Project

*DAT602 – Database Application Development*

Jackson Hayes

NMIT ID - 13521772

S2 2024

Table of Contents

[Milestone One 4](#_Toc182384225)

[Project Introduction 4](#_Toc182384226)

[Storyboards 6](#_Toc182384227)

[Home Screen 7](#_Toc182384228)

[Register Screen 7](#_Toc182384229)

[Login Screen 9](#_Toc182384230)

[Main Lobby 10](#_Toc182384231)

[Tiles On a Game Board 12](#_Toc182384232)

[Placing Items on Tiles 13](#_Toc182384233)

[Player Gameplay 14](#_Toc182384234)

[Item Movement (NPC Effect) 15](#_Toc182384235)

[Admin Console 16](#_Toc182384236)

[Edit profile Screen 17](#_Toc182384237)

[Add player screen 18](#_Toc182384238)

[ERD Diagram 20](#_Toc182384239)

[Entity Relationship Diagram Rationale 21](#_Toc182384240)

[Player 21](#_Toc182384241)

[Game 21](#_Toc182384242)

[Game\_Player 21](#_Toc182384243)

[Tile 21](#_Toc182384244)

[Item 22](#_Toc182384245)

[Inventory 22](#_Toc182384246)

[Item\_Inventory 22](#_Toc182384247)

[Chat Session 22](#_Toc182384248)

[Chat 22](#_Toc182384249)

[CRUD Analysis 23](#_Toc182384250)

[CRUD Process Descriptions 27](#_Toc182384251)

[SQL Queries and Procedures 28](#_Toc182384252)

[Login and Registration 28](#_Toc182384253)

[Gameplay 29](#_Toc182384254)

[Administration 29](#_Toc182384255)

[Milestone Two 29](#_Toc182384256)

[Implementation 30](#_Toc182384257)

[1. Player login, including lock out 30](#_Toc182384258)

[2. Player registration 30](#_Toc182384259)

[3. Laying out tiles on a game board 30](#_Toc182384260)

[4. Placing an item on a tile 31](#_Toc182384261)

[5. Player Movement 31](#_Toc182384262)

[6. Game play scoring 31](#_Toc182384263)

[7. Player Acquiring Inventory 31](#_Toc182384264)

[8. Move an Item (NPC effect) 32](#_Toc182384265)

[9. Kill running games 32](#_Toc182384266)

[10. Add new players 33](#_Toc182384267)

[11. Update data of a player 33](#_Toc182384268)

[12. Delete a player 33](#_Toc182384269)

[Milestone Three 33](#_Toc182384270)

[C# .Net application Exception Handling 33](#_Toc182384271)

[GUI Methods 34](#_Toc182384272)

[DAO Methods 46](#_Toc182384273)

[Concurrency Management 52](#_Toc182384274)

[MySQL Concurrency 53](#_Toc182384275)

[C# .NET Concurrency 53](#_Toc182384276)

[Comparison 54](#_Toc182384277)

[References 54](#_Toc182384278)

# Milestone One

## Project Introduction

The game ‘BerryBlitz’ is a 2D berry picking multiplayer game that combines strategy and competition in a simple environment. The game is designed to allow 2 players to compete against each other with the goal of collecting the most berries located around the game board or reaching a predetermined score before the other person. Players will navigate a tiled map on a turn based system, moving from one tile to another, collecting berries to increase their score.

To make the game more engaging, there will also be “Poisonous Berries” scattered around the board that will move position every 2-3 turns. The poisonous berries will appear different to the regular berry tile and if collected by a player will deduct points from the player’s score.

A further analysis of the game mechanics and interactions include:

1. **Player interaction:**
   * Players interact with the game board through a point and click interface, allowing the player to move across a grid-based map. Each move should be a strategic decision as players attempt to collect the good berry tiles and avoid the poisonous ones.
   * The game begins with players starting on designated “Home Tiles”, from here they can move out further onto the board by clicking on adjacent tiles during their turn. Movement is restricted by the rule that only one player can occupy a tile at a time, which adds a small amount of strategy to the game.
2. **Map and tiles:**
   * The game map consists of a grid of tiles, each representing a specific tile type. Tiles can hold items (Berries), be occupied by players or represent a home/starting tile
3. **Item collection and inventory:**
   * Placed across the map are different items represented by tile type. This includes:
     1. a home tile, where the player will start the game
     2. a berry tile, where the player will collect a berry and increase their score
     3. a poisonous berry tile, where if collected by the player, they will lose score. These will move around the map every 2-3 turns
   * The players inventory can be represented by the amount of score/berries the player has in that specific game. This will be needed to end the game when a predetermined win condition is added.
4. **Game state:**
   * The game maintains a persistent state in the database, ensuring that all player actions or item collections are updated in real time. This allows the game to support being played by multiple people
   * When a player exits a game, their current state, including their position and score for that game are saved. Upon returning, the player can resume where they left off.
5. **Multiplayer:**
   * The game will support multiplayer interactions, with players competing against each other on the same game map. Communication between these 2 players is available through the in-game chat system.
   * The main lobby will display a list of online players and their high score.
6. **Game objective:**
   * The main objective of the game is to collect a predetermined number of berries before the other player. The first to reach this target score wins the game
   * In addition, there will be “Poisonous berries” located around the game board which will move around intermittently. Collecting one of these berries will reduce the players score and require them to collect an additional berry to win the game

## Storyboards

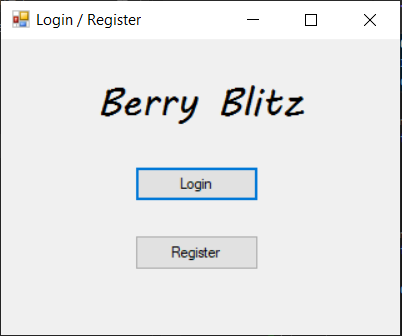
A screenshot of a computer

Description automatically generated

The general flow of the application follows the screenshot above, starting with the home screen on the far left of the screenshot, where the user has the option to either login or register an account. Each button opens up their respective form (Login button opens login page, register button opens register page) Once logged into an account the user is then moved to the main lobby where they can choose to create a new game of join an existing one. If the user account has administrator privileges, then the option to open up the admin console in the main lobby will show, otherwise it won’t be displayed to regular users.

A more detailed look into each storyboard and their components are included below:

### Home Screen



**Components:**

1. **Login Button:** Opens up the login screen window
2. **Register Button:** Open up the registration screen window

Once the application is run, the home screen (Login and Register options) will be displayed as the first screen of the application. The user has the option to either login or register a user account which will also open the respective form for whatever button is pressed.

**User story:**

As a new or returning user of the BerryBlitz game, I want to be presented with a home screen that gives me the option to either log in to my existing account or register a new one, so that I can easily access the game based on my current account status.

