pieceValueScore += piece.PieceValue

    return pieceValueScore

**Coded Version**

using ChessWebApp.ChessEngine.BoardLib;

using ChessWebApp.ChessEngine.PieceLib;

using ChessWebApp.ChessEngine.PlayerLib;

namespace ChessWebApp.ChessEngine.AiLib;

public class StandardBoardEvaluator : IBoardEvaluator

{

    private const int CheckmateBonus = 10000;

    private const int CheckBonus = 50;

    private const int DepthBonus = 100;

    public int Evaluate(Board board, int depth)

    {

        return ScorePlayer(board, board.WhitePlayer, depth) -

               ScorePlayer(board, board.BlackPlayer, depth);

    }

    private int ScorePlayer(Board board, Player player, int depth)

    {

        return PieceValue(player) +

               Mobility(player) +

               Check(player, depth) +

               Checkmate(player, depth);

    }

    private int Checkmate(Player player, int depth)

    {

        return player.Opponent.IsInCheckmate ? CheckmateBonus \* Depth(depth) : 0;

    }

    private static int Depth(int depth)

    {

        return depth == 0 ? 1 : DepthBonus \* depth;

    }

    private int Check(Player player, int depth)

    {

        return player.Opponent.IsInCheck ? CheckBonus : 0;

    }

    private int Mobility(Player player)

    {

        return player.LegalMoves.Count();

    }

    private int PieceValue(Player player)

    {

        int pieceValueScore = 0;

        foreach (Piece piece in player.ActivePieces)

        {

            pieceValueScore += piece.PieceValue;

        }

        return pieceValueScore;

    }

}

This algorithm is to be used by the minimax ai algorithm or any other implementation of an ai algorithm that needs an evaluator. It takes in a current boards state and how many moves ahead the boards position is and acts accordingly. When evaluating the greater the score the better it is for the white player and the lower the score the better it is for the black player. If the board state leaves an opponent in checkmate a huge point score will be given this is because this is the greatest scenario possible, so it is automatically the best move. Depth provides a bonus for moves that are sooner rather then later since quick wins are more reliable than several moves ahead. Check provides a bonus for a similar reason as checkmate. Mobility rewards the player for being able to make a lot of moves after this is good because in chess one of the most important factors is positioning and without this the ai will just go for captures and quick points rather then a developed line-up. The PieceValue method sums all of the pieces still alive this will allow the ai to calculate sacrifices.

**Validation Algorithm (API):**

Wrapped by validation middleware

Regex REGEX = new ("<regex string>")

void Validate():

    if REGEX.IsMatch(this.Value):

        throw ValidationError

**Coded Version**

using System.Text.RegularExpressions;

using FluentValidation;

using FluentValidation.Results;

using ValueOf;

namespace ChessWebApp.Api.Domain.Common;

public sealed class EmailAddress : ValueOf<string, EmailAddress>

{

    private static readonly Regex EmailRegex =

        new(@"^[\\w!#$%&’\*+/=?`{|}~^-]+

            (?:\\.[\\w!#$%&’\*+/=?`{|}~^-]+)\*

            @

            (?:[a-zA-Z0-9-]+\\.)+

            [a-zA-Z]{2,6}$",

            RegexOptions.IgnoreCase |

            RegexOptions.Compiled |

            RegexOptions.ExplicitCapture,

            TimeSpan.FromMilliseconds(50));

    protected override void Validate()

    {

        if (!EmailRegex.IsMatch(Value))

        {

            string message = $"{Value} is not a valid email address";

            throw new ValidationException(message, new []

            {

                new ValidationFailure(nameof(EmailAddress), message)

            });

        }

    }

}

\*note regex is split into line per segment for readability of examiner. In practice the regex is all in one line.

Regex: first segment allows the user to enter a string of any allowed characters (at least one). Then the user can optionally put a dot if a dot is entered the user must use at least one allowed character after this can be repeated. Then the user must have an @ symbol. After which the user optionally can enter the first part of the top level domain. If they do this ends with a dot. Then the final part of the top level domain which can be from 2 to six characters. After this we use enums with the bitwise or to use multiple flags these just make the regex slightly more performant. In the validate we check the emails value against the regex if it fails an error is thrown this will be captured by the validation middleware. A similar validation format will be used for all different form values this allows for a very scalable validation style often used in industry.

