## Validation techniques

I use both server and client validation for various fields. As shown below I have two functions to ensure that the username is valid. A JavaScript function that prevents the submission of the form on the client. But I also have a server-side function in case the client-side function is bypassed to prevent damage.





I also use a confirm password box to ensure that the user does not type a password with typos and block the submit until they both match.





## Complex algorithm

For my complex algorithm, I have decided to use an ELO ranking formula. This uses a logistic curve to calculate the probability of a player winning or losing. This curve was chosen over normal distribution because it modelled the real-life situation more in which higher players tend to underperform while lower players tend to over perform as based on a study. Compared to other formulas such as Microsoft’s Trueskill or Glicko rating system which factor in uncertainty and other values, this makes it difficult to create a leader board with values that players can understand. The logistic curve ELO formula is a very well-known and understood by the chess community allowing them to interpret its values. Also, a constant k-factor was chosen since assigning a separate k-factor for each player based on other factors would alter the total amount of ELO in the system making it not a zero-sum game.



## Efficient and secure access of files

I use authentication to allow users access to their information. If a user is not authenticated when they try to access a login required page, then they get redirected to the login page and notified to log in.

I decided to use a SQL database because it allows me to query the database for the required information and allows me to take advantage of primary and foreign keys. Compared to a XML, it would take up less storage memory. However, a SQL database is designed to handle concurrent access as two users could simultaneously register and require the database to store their credentials which would cause problems if a CSV or XML was used.

I also used the Username as the primary instead of an auto-generated ID since all username are required by the application to be unique making it suitable to be a primary key. I also use the appropriate sizes for fields especially for Salt and Password which are always going to be of that size.

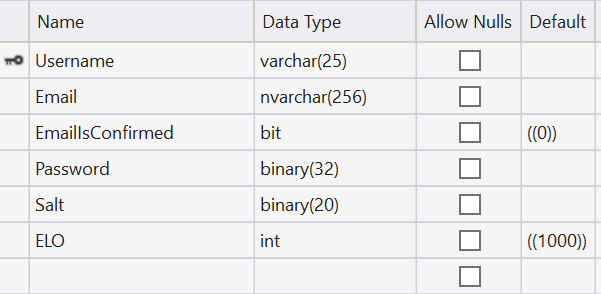


Figure 1 Player table

This obviously saves more space and makes it easier to make queries involving joins since an ID does not need to be found from the username.



I have also separated the confirmation and reset token storages into their own tables rather than with the player table. This was done to save space as the whole row can be deleted rather than having empty columns. This is especially true for the email confirmation token which is only required until the user confirms their email and then this storage is never used again for that user.

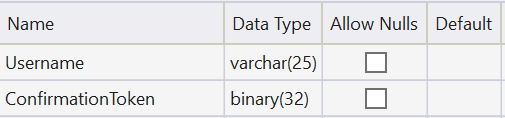


Figure 2 Confirmation Token table

I also have a password reset mechanism in case a user forgets their password. This would quickly allow them to regain control of their account in that event,

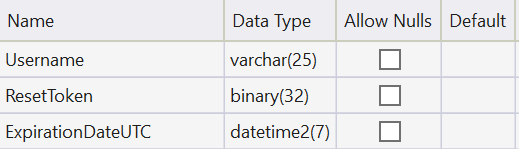


Figure 3 Reset Token table

## Appropriate data structures

(Some class methods are omitted)

I use dictionaries map each individual client to their relevant information. I combine these with the classes for the player and the game to make manipulating them easy. Because I am handling concurrent connections, it is also necessary to use the concurrent version to avoid errors or using locks for a workaround. It was also necessary to use a Lazy Singleton so that only once instance would be made with the SignalIr hub context so that it could call client methods. It was also necessary to use a dictionary since they are mutable and I can add or remove values.



My play connection class stores the individual players information such as which side they are on, their connection ID and the ID of the game they are playing.

My game class then stores the board state, the players and its own ID. It also stores a timer that executes if the player does not make a move in time.



## Security

For security, I have implemented a salted hash function to store passwords. As shown below I use a secure crypto library to generate a random salt that cannot be guessed then I append that salt to the end of the bytes of the plaintext password. I also use the same function to generate the tokens so they cannot be brute forced.



Finally, I hash this concatenated value and store it in a database. Then when a login occurs, I retrieve the hash and salt from the database. Concatenate the salt to the plaintext password, then hash it. I compare it with the hashed stored in the database to tell whether the credentials are correct. Altogether this prevents a database leak from exposing passwords in plaintext while the salt prevents using precomputed lists of hashes to easily lookup the password.



I also enforce password rules to make sure the passwords are not easily guessable.



I have also implemented database authentication to make it more difficult manipulate it without the right authentication.