### Register Screen

A screenshot of a login form

Description automatically generated

A screenshot of a computer error

Description automatically generated

**Components:**

1. **Register Button:** When clicked and the entered credentials are valid (don’t link to an existing account) the account will be created in the database.
2. **Cancel Button:** Closes the registration screen and returns the user to the home screen.
3. **Error pop up:** Error message displayed if the user tries to register an account with the same name as an account that already exists. A similar pop up message will be displayed if required input fields are left empty.

If the user chooses to register a user account from the home screen, they will be brought to the register page shown above. Here the user will need to enter some basic information such as a username, password and email address to register. If successful, the user will be prompted by a pop-up window saying their account was created. They can then proceed to close this screen, opening the home screen again and login as usual.

The potential outcomes for the register screen include:

* Entered user credentials are valid and aren’t associated with an existing account. The account is successfully created.
* Information entered is not valid and/or links to an existing account (same email or username). User account is not created.
* Information is entered into the form but cancel button is pressed before registering. User account is not created and the screen closes without saving any information.

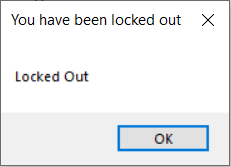
**User story:**

As a new player, I want to create a user account by entering my username, password, and email on the registration screen, so that I can securely participate in the game. If I try to use an existing username or leave required fields empty, I expect to receive an error message to guide me in successfully registering

### Login Screen

A screenshot of a computer

Description automatically generated



**Components:**

1. **Login Button:** When clicked and if login credentials are valid, the user will be redirected to the main lobby screen for the application.
2. **Cancel Button:** Returns the user to the home screen.
3. **Locked out pop up:** Error message displayed once the user account has exceeded the limit of login attempts. Removes access to their account and requires the administrator to unlock it.

If the user chooses to login to an existing account from the home screen, they will be brought to login screen shown above. Here the user must enter a valid username and password, then click the “Login” button to proceed.

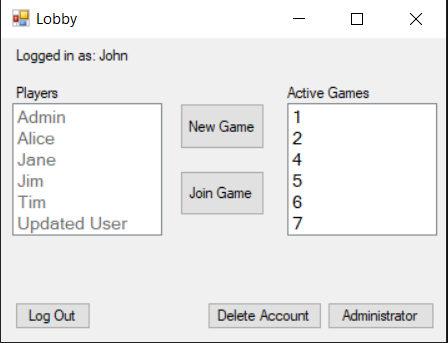
The potential outcomes for the login screen include:

* Entered user credentials are valid and link to an existing account. The user will be successfully logged in and redirected to the main lobby of the application
* Invalid credentials or the account doesn’t exist. A pop up window displaying an error message will be displayed and the user will need to try login again. Login attempts counter tied to the user account will increment
* User account has been locked due to exceeding the number of login attempts or account is banned. An error message will display when they attempt to login. They are unable to login to their account and require the system administrator to unlock.
* The user can exit out of the login screen via the “Cancel” button, returning them to the home screen.

**User story:**

As a returning player, I want to enter my username and password on the login screen to access the main lobby of BerryBlitz, so that I can continue where I left off or start a new game. If I exceed the allowed login attempts, I expect my account to be locked and to receive a notification, prompting me to contact the administrator for account reactivation.

### Main Lobby



Components:

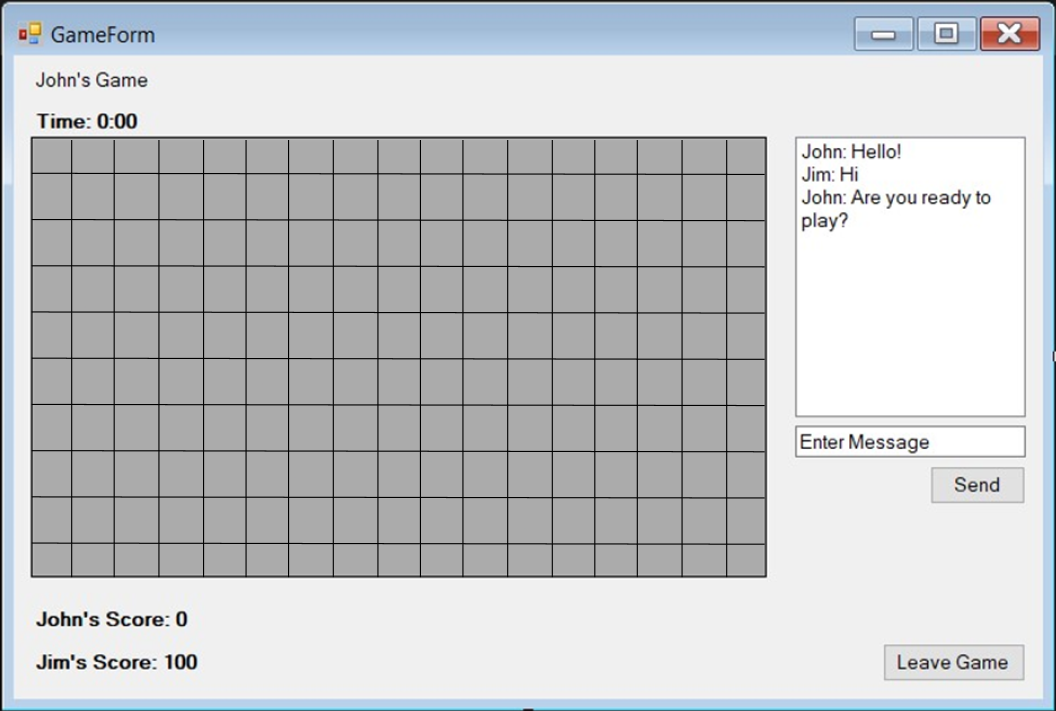
1. **Players List box** – A list of players currently registered and their high score.
2. **Active games List box** – A list of the active games that a player can join. Shows the owner of the game and the amount of people currently in that game.
3. **New Game** – When clicked, a new game will be created in the database, and the user will be redirected to the game window. Creating a new game will lay out the tiles on the game board depending on the dimensions (row, col) that is preset in the database.
4. **Join Game** – When clicked, the user will join the game that is currently selected in the ‘active games’ list box. If nothing is selected, the user will be prompted to select an active. If the game is full (already has 2 players) and error will be displayed telling the user that the game is full.
5. **Log Out** – When clicked, the application will exit and reopen the home screen. This will log the user out of their account and require they log in to continue.
6. **Delete Account** – This button allows the user to delete their own account after confirming that want to do so. Permanently removes the user account from the database.
7. **Administrator** – This button will only appear to accounts that have admin privileges. When clicked another window for the admin console will open, where the user can manage the application.

Shown above is the main lobby for the application and will appear once the user has successfully logged into an account. The lobby works as a hub for the application, giving the user an overview of other users, current games, and the option to create a new game or join an existing one. The user has the ability to log out of their account once they are finished via the “Log Out” button in the bottom left corner. If the user account has administrator privileges, the button for the admin console will appear in the bottom right corner, otherwise, the admin console button will not show for regular users.