**Generating Json Web Token:**

string issuer = parse from config json

string audience = parse from config json

byte[] bytes = UTF8StringToBytes(jwt key from config)

SymmetricKey securityKey = new(bytes)

Credentials credentials = new(HmacSha256(securityKey))

JwtTokenHandler handler = new()

TokenDescriptor descriptor =

{

    Subject = ClaimsIdentity(

        {new(RegedName.Sub, username),

         new(RegedName.Email, email),

         new(RegedName.Jti, Guid.New())

        }

    ),

    Expires = in 6 months,

    Audience = audience,

    Issuer = issuer,

    Credentials = credentials

}

SecurityToken token = CreateToken(descriptor)

return token.ToString()

**Coded Version**

public string GenerateJwt(User user)

{

    string issuer = \_configuration.GetValue<string>("Jwt:Issuers:Https");

    string audience = \_configuration.GetValue<string>("Url:UiHttps");

    byte[] keyBytes = Encoding.UTF8.GetBytes(

        \_configuration.GetValue<string>("Jwt:Key"));

    SymmetricSecurityKey securityKey = new(keyBytes);

    SigningCredentials credentials = new(securityKey,

        SecurityAlgorithms.HmacSha256);

    JwtSecurityTokenHandler jwtTokenHandler = new();

    SecurityTokenDescriptor tokenDescriptor = new()

    {

        Subject = new ClaimsIdentity(new []

        {

            new Claim(JwtRegisteredClaimNames.Sub, user.Username.Value),

            new(JwtRegisteredClaimNames.Email, user.Email.Value),

            new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())

        }),

        Expires = DateTime.Now.AddMonths(6),

        Audience = audience,

        Issuer = issuer,

        SigningCredentials = credentials

    };

    SecurityToken token = jwtTokenHandler.CreateToken(tokenDescriptor);

    return jwtTokenHandler.WriteToken(token);

}

Json web tokens are a tool for authenticating a user as valid. They can be used and created via private and public key encryption but also using secrets encryption. My algorithm uses the secrets version (symmetric encryption). The secrets come from the config which is a json file. This config technique is commonly used in dotnet apps and is safe in fact it comes with the project built in. So first to use symmetric encryption we need to register the security key. We first grab it from the config converting it into a byte array we will use the UTF8 conversion of a string since this is convention for web applications. We now use this byte array as the key and create a key object we also turn this key into a credential by passing it through a Sha256 hashing algorithm. With this we can now complete the body of the token. We create new descriptor this holds the values we want our jwt to be encoded with. Now all of the pieces are made we simply pass it to the jwt creator and our token is made we now pass the stringified version back to the caller.

**Board Generation When State Changed Algorithm:**

Psuedo code excluded for this algorithm due to its complex class oriented nature. It will be easier just to use source code as I am used to the class patterns.

Board.cs

private Board()

{

    EnPassantPawn = Builder.EnPassantPawn;

    \_tiles = CreateGameBoard();

    WhitePieces = CalculateActivePieces(\_tiles, Alliance.White);

    BlackPieces = CalculateActivePieces(\_tiles, Alliance.Black);

    List<Move> whiteStandardLegalMoves =

        CalculateLegalMoves(WhitePieces).ToList();

    List<Move> blackStandardLegalMoves =

        CalculateLegalMoves(BlackPieces).ToList();

    WhitePlayer = new WhitePlayer(this, whiteStandardLegalMoves,

        blackStandardLegalMoves);

    BlackPlayer = new BlackPlayer(this, blackStandardLegalMoves,

        whiteStandardLegalMoves);

    CurrentPlayer = Builder.NextMoveMaker.ChoosePlayer(WhitePlayer,

        BlackPlayer);

}

private static List<Tile> CreateGameBoard()

{

    Tile[] tiles = new Tile[Utils.NumberOfTiles];

    for (int i = 0; i < tiles.Length; i++)

    {

        tiles[i] = new Tile(i, Builder.BoardConfig[i]);

    }

    return new List<Tile>(tiles);

}

private static IEnumerable<Piece> CalculateActivePieces(IEnumerable<Tile> tiles,

    Alliance alliance)