**User story:**

As a logged-in player, I want to see a main lobby with a list of other online players and their account statistics, as well as active games I can join, so that I can quickly engage in the multiplayer experience. I want options to log out, delete my account, create a new game, or join an active game if available. If I am an administrator, I want additional access to the admin console.

### Tiles On a Game Board



**Components:**

1. **Game board** – A game board consisting of tiles and where the game will be played. The number of tiles generated during game creation depends on the preset amount of rows and columns set in the database.
2. **Game specific information** – Information relevant to that specific game is displayed around the screen (top left and bottom left corners). This includes the owner of the game, the total time the game has been running for and both players scores.
3. **Chat system** – Here both players have the option to chat between each other if they wish. Messages should stay consistent when loading or closing the game.
4. **Leave Game button** – When clicked, players can leave the game and return to the main lobby.

This is the general layout and expected components for each game board created by a user. The size of the gameboard is preset in the database and generates the appropriate number of tiles depending on the set amount of max rows and columns. Each generated tile has the chance to randomly be assigned an item/tile type for the player to collect, otherwise the tile will be marked as empty.

Each game will have its own chat session, which is shown on the right hand side, where the people playing in that game will have the option of chatting with each other.

Basic information such as who’s game it and time in game are shown at the top of the screen and scores displayed at the bottom. The user will have the option to leave the game which will return them to the main lobby of the application.

All information is expected to save and load in case users wish to continue with the game at a later date. This includes the current game state, chat messages, score and time.

**User story:**

As a player in a game session, I want to view a game board composed of tiles that display important game information like scores, game duration, and player identities, so that I can track progress. I want the option to leave the game and return to the lobby at any time, while knowing my game state will be saved, allowing me to resume later.

### Placing Items on Tiles

A screenshot of a computer game

Description automatically generated

Once a new game has been created, a procedure will automatically trigger during the board creation process to randomly select tile types (items) to place on each tile of the board. As seen in the screenshot above, there are 5 different kinds of tiles, with 3 of them being items a player can acquire and add to their inventory.

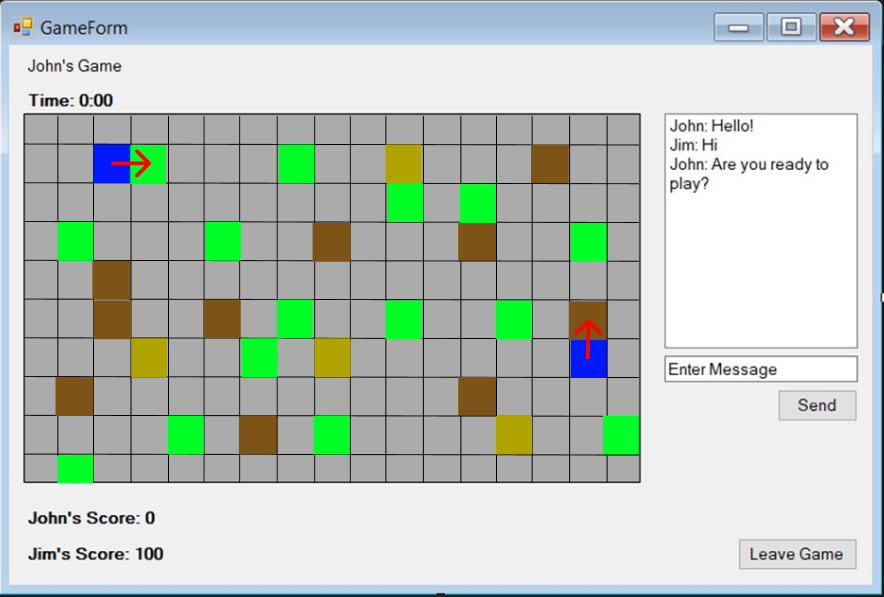
**Tile Key:**

* **Blue Tile:** A blue tile represents the player. This is also where players will first start the game, for example, player 1 will start in the top left corner of the board and player 2 starting in the top right corner.
* **Grey Tile:** A grey tile is an empty tile which contains no item and is used to fill up the game board. The player can move to these tiles, but only if it is a valid move (adjacent to their current position).
* **Green Tile:** A green tile is used to represent the ‘Berry’ Item. They act just like a grey tile except the player will gain points/score and get a berry added to their inventory upon moving to a green tile.
* **Golden/Yellow Tile:** This tile is representing the ‘Golden Berry’ Item. The Golden tile acts exactly the same as a green tile but awards the player will more points once collected. There is a low chance for these tiles to be generated on a game board.
* **Brown Tile:** The brown tile is used to represent the ‘Thorn’ Item. They act just like a grey tile except the player will lose points/score and get a thorn added to their inventory upon moving to a brown tile.

**User story:**

As a player starting a new game, I want to see various item types randomly placed across tiles on the game board, including berries and poisonous thorns, so that the game environment feels dynamic and offers different strategies for collecting points.

### Player Gameplay



Players can navigate around the game board by clicking on adjacent tiles with the cursor which will move their character to the new location/tile. If the player tries to move to a tile that is not adjacent and/or an invalid move, an error will be displayed. Players will receive the same error if trying to move to a tile which is out of bounds or already occupied by another player.

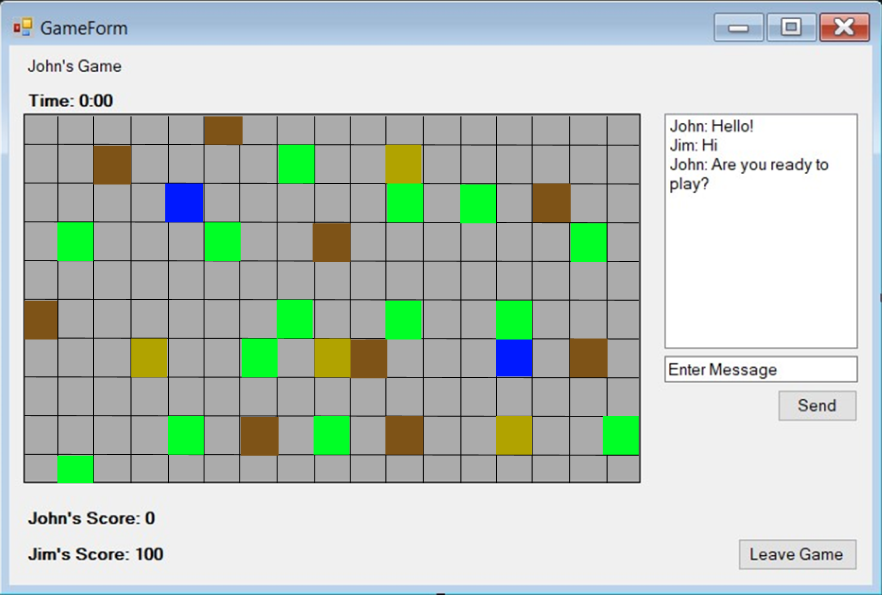
Players can gain and lose points by moving their character to one of the tiles containing an item. If the player moves onto a Green or Golden tile, they will gain points, If they move to a brown tile, they will lose points. Once a tile containing an item gets occupied by a player, the item will be removed from the game board and placed in the player’s inventory.

The inventory is specific to each player and each game, meaning that a player can have many different inventories and will have a different one each game they participate in. The inventory will display basic information such as quantity of items acquired and amount of points gained during gameplay.

**User Story:**

As a player, I want to navigate the game board by moving my character to adjacent tiles, with the goal of collecting berries for points while avoiding thorns that deduct points. My movements should be restricted to adjacent and unoccupied tiles, and my inventory should update based on items I collect during gameplay.

### Item Movement (NPC Effect)



After every 5 player movements, the brown tiles (Thorns) will move position on the game board to a random location. This will cause only the existing brown tiles to move location and will create new thorn locations on the game board depending on how many thorns there were to begin with. This adds a dynamic to each game and keeps it from being repetitive once a game has started.

These thorn tiles/items will move to any other tile on the game board at random with the exception that the tile does not already contain an item or is being occupied by a player.

User Story:

As a player, I want the game to have a random dynamic like moving thorns to different tiles after every 5 moves, adding an unpredictable element to the game that makes gameplay more challenging and strategic.

### Admin Console

A screenshot of a computer

Description automatically generated

**Components:**

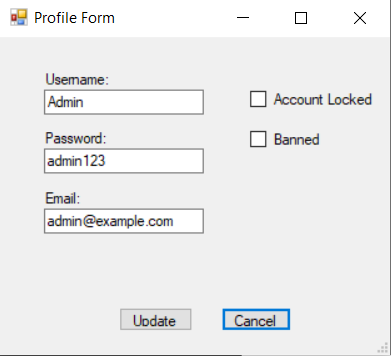
* **Players List** – A list of all the player accounts registered in the application. The admin can select a record by clicking on its row.
* **Current Games List** – A list of all the games currently active or created in the application. The admin can select a record by clicking on its row.
* **Add player** – The admin can add a new user account via the admin console. This will open the profile screen, where they can enter user credentials similar to the register screen process.
* **Edit player** – when clicked, the profile screen will be displayed, allowing the administrator to edit the details of an existing account.
* **Delete** – when clicked, the selected row will be deleted. This can be either an existing account or active game. The admin will be prompted for confirmation before deleting/terminating.

The admin console can only be accessed by user accounts with administrator privileges and gives them the ability to manage the application by adding player accounts or editing existing ones. The admin can also choose to delete existing accounts or terminate active games.

**User story:**

As an administrator, I want access to a console that allows me to view all player accounts and active games, add new players, edit existing accounts, and delete player profiles or terminate games, so that I can maintain control over the game environment and manage user access effectively.

### Edit profile Screen



**Components:**

1. **Text boxes** – Text boxes for the administrator to edit the account details such as the username, password, email address and locked/banned state linked to the account
2. **Update button** – when clicked, will save and update the information currently in the text boxes to the database for the current player.
3. **Cancel button** - when clicked, will close the profile management window and return the user to the admin console.
4. **Error message** – An error message will be displayed if the user tries to update the username to one that is already registered within the application.

The Edit profile screen is used as a way for the administrator to manage and update existing user account details. Once the admin is finished, the Update button will save the details currently entered in each text box or state of the checkbox. The cancel button will close the profile screen and return the user to the admin console, not saving any entered information.

**User story:**

As an administrator, I want to edit a player’s account details, such as username, password, or status, and update the database with these changes, so that I can ensure player information is accurate and up to date without creating a new account

### Add player screen

A screenshot of a login box

Description automatically generated

**Components:**

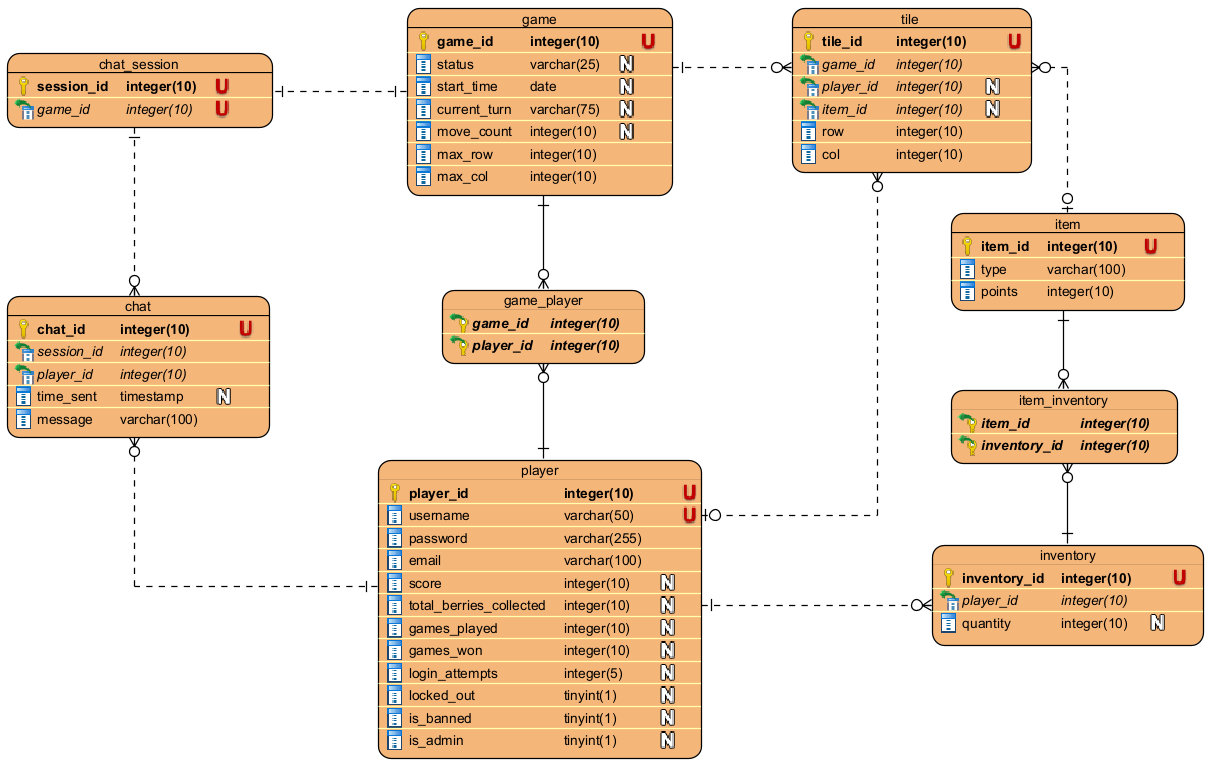
1. **Text boxes** – Text boxes for the administrator to add the new account details such as the username, password and email address linked to the account.
2. **Add user button** – when clicked, will save the information currently in the text boxes to the database, creating a new player record.
3. **Cancel button** - when clicked, will close the profile management window and return the user to the admin console.
4. **Error message** – An error message will be displayed if the user tries to update the username to one that is already registered within the application.