{

    List<Piece> activePieces = new();

    foreach (Tile tile in tiles)

    {

        if (!tile.IsTileOccupied)

        {

            continue;

        }

        Piece piece = tile.Piece;

        if (piece.Alliance == alliance)

        {

            activePieces.Add(piece);

        }

    }

    return activePieces;

}

private IEnumerable<Move> CalculateLegalMoves(IEnumerable<Piece> pieces)

{

    List<Move> legalMoves = new();

    foreach (Piece piece in pieces)

    {

        legalMoves.AddRange(piece.CalculateLegalMoves(this));

    }

    return legalMoves;

}

Pawn.cs (example of piece move calc)

public override IEnumerable<Move> CalculateLegalMoves(Board board)

    {

        List<Move> legalMoves = new();

        foreach (int moveTransformation in CandidateMoveTransformations)

        {

            int destinationCoordinate = Position + moveTransformation

                \* Alliance.GetDirection();

            if (!destinationCoordinate.IsValid() ||

                IsExclusion(Position, moveTransformation))

            {

                continue;

            }

            Tile destinationTile = board[destinationCoordinate];

            if (moveTransformation is 8 && !destinationTile.IsTileOccupied)

            {

                if (Alliance.IsPromotionSquare(destinationCoordinate))

                {

                    legalMoves.Add(new Promotion(new Move(board, this,

                        destinationCoordinate)));

                    continue;

                }

                legalMoves.Add(new Move(board, this, destinationCoordinate));

                continue;

            }

            if (IsJumpMove(board, moveTransformation, destinationCoordinate))

            {

                legalMoves.Add(new PawnJump(board, this,

                    destinationCoordinate));

                continue;

            }

            switch (moveTransformation)

            {

                case 7 or 9 when destinationTile.IsTileOccupied &&

                    destinationTile.Piece.Alliance != Alliance:

                {

                    if (Alliance.IsPromotionSquare(destinationCoordinate))

                    {

                        legalMoves.Add(new Promotion(new Move(board, this,

                            destinationCoordinate)));

                        continue;

                    }

                    legalMoves.Add(new Move(board, this, destinationCoordinate));

                    continue;

                }

                case 7 or 9 when board.EnPassantPawn is not null &&

                                 board.EnPassantPawn.Position +

                                    Alliance.GetDirection()

                                    \* Board.Utils.NumberOfRows ==

                                 destinationCoordinate:

                {

                    if (Alliance.IsPromotionSquare(destinationCoordinate))

                    {

                        legalMoves.Add(new EnPassant(board, this,

                            destinationCoordinate));

                        continue;

                    }

                    legalMoves.Add(new EnPassant(board, this,

                        destinationCoordinate));

                    continue;

                }

            }

        }

        return new ReadOnlyCollection<Move>(legalMoves);

    }

This algorithm is for when the state of a board has changed, and all of the related classes must be refreshed. It actually goes deeper then this but for my design I will keep it surface level the multi file solution will be in the technical solution. We start off by getting all of the positional state from the board config in the create gameboard method. The board config is in a builder method that uses static properties to abstract the alteration of game state away from the user of the chess engine to simple build calls. This paradigm is frequently used in OO languages. Now the board is set we can begin setting up the properties that will be used we will first work out all of the players pieces locations and store them in a list. Using these new lists of pieces we can calculate each players move by looping through every piece and working out their moves individually. Now we have all of the data required to complete the blueprint for a play and we will create both of the players. After this we move the game into the next turn using the builder this means on the next board creation the current turn will be different an example of the use of the builder paradigm.

In the pawn class we loop through a list of the possible move vectors. A move vector is the displacement from the current position in terms of tiles. We will go through each of these checking if they are valid (on the board) and that they are not excluded (the vector overflows onto another side of the board). We then use a defensive programming style to avoid nesting, I personally think this style looks neater and is more readable. We go through the special cases such as enpassant and promotion then just add normal moves.

Diagram

Description automatically generatedThese are the simple algorithms that will be made. The chess engine will require a lot of complicated OOP related algorithms to be completed. Below is a class diagram showcasing the complexity of what I want to create.

Class Descriptions for Chess Engine:

Board: Centre of the application

+ WhitePieces : IEnumerable<Piece> <<get>> - All white pieces still in play.

+ BlackPieces : IEnumerable<Piece> <<get>> - All black pieces still in play.

+ AllLegalMoves : IEnumerable<Move> <<get>> - Legal moves from both players.

+ WhitePLayer : Player <<get>>

+ BlackPlayer : Player <<get>>

+ GetEnumerator() : IEnumerator<Tile> - Implemented from IEnumerable allows foreach to loop through tiles.

+ <<override>> ToString() : string – for debugging so if needed we can render the board on the console.

GetEnumerator() : IEnumerator – see GetEnumerator()

+ {static} CreateStandardBoard() : Board – Creates the initial state of the chess board using the builder.

Board.Builder: subclass of the board. Builder design pattern is used to create boards.

+ {static} BoardConfig : IDictionary<int, Piece> <<get>> - Piece and coordinate pair used for the creating of the tiles.

+ {static} NextMoveMaker : Alliance <<get>> - The next current alliance when the board is generated.

+ {static} SetPiece(piece:Piece) : void – Set a piece in the config

+ {static} SetMoveMaker(alliance:Alliance) : void – Set the next move maker

+ {static} Build() : Board – The build method that calls the boards private constructor.

+ {static} SetEnPassantPawn(pawn:Pawn) : void : Set static enPassant pawn

Board.Utils: subclass of the board. Has some common values.

+ <<const>> NumberOfTiles : int = 64 – all constants are self explanatory

+ <<const>> NumberOfColumns : int = 8

+ <<const>> NumberOfRows : int = 8

enum Row { Quality of life enums to stop me from just using integers

First= 0,Second,Third,Forth,Fifth,Sixth,Seventh,Eighth

}

enum Column {

First= 0,Second,Third,Forth,Fifth,Sixth,Seventh,Eighth

}

BoardExtensions: Extension methods related to the Board classes. I love extension methods.

+ {static} IsValid(coordinate:int) : bool – would integer be a valid coordinate

+ {static} ToColumn(coordinate:int) : Board.Utils.Column – integer converted to column enum

+ {static} ToRow(coordinate:int) : Board.Utils.Row – integer converted to row enum

+ {static} ToCoordinate(position:string) : int – fen string notation converted to a number coordinate

+ {static} ToPosition(coordinate:int) : string – coordinate converted to fen string

Tile: A component of the board with a coordinate

+ Piece : Piece <<get>> - the piece occupying the coordinate of the tile

+ IsTileOccupied : bool <<get>> - is the piece not null

+ TileCoordinate : int <<get>> - Coordinate of this tile on the board.

+ Tile(tileCoordinate:int, piece:Piece)

+ <<override>> ToString() : string – displays the tile as a string for debugging. Will be displayed in a grid if used with the board.

Move: Class that handles executing a standard move. Castle, EmptyMove, PawnJump, Promotion, EnPassant all inherit and override from this class.

+ MovedPiece : Piece <<get>> - the piece that will be tried to move

+ DestinationCoordinate : int <<get>> - the location we want to move this piece to.

+ IsFirstMove : bool <<get>> - Has the piece moved before

+ Move(board:Board, movedPiece:Piece, destinationCoordinate:int)

+ <<virtual>> Execute() : Board – carry out the move

+ KeepStateExcludingPieces(excludePieces:Piece[]) : void – Keep every piece on the board except the pieces passed.

Move.Builder: Subclass of move. Another use of the builder design pattern.

+ {static} CreateMove(board:Board, currentCoordinate:int, destinationCoordinate:int) : Move

Enum MoveStatus { useful enum for the progress of a move.

Done,Illegal,LeavesPlayerInCheck

}

MoveTransition: Allows us to make a move and check if its been completed or not.

+ MoveStatus : MoveStatus <<get>> - see move status

+ Move : Move <<get>>

+ Board : Board <<get>>

+ MoveTransition(transitionBoard:Board, move:Move, moveStatus:MoveStatus)

Piece: an abstract class for the pieces. Is completed by the classes DirectionalPiece, King, Knight, Pawn. Directional Piece is further implemented by the classes Bishop, Queen, Rook.