The Add player screen is used as a way for the administrator to manually add player accounts in the application. Once the admin is finished, the Add User button will save the details currently entered in each text box and create a new player record in the database. The cancel button will close the screen and return the user to the admin console, not saving any entered information. This feature acts as another way to register player accounts, but only can be done by an administrator

**User story:**

As an administrator, I want to create a new player account by entering details like username, password, and email in text fields, allowing me to add users directly without requiring them to go through the public registration process.

## ERD Diagram



## Entity Relationship Diagram Rationale

### Player

The Player table stores all information related to the players who use the application. This includes login information (username, password, email and login attempts), gameplay statistics (score, total berries collected, games played and games won) and their account status level (locked, banned, admin, etc.). Each player is uniquely identified by ‘player\_id’.

The Player and Game tables within the database are linked together through a join table due to the many to many relationship. This is because a player can join and has statistics related to many games and a game needs many players to function correctly.

### Game

The Game table tracks individual game sessions. It contains information about the game’s status, the date when it was created, who’s turn in the game it currently is and the link to the associated chat session. There are also values for setting the size of the game board (max\_row, max\_col) and a tracker to see how many moves have been made during that game for dynamic NPC movement reasons.

As mentioned above, the Game and Player table share a many-to-many relationship, besides this, the Game table shares relationships to both Tile and Chat Session tables. Because each game have their own separate chat sessions and a chat session cannot belong to another game, they share a one to one relationship. There is a one-to-many relationship between Game and Tile due to the game board having many tiles but the tile only belonging to one game.

### Game\_Player

### Tile

The Tile table represents the individual tiles that make up the game board. Each tile is associated with a specific game and can be occupied by only 1 player. Tiles can be of different type or essentially containing an item the player acquire. The player ID and Item ID associated with a tile can be null, meaning that the tile is not occupied by a player or item (empty tile). The row and col values are the position which that tile occupies on the game board.

The Tile table is related to 3 other tables, Game, Player and Item. An Item can belong to many Tiles but a Tile cannot have many items, but only 1 at a time (be of only a single tile type at a time). The Player and Tile tables are linked through the Tile needing to know what player is or is not currently occupying it.

### Item

The Item table stores information about the different items that the players can collect in the game such as berries that increase the player score or thorns which deduct points from the player’s score. Each item has a type (berry, thorn, etc.) and a point value that contributes to the player’s score

Other than the relationship with the Tile table mentioned above, the Item and Inventory Tables and connected through a join table due to the fact that a player will have a different inventory in each game, meaning an Item can belong to many inventory’s and an inventory can have many different items in it.

### Inventory

The inventory table is used to represent a player’s inventory which acts as a container for storing items the player collects during the game. Each player has one inventory per game they play which can also contain multiple items at once.

The player can have many inventories, but an inventory can only belong to one player.

### Item\_Inventory

### Chat Session

The Chat Session table represents a session of chat that is linked with a game. The chat is used for communication between players with each game having a separate chat session.

### Chat

The Chat table stores information related to individual chat messages that are sent by players during the game. The stored information includes a time stamp of when the message was sent and the text message that it contains.

The chat table is related to chat session table with a chat session containing many different messages but a message only belonging to one session. The player can also have many different chat messages, but the message can only belong to one player.

## CRUD Analysis

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### CRUD Process Descriptions

1. **Register a player account**:  
   This process creates a new player account by inserting the player details (ID, username, password, email) into the database.
2. **Log in player**:  
   This process retrieves player details to verify the login credentials, allowing access to the game. Increments login attempts each time and resets once a successful login occurs
3. **Ban player account**:  
   This process restricts a player's access to the game by marking their account as banned and restricting them from logging in, usually through an update to the database by the app administrator.
4. **Lock player account**:  
   This process temporarily disables a player's account, for security or disciplinary reasons, preventing them from logging in. Will automatically trigger if login attempts linked to an account exceeds a preset limit or can be manually triggered by the administrator
5. **Edit player account**:  
   This process allows updates to the player’s account details, such as updating their username, password, or other information. Can only be done by the administrator.
6. **Delete player account**:  
   This process removes the player's account and related data from the database, permanently deleting their records.
7. **Create Game (Places Tiles on a Board)**:  
   This process initializes a new game by generating and placing tiles on a game board, setting up the play environment.
8. **Place item on a Tile**:  
   This process is for manually placing specific items, onto tiles on the board, affecting gameplay dynamics. This process is also automatically run during a new board creation, with each game tile receiving a random item ID
9. **Move an Item (NPC effect)**:  
   This process moves the thorns item on the game board, replacing randomly selected existing items with thorns and vice versa. Gets ran every 5 player moves in a game to create an additional dynamic.
10. **Join Game**:  
    This process allows a player to join an existing game session, associating them with a game and updating its status if not already active.
11. **Kill/Delete Game**:  
    This process terminates an ongoing game session, removing it from the database and ending player participation.
12. **Move player position**:  
    This process updates the player's current position on the game board, based on their movements during gameplay. Players can only move to adjacent tiles to the one they are currently on and cannot move to a tile that is already occupied.
13. **Scoring / Collecting item (Berry)**:  
    This process updates the player's score when they collect a berry item, increasing their score depending on the points associated with the item.
14. **Scoring / Collect item (Thorn)**:  
    This process updates the player's score when they collect a thorn item, deducting points from the player.
15. **Load chat messages**:  
    This process retrieves chat messages from a specific game session, allowing players to view previous communication within the game.
16. **Send chat message**:  
    This process allows players to send chat messages to other players within a game session, contributing to in-game communication and getting saved within a chat session.

## SQL Queries and Procedures

Each table within the database was populated with a very small amount of test data to validate how the database functions and to link it to windows forms application which visually displays the game and interface. There are also some simple queries used to further validate the database creation.

### Login and Registration

To demonstrate the login and registration process and ensure it has a valid connection to the database, I have linked both the registration and login screens in the application. This means users can open the application and register an account where they should get a message back from database starting the user has been created.

Once an account has been created, the user can navigate back to the login screen where they can login to the application using the same details, they just registered an account with. Additionally, the user could log into the game using one of the test login’s that were made during database creation.

Once logged into an account successfully, the user will be redirected to the game lobby window.

### Gameplay

To demonstrate the gameplay form and connection to the database, the user, once logged into an account and on the main lobby window, can click the ‘New Game’ button to create a new game record in the database. If successful a message will be displayed stating that the game was created.

### Administration

To demonstrate the administration aspects and connection to the database, once on the main lobby window, the user can open up the administrator console using the button in the bottom right corner which redirects them to another window.

The admin console then displays a list of both registered accounts and active games. These lists change if new accounts or games are created will further validate the account registration connection by listing any newly created accounts. As far as functionality is concerned, there is no direct administrative features at this stage, only a display of the registered accounts and active games to show the connection to the database records.