+ Alliance : Alliance <<get>> - so we know if the pieces can move during a turn

+ Position : int <<get>> - coordinate of the piece

+ IsFirstMove : bool <<get>> - Has the piece moved before

+ <<override>> ToString() : string – So we can display the piece as a symbol when debugging

+ {abstract} CalculateLegalMoves(board:Board) : IEnumerable<Move> - Calculates every legal move that can be performed by the piece

+ {abstract} MovePiece(move:Move) : Piece – attempts to move the piece

Player: an abstract class for representing a player. Is completed by the classes WhitePlayer and BlackPlayer.

+ {abstract} Alliance : Alliance <<get>> - Players alliance

+ {abstract} ActivePieces : IEnumerable<Piece> <<get>> - pieces based off alliance

+ {abstract} Opponent : Player <<get>> - other player (so we can see his moves)

+ King : King <<get>> - So we know where the king is

+ LegalMoves : IEnumerable<Move> <<get>> - Legal moves of all of the pieces

+ IsInCheck : bool <<get>> - is king under attack

+ IsInCheckmate : bool <<get>> - are checkmate conditions met ending the game.

+ IsInStalemate : bool <<get>> - are stalemate conditions met ending the game.

+ IsLegalMove(move:Move) : bool – Confirms if a move can be made.

+ MakeMove(move:Move) : MoveTransition – attempts a move

Enum Alliance { Helpful enum for determining what alliance components are

None,White,Black

}

AllianceExtensions: Extension methods for the alliance enum

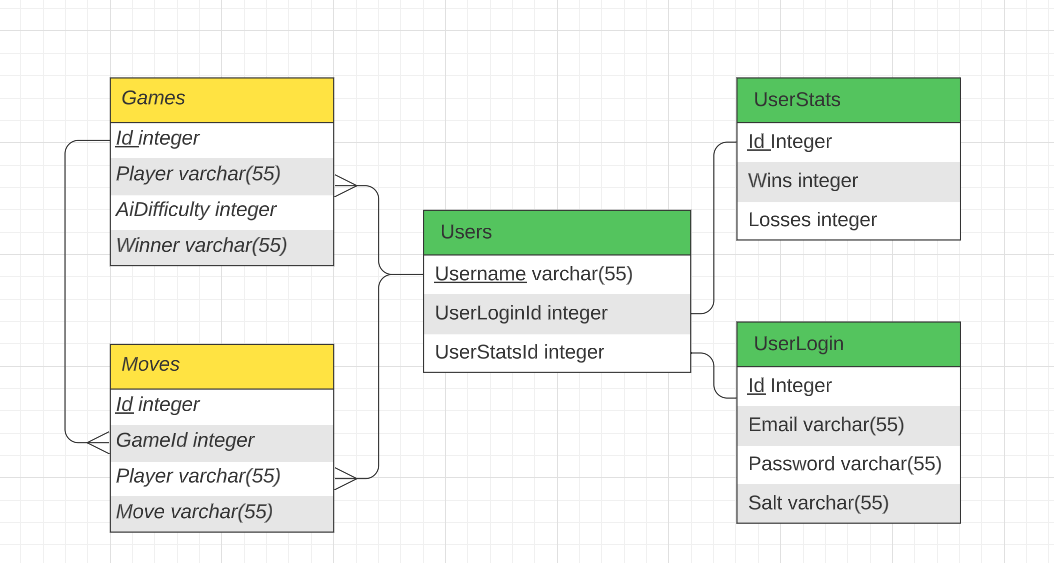
+ {static} GetDirection(alliance:Alliance) : int – gives the directional coefficients of the pawn movements for an alliance (-1 or 1)

+ {static} ChoosePlayer(alliance:Alliance, whitePlayer:Player, blackPlayer:Player) : Player – from alliance decide which player it corresponds to

Structured Query Language

My project will consist of several cross linked tables. These tables will be normalised to allow for a more flexible database design. Conceptially the tables are split between two main areas: the storing of game data and user information. It is likely that when I come to my solution I will break the SQL repositories into these key areas to keep the program modular.

**Entity Relationship Diagram:**

****

Yellow tables are relevant to games and green tables are relevant for users. A user has login data and statistic data. The statistics consists of wins and losses storing this allows us to calculate a users win rate providing insight into who the best player is. The login information consists of the email, a hashed password and the salt used to hash the password. Having the password be hashed prevents users of the database management/api management to not be able to see the raw password.

The games table provides a list of every game showing the player the difficulty they selected (1 to 3 at the moment) and who won. This will allow us to track a users match history if desired to see how they win. The moves table will by far be the biggest table we store every move stating the game id the player that moved it and the move. This will allow anyone to use the api to analyse a players move history such as finding their most common move.

**Baseline CRUD SQL:**

For a standard crud api we will need a series of sql queries. Note that for the following queries the @ signifies we will be getting this data from parameters.

User create:

INSERT INTO UserLogin

(

    Password,

    Salt,

    Email

)

VALUES

(@Password, @Salt, @Email);

INSERT INTO UserStats

(

    Wins,

    Losses

)

VALUES

(@Wins, @Losses);

INSERT INTO Users

(

    Username,

    UserLoginId,

    UserStatsId

)

VALUES

(@Username, (SELECT max(Id) FROM UserLogin), (SELECT max(Id) FROM UserStats))

We start by populating the user login and user stats this is not relational so a simple insert can be applied. Then for the user we know that the ids we are going to use will be the most recent. Most recent will be the largest integer since it increments so we use the aggregate max function to gather this.

User read:

SELECT

    \*

FROM

    Users

    INNER JOIN

        UserLogin

    ON

        Users.UserLoginId = UserLogin.Id

    INNER JOIN

        UserStats

    ON

        Users.UserStatsId = UserStats.Id

WHERE

    Username = @Username;

To select all of a specific users data we will cross link the tables using an inner join on the ids and afterwards just get it using the unique username.

User update:

UPDATE

Users

SET

    Username = @Username,

    Password = @Password,

    Email = @Email,

    Wins = @Wins,

    Losses = @Losses

FROM

    Users

    INNER JOIN

        UserLogin

    ON

        Users.UserLoginId = UserLogin.Id

    INNER JOIN

        UserStats

    ON

        Users.UserStatsId = UserStats.Id

WHERE

    Username = @Username

For a complete update will we yet again cross join the tables using an inner join and from there we will just set the values on the unique username.

User delete:

DELETE

FROM

    Users

WHERE

    Username = @Username

Delete is the simplest no cross tabling is required. The reason we do not want to delete the login information and win rate is so that a user cannot just remove their history with my app by just deleting their account. Also saving the login allows us to recognise emails so a user cannot generate unlimited accounts on one email.

This is all just a simple CRUD for the user there will also be similar queries for the game and move. Also, there will be further queries for the user for when we only want statistics information for example, but these are the baseline queries that all others will be built upon.

Data Analysis

Sometimes we will need to write queries for data analysis such as finding a player favourite move. Here is that query:

SELECT

    Move

FROM

    Moves

WHERE

    Player = @Player

GROUP BY

    Move

ORDER BY

    COUNT(\*) DESC

LIMIT 1;

This query first groups all move records that have the same name then orders these groups by the group that has the largest count using the aggregate function from here we simply return the move at the top of the table since this is the move with the highest count.

Another query I will use for analysis is a win rate calculation:

SELECT

    COUNT(\*) \* 100.0 / (

        SELECT

            COUNT(\*)