The admin console feature is currently accessible by any user but will only be usable by accounts with administrator privileges in the future.

# Milestone Two

The goal of the second milestone within this project is to continue with developing a prototype application, where logic, game management and its database is implemented using TSQL procedures, C# data access objects and a GUI using Windows Forms components. This stage/milestone builds on the CRUD analysis conducted in the previous phase and focuses on developing and implementing core functionality that is required for the game. The system aims to have user management (login, registration), game play management (board layout, item placement, scoring, movement, etc.) and administrative features (adding/editing/deleting user accounts and games).

The database plays an important role in controlling the applications logic with all essential operations being held within a procedure. These TSQL procedures are then invoked from the applications DAO to interact with the database. Windows forms GUI components are then linked to the connected DAO, enabling users to perform various tasks and actions such as logging in, registering and gameplay.

## Implementation

### 1. Player login, including lock out

The **loginUser** procedure is used to validate the user’s credentials and check their status (locked, banned, etc.). After each failed login attempt, the login attempts counter is incremented, and if after a preset threshold is reached, the user account is locked, requiring the application admin to manually reset their account locked\_out status.

The application uses a corresponding DAO method to connect to the database and call the loginUser procedure, returning whether the login was successful or not. The GUI component within windows forms consists of a login form to capture the players, username and password input. When the user submits the login form, the GUI triggers the previously described DAO method. If login is successful, the application allows the user to access the game, otherwise it will display an error message or inform the user the account is locked.

### 2. Player registration

The **registerUser** procedure is used to handle the creation and insert of new user accounts into the tblPlayer table within the database. It ensures that all required fields such as username, email and password are provided and filled out by the user to be accepted.

The registerUser DAO method then encapsulates the connection to the database and called the previously mentioned TSQL procedure. It takes user input as parameters and manages inserting the new player into the system. The GUI component consists of a registration form which collected the user’s data (username, email and password) and once submitted, triggers the registerUser DAO method. Feedback is given in the form of a success/error message if registration fails (account already exists) or if required fields are left empty.

### 3. Laying out tiles on a game board

The **makeBoard** procedure generates the initial layout of the game board by creating and organizing tiles based on a preset board size within the tblGame table. The procedure creates individual and unique tiles depending on the board size specified (10x10 board size will make 100 tiles for that game id). The procedure has an internal call to the getTileType function which returns a tile type (item) to be randomly placed on the tiles during board generation. This gives each game a unique and random placement of items, bringing in a changing dynamic to the application.

The makeBoard DAO method calls the previously mentioned TSQL procedure to initialize the game board. It sets parameters like the number of rows and columns and manages the boards structure in the database. In this early iteration of the game, the user currently has no ability to change the board size within the application/GUI. The GUI sends the preset data when the Create Game button on the Lobby form is clicked, which uses the makeBoard DAO method to generate a new board/game.

### 4. Placing an item on a tile

Due to the makeBoard procedure handling the items/tile types during board generation, a procedure for placing an item on a tile seemed a little irrelevant, although a procedure was created to meet the assessment criteria. The **placeItemOnTile** procedure allows a user to set a specific item on a tile. It takes in a item id and tile id as parameters and simply replaces the existing item on that tile with whichever the user gives in the parameter.

The placeItemOnTile DAO method is responsible for calling the SQL logic that places an item on a specific tile. It handles updates to the game state where items (berries, thorns, null) are placed on tiles. The current iteration of the game has a very basic and probably unnecessary GUI component to implement this TSQL procedure in the way of a button within the game form which triggers the placeItemOnTile method to update the item on a tile.

### 5. Player Movement

The **movePlayer** procedure handles player movement and checks for if the player is able to move to the new tile or not (if adjacent to the player’s existing location). The player can move to any new tile within a 1 tile radius around there current location and will receive an error if trying to move out of bounds or to a tile that is not adjacent. The procedure takes in the game id, player id and new tile location (row and column) as parameters.

The movePlayer DAO method handles updating the players position in the database. It checks if the move is valid and then updates the player’s current tile position in the given game. Since there is no game board GUI in the current application iteration, the games GUI captures player movement using a button which triggers the movePlayer DAO method and if successful, the players position is updated and a success message is displayed, otherwise an error.

### 6. Game play scoring

Gameplay scoring is tied into player’s moving to a tile that has an item on it and acquiring this item into their Inventory, removing it from the tile. The scoring procedure **updatePlayerScore** is internally called within the player movement procedure which gets checked each time a player moves to a tile that is not empty. The updatePlayerScore procedure adjusts the players score based on which item is acquired during gameplay.

The updatePlayerScore DAO method calls this TSQL procedure to update the players score in the database based on actions that occur during the game (collecting berries, running over thorns, etc.). Since the scoring is handled within another procedure, to demonstrate its functionality separately to the rest of the application, the GUI uses a button to update the player’s score in real time. The player and item that is acquired is currently preset in the application to demonstrate this TSQL procedure.

### 7. Player Acquiring Inventory

The player gameplay acquiring inventory functionality is tied into the movePlayer procedure and gets internally called once a player moves onto a tile with an item on it, but for the criteria of the assessment a separate procedure with the name of **acquireItem** was implemented to demonstrate the item acquisition independently to other application procedures. The acquireItem procedure takes in a player id and tile id as parameters and essentially retrieves the item associated with the tile id and adds it to the inventory of the player matching the parameterized player id. The acquired item is then removed from the tile and item id of the tile set to null.

The acquireItem DAO method adds items to that players inventory and updates both the tile which the item was on and the players inventory record in the database. As mentioned previously, the gameplay acquiring inventory is intended to be called within the movePlayer procedure but for the demonstration of the assessment, a button was added to the GUI and when clicked, interacts with the acquireItem method. The item is removed from the preset tile id and added to the preset player’s inventory.

### 8. Move an Item (NPC effect)

The **moveThorns** TSQL procedure is used to move items (NPCs) on a given game board. The logic affects the game dynamics by changing the game board layout and flow after there have been 5 player moves (5 calls of the movePlayer procedure). The procedure follows similar structure to randomly placing items on a tile but instead, retrieves all tiles with a thorn item and replaces it with another item id and then moves the previous thorns to another tile within the board.

The moveThorns DAO method calls the same TSQL procedure to move the items (NPCs) on the board. It updates the position of most but not all items on the board. The GUI component for this is reflected as a button, and when clicked, the thorns for a predefined game id are moved around the board to a random location. In the final implementation this procedure will likely be called during the movePlayer procedure to streamline efficiency and keep the game running smoothly.

### 9. Kill running games

The **deleteGame** procedure is used to kill a game whether it be active or inactive and removes it from the tblGame table within the database. While this can be seen as not ‘killing’ a running game but deleting, I thought it fit the structure of the application better, as I want games to either be joinable or finished (ready for deletion).