        FROM

            Games

        WHERE

            Player = @Player

    ) AS WinRate

FROM

    Games

WHERE

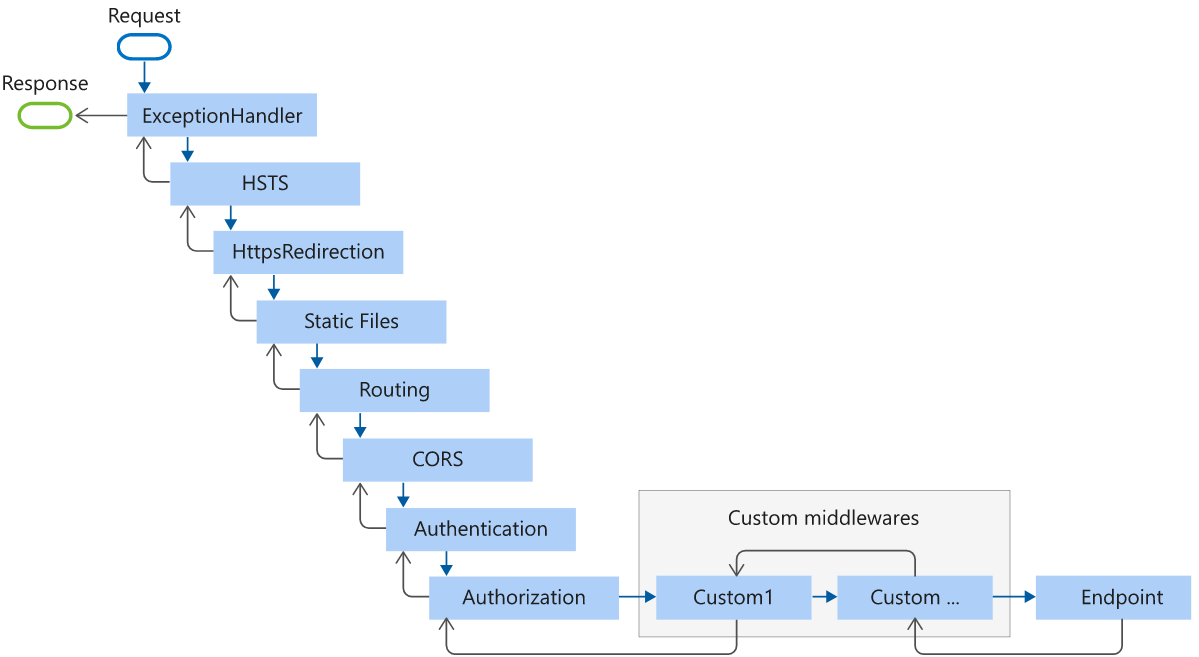
    Player = @Player AND Winner = @Player

In this query I begin by getting the number of games that belong to the player where they win. I multiply this by 100 so it is a percentage I then divide by the number of games that belong to the player in total.

This can be expressed by:

API Architecture

During my work experience placement in year 12 I had the fortunate opportunity to work with software engineers creating a proof-of-concept app. The lead engineer taught me a lot about the importance of good architecture when creating a backend. I will be creating a plan of how to apply this knowledge to good effect.

I will start by assessing the tools I will need. I will be using .NETs minimal apis this is a rather new thing in dotnet. Minimal apis allow me to make extensive use of .NETs dependency injection system. This elegant system allows me to turn a complicated multi file app into a series of 1 line steps this is often called a pipeline.

I will be using my own custom exception handler which will be applied as a middleware that almost wraps every step so that will look different however this pipeline is exactly how my pipeline will be. The nitty gritty stages including HSTS, HttpsRedirection, static files and routing are kindly dealt with by the project type itself so I do not have to worry about implementing them.

CORS is relatively trivial to setup I will be allowing anything from the domains of my website, but any other address will be denied. This provides a level of security to my requests. I will subscribe my websites addresses to the builder and that will be that layer sorted.

Authentication and authorisation will be dealt with by JWTs. This is like CORs in the fact it is just a series of options I am either ticking off or not I will select JwtBearer as my authentication process.

Now all that is left is to register the endpoints.

[HttpGet("users/{username}")]

public sealed class GetUserEndpoint : Endpoint<GetUserRequest, UserResponse>

{

    private readonly IUserService \_userService;

    public GetUserEndpoint(IUserService userService)

    {

        \_userService = userService;

    }

    public override async Task HandleAsync(GetUserRequest req, CancellationToken ct)

    {

        User? user = await \_userService.GetAsync(req.Username);

        if (user is null)

        {

            await SendNotFoundAsync(ct);

            return;

        }

        UserResponse userResponse = user.ToUserResponse();

        await SendOkAsync(userResponse, ct);