The deleteGame DAO method handles removing the selected game record from the database, effectively ending the game and deleting any associated data from the database. The GUI component for this can be seen in the admin console form and shown as a ‘Delete’ button which removes the selected row from the games list box when clicked. A confirmation message is displayed and needs to be accepted for finally deleting the record.

### 10. Add new players

The **addUser** procedure follows almost identical structure and SQL code as the registerUser procedure. The difference between the two is one is for the user of the application and one is for the administrator. The addUser procedure inserts a new player/user account into the tblPlayer table, adding relevant details such as username, email and password.

The addUser DAO method calls the procedure to add a new user account to the system. The GUI within the admin console provides an interface for admins to add new players. The TSQL procedure is then called once the new user account details have been filled out and submitted.

### 11. Update data of a player

The **updatePlayerProfile** procedure involves running an UPDATE query to modify an existing user account’s details in the tblPlayer table of the database. The procedure is used for admins of the application to edit user information without having to register a new account.

The updatePlayerProfile DAO method updates any changed player details in the database, such as username, password and whether their account is banned or locked. The GUI provides the user a form for editing account details and when submitted, the updatePlayerProfile procedure is triggered to apply the new changes to the database record.

### 12. Delete a player

The **deletePlayer** procedure follows identical structure and SQL to the deleteGame procedure. The difference being the deletePlayer procedure takes in a player id parameter and requires the admin has an existing user selected in the players list box of the admin console.

The deletePlayer DAO method calls the necessary SQL to remove the selected player from the database. The GUI component allows administrators to select a record from the players list box which was populated from all entries in the tblPlayer database table. Once the delete button is clicked, the method is called and the user is removed from the system.

# Milestone Three

## C# .Net application Exception Handling

In this project, exception handling or error handling has been added to the GUI and DAO layers, specifically refactoring each method to include error checking conditions in the form of try / catch blocks. This includes validation checks for inputs and exception handling using the mentioned try / catch blocks to manage unexpected errors in a clean way. The error handling is designed to prevent invalid data from being processed, ensure the application provides feedback to the user in case of errors, and to protect the application from crashing due to unhandled errors.

### GUI Methods

In the Windows Forms application, data from methods that call TSQL stored procedures is used to check error conditions and provide appropriate feedback to the user. Each method captures the result returned by the stored procedure and evaluates it to determine the outcome. For example, the addUser method processes return values such as transaction committed for successful user registration, and transaction rolled back for failed operations, meaning the application will display messages depending on the result returned. A try/catch block is used to ensure that any unforeseen exceptions are caught, logged, and displayed to the user.

For each GUI method, exception handling has been implemented to address any potential user input or display issues from the database connection, with each method now including the following:

* All GUI methods have been surrounded with try / catch blocks to catch and handle any errors that occur during the loading, displaying and/or use of GUI components.
* Data and results returned from TSQL stored procedures is processed within the ‘try’ block to determine what message to display to the user (success or failure).
* Standard error messaging has been used in the catch block to display any unseen errors that may occur.

Below are screenshots and descriptions of each GUI method that involves the implemented exception handling feature mentioned above and a test of the error handling in operation:

**Register user**

A computer screen shot of text

Description automatically generated

The register user GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Account registered successfully
* **If transaction rolls back:** Failed to register account

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

A screenshot of a computer error

Description automatically generated

**Login user**

A screen shot of a computer program

Description automatically generated

The login user GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Logged in successfully
* **If transaction rolls back:** Failed to log in

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

A screenshot of a computer error

Description automatically generated

**Make board**

A computer screen shot of text

Description automatically generated

The make board GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** New game created successfully
* **If transaction rolls back:** Failed to create a new game

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

A screen shot of a computer screen

Description automatically generated

**Placing item on a tile**

A screen shot of a computer code

Description automatically generated

The place item on a tile GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Item place successfully on tile:
* **If transaction rolls back:** Failed to place item

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

A screenshot of a computer error

Description automatically generated

**Player game movement**

A screen shot of a computer program

Description automatically generated

The move player GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Player moved successfully to tile:
* **If transaction rolls back:** Failed to move player

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

A screenshot of a computer error message

Description automatically generated

**Player game scoring**

A screen shot of a computer program

Description automatically generated

The update player score GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** player score updated successfully
* **If transaction rolls back:** Failed to update score:

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

**Item acquiring**

A screen shot of a computer code

Description automatically generated

The acquire item GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Item added to inventory
* **If transaction rolls back:** Failed to acquire item

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

**Moving an item (NPC effect)**

A computer screen shot of text

Description automatically generated

The move thorns GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Thorns (NPC) moved successfully
* **If transaction rolls back:** Failed to move the thorns

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

**Kill/Delete existing games**

A computer screen shot of text

Description automatically generated

The delete game GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

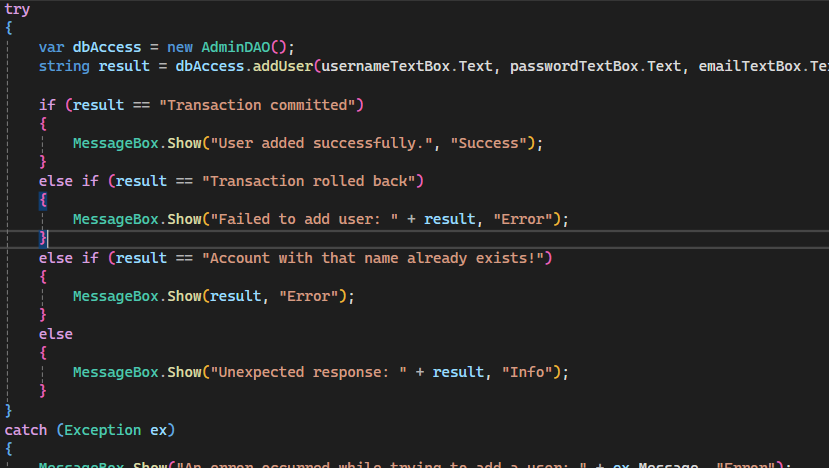
* **If transaction is committed:** Game deleted successfully
* **If transaction rolls back:** Failed to delete game

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error message

Description automatically generated

**Add new users**



The add user GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** User added successfully
* **If transaction rolls back:** Failed to add user

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

**Update user account data**

A screen shot of a computer program

Description automatically generated

The update player profile GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** Profile updated successfully
* **If transaction rolls back:** Failed to update user profile

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.

A screenshot of a computer error

Description automatically generated

**Delete a user account**

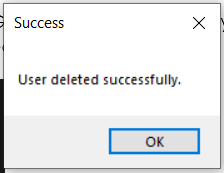
A screen shot of a computer program

Description automatically generated

The delete player GUI method uses a try/catch block, with the try block running a series of if statements to process results/messages returned from the stored procedures. This method provides feedback to the user via a message depending on the resulted data from methods that call TSQL procedures with the following:

* **If transaction is committed:** User deleted successfully
* **If transaction rolls back:** Failed to delete user:

The exception handling and error messages were tested multiple times by running the DAO method via the GUI component with both valid and invalid parameters to test how the procedure handled different inputs and what feedback it would provide the user.