    }

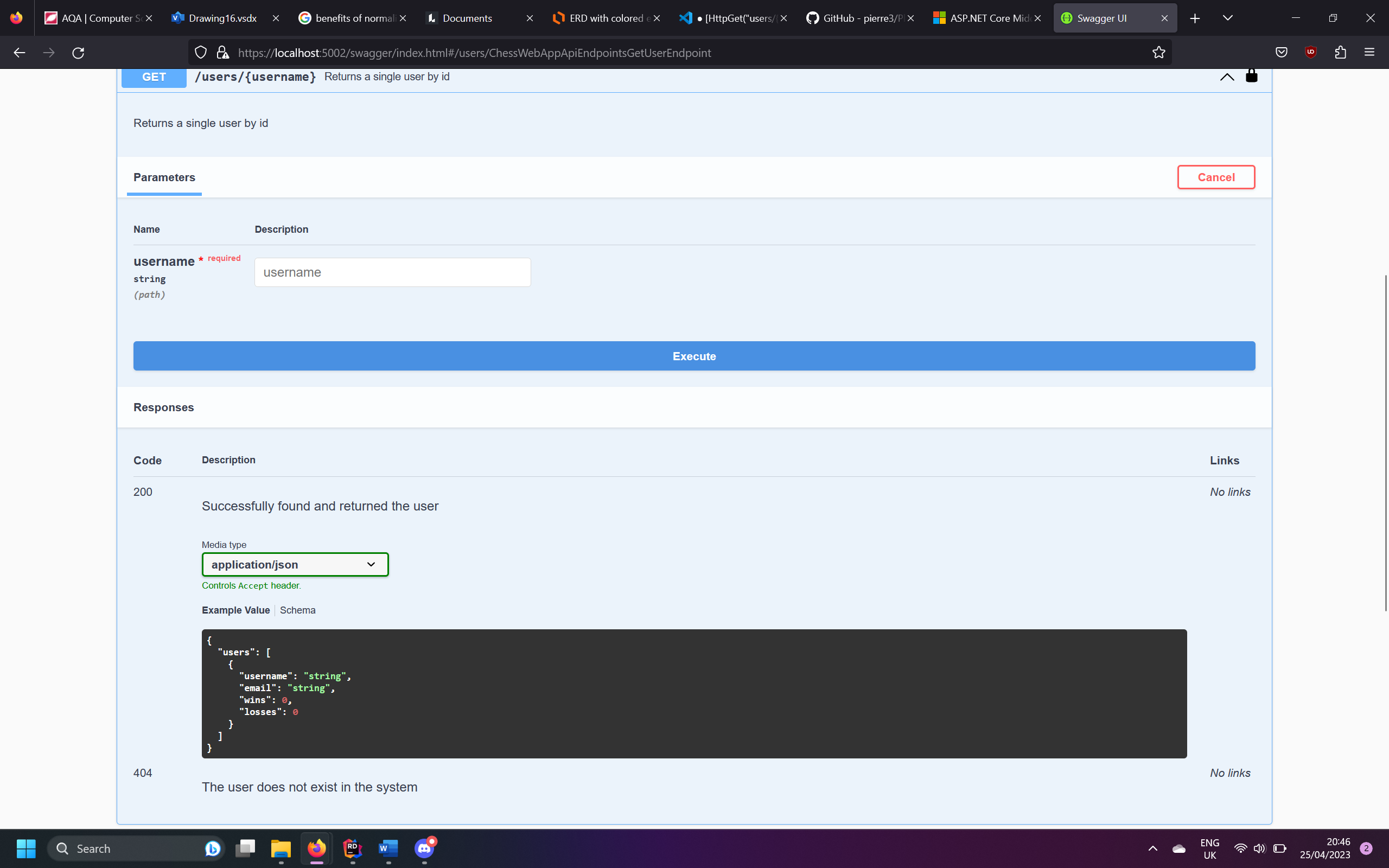
}

Above is an example of an endpoint I cooked up. I have used fast endpoints. Fast endpoints scans the assemblies of the compiled solution to look for any classes that have inherited endpoint and registers them. This removes the need for me to manually hook it up. Notice the use of a user service the service will rap the SQL repository the reason we don’t just implement straight sql into the endpoint is so that when errors occur it becomes obvious the point of concern rather then having to fight if the issue is a logical one or not.

After creating a class for each of the endpoints my api pipeline will be complete and will be runnable.

I could manually test each request through the network tab on the browser but that is way to slow. That is why it is convention to use a visualisation tool such as swagger. I simply add swagger to my pipeline, and it picks up on all the details of my endpoints and also my models.

Example of swagger for the endpoint I have shown.



Note this endpoint may or may not exist in this form in the actual solution.

We can also see the schema.