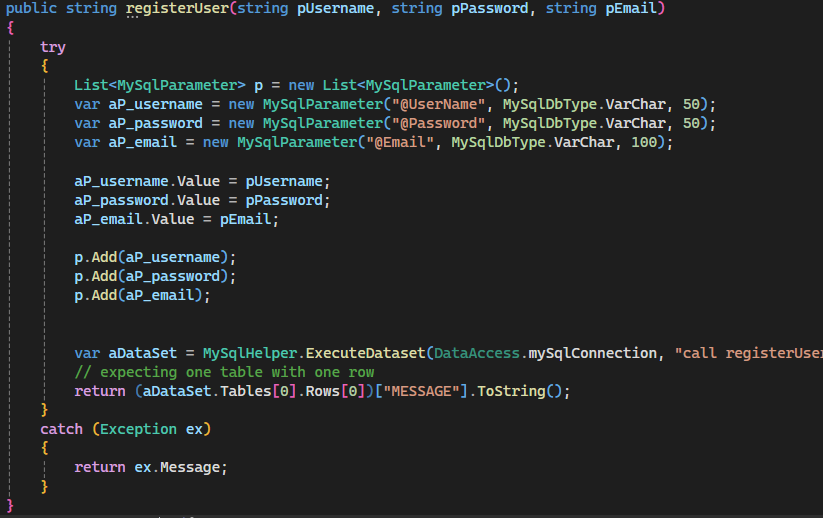
### DAO Methods

For each DAO method, exception handling has been implemented to address any potential database connection issues, with each method now including the following:

* All database methods have been surrounded with try / catch blocks to catch and handle any ‘SQLExceptions’ or other errors that can occur during database connection.
* Error messaging has been implemented in case there is an issue, the application will inform the user that a database error has occurred and will advise them to try again.
* Exception details are logged for troubleshooting and debugging of any errors that occur.

Below are screenshots of each DAO method with the implementation of the exception handling features mentioned above:

**Register user**



**Login user**

A computer screen shot of a program code

Description automatically generated

**Make board**

A computer screen shot of a program code

Description automatically generated

**Placing item on a tile**

A computer screen shot of a program code

Description automatically generated

**Player game movement**

A screen shot of a computer program

Description automatically generated

**Player game scoring**

A computer screen shot of a program code

Description automatically generated

**Item acquiring**

A computer screen shot of a program code

Description automatically generated

**Moving an item (NPC effect)**

A screen shot of a computer program

Description automatically generated

**Kill/Delete existing games**

A computer screen shot of a program code

Description automatically generated

**Add new users**

A computer screen shot of a program code

Description automatically generated

**Update user account data**

A computer screen shot of a program code

Description automatically generated

**Delete a user account**

A screen shot of a computer program

Description automatically generated

## Concurrency Management

Concurrency management is very important in database applications that require multiuser-ness or processes that need to modify or access data at the same time. Concurrency helps to prevent inconsistencies in the data and other issues that can happen due to concurrent operations. In terms of the project application, concurrency is important for things such as the multiplayer gameplay aspect and chat sessions feature to provide consistent data retrieval and the possibility of having multiple people playing the same game at once.

### MySQL Concurrency

In MySQL, concurrency is managed and handled through transactions, with there being different isolation levels of data visibility to balance the consistency and performance. These levels include read uncommitted, read committed, repeatable read and serializable, with the default being a repeatable read level of isolation (*MySQL :: MySQL 8.4 Reference Manual :: 17.7.2.1 Transaction Isolation Levels*, n.d.). Repeatable read ensures data that is read within a transaction remains consistent throughout that transaction and prevents dirty or non-repeatable reads from occurring

MySQL uses optimistic concurrency through the previously mentioned isolation levels so any conflicts are addressed by retrying transactions without locking data. The use of optimistic concurrency seems to be work well in environments with minimal conflict risk which allows for higher concurrency with little overhead (Kumar, 2024). Pessimistic concurrency on that other hand, actively looks for data and is not supported within MySQL’s isolation levels/modes, meaning developers are required to manually implement locking for situations that need stricter consistency control.

Stored procedures in MySQL can use specific isolation levels to manage concurrency based on the procedure’s needs and requirements. For example a stored procedure could operate using the ‘Repeatable Read’ level, ensuring that all data read during that transaction remains consistent, which reduces the likelihood for non-repeatable reads.

### C# .NET Concurrency

In C#/.NET, concurrency management is also done through using optimistic and pessimistic methods, specifically when using LINQ and entity frameworks. Each approach can serve different purposes and can be used in various situations to achieve different requirements.

Using the approach of optimistic concurrency via LINQ and entity framework avoids locking records when they’re read and instead checks for conflicts during an update phase. In entity framework, concurrency is achieved by using a ‘ConcurrenyToken’ attribute to rack any changes made to records in the database. Before updating records, entity framework compares that token with the database’s current value and raising an exception if a conflict is identified. This approach can be efficient in systems/applications where there are multiple users which need to read the same data simultaneously, which minimizes any locking overhead and increasing performance slightly.

LINQ concurrency management with direct connections to SQL typically supports this optimistic concurrency approach and relying on tokens such as time stamps or row versions to detect changes before they’re committed. If a conflict is detected, such as when data has changed since it was last read, an exception is raised, prompting the application to handle that conflict (Cmastr, 2021). Similarly to MySQL concurrency, LINQ does not natively support pessimistic concurrency and requires customization to locking using stored procedures.

### Comparison

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **C# .NET (LINQ/Entity Framework)** |
| **Primary Approach** | Transaction isolation modes/levels | Optimistic and pessimistic concurrency |
| **Optimistic concurrency** | Implicitly managed through different isolation levels | Supported using concurrency tokens and row versioning |
| **Pessimistic concurrency** | Not natively support, requires manually development | Support in the Entity Framework with explicit locking |
| **Isolation levels** | Isolation levels are defined for managing concurrency | N/A (Managed via tokens) |
| **Conflict management** | Managed by retrying transactions or by using manual locking | Managed automatically through concurrency exception handling |
| **Use cases** | Applications with minimal conflict risk and high performance | High contention Applications that require data integrity |

To conclude, the choice and use of concurrency management depends on the applications requirements, with MySQL being flexible due to isolation levels and typically suiting multiuser applications, while C# .NET’s concurrency being ideal in environments that require more control over the data integrity and conflict management.

# References

Cmastr. (2021, September 15). *Optimistic concurrency: Overview - ADO.NET*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/optimistic-concurrency-overview>

*MySQL :: MySQL 8.4 Reference Manual :: 17.7.2.1 Transaction Isolation Levels*. (n.d.). <https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html>

Kumar, S. (2024, June 11). Optimistic locking in MySQL - Saurabh Kumar - Medium. *Medium*. <https://medium.com/@saurabhk1593/optimistic-locking-in-mysql-97abf4c07783>