OCTOBER 18, 2024

FULL STACK GAME DEVELOPMENT

DAT602 ASSESSMENT

KIRA BYRNE
#13509995
NMIT

Table of Contents

INTRODUCTION

The Plot

DUST 2 DUST is a turn-based, grid-based, multiplayer fighting game set in the later age of the American frontier. The year is 1898 in the infamous town of Tombstone, Arizona. Each player is a gunslinger, fighting their way through law and lawlessness in the establishing wild west with only a Colt 45 revolver at their side. The player must defend themself through one-on-one shootouts with other players, collecting bullets around the map between fights where they can, ensuring they are equipped to handle the next fight before they are challenged!

Gameplay

Account & Menu

The player will make an account with an email, username, and password. They can log in with either the email address or username but the password will validate either.

The player can now log into their account where they can enter the game, read a tutorial that tells them the plot of the game and how to play, seek admin support which will open a mailto: link to the admin email (?), or quit the game.

The Player Character

Each player character is made equal upon starting a game. They have a revolver, 3 bullets, and 10 health points.

- The revolver isn't necessarily an item as listed below but a simple representation of a tool to use the bullets. It is only a name, nothing 'physical' within the game.
- A player is able to move around the map, tile by tile, and interact with the world upon clicking when in proximity to a tile of interest.
- As a player character survives the environment, they will score points. 30 points are given for every 30 seconds a player survives in the game.

Map

Upon starting a new game, the player will spawn into a 9x9 map on a home tile. They can move either by clicking the next tile (NSEW) or key stroking using the UDLR keys. The player will move around the map rather than the map moving around the player.

The map in itself will be a vector of tiles. 9x9 for each tiled map that is visible to the player, and 3x3 for the wider map the player can navigate upon reaching the edge of the map they currently reside on. Each tile will act as a coordinate identified by the primary key of each major tile of the map (i.e. the 3x3 of tiles).

NPCs & Items

Within each map will randomly spawn NPCs and items. Other players can login and wander around the map to find these items and NPCs.

Items the player can find:

- Bullets, needed for doing damage in duels.
- Whiskey, for replenishing the player's health score.

Scoring points

Players can earn points through different methods:

- The initial method is by staying in the game as long as they can without dying. For every 15 seconds a player is exploring the map, 10 points are added to the score.
- Players can earn extra points through finding items like bullets and whiskey.
- Time-based points stop accumulating during duels, however, winning a duel will allot the player 1000 points.

Dueling

Players can challenge each other to duels. When a player reaches up to 2 tiles away from another player will be able to left-click an enemy player to target them and an attack button to fire a bullet.

When a player fires a bullet (attacks), a cooldown debuff will apply which is a short window of time where the attacking player cannot attack again for 1.5 seconds.

- A timestamp of how long they survived in the game
- The final score earned before they died
- Their last high score if exists (select score where TOP timestamp ascending)

CRUD Analysis

DUST 2 DUST CRUD ANALYSIS

	TABLE																												
	ACCOUNT								CHARACTER																INVENTORY				ITEM
PROCESS	Username	First name	Email	Pessword	Login attempt	Status	Account type	Attempts	CharacterID	Player username	CharacterName	GameID	Health	Current score	High score	Status	Position	Invincibility	lastAttack	AttackCooldown	lastMove	atk	CharacterID	ItemID	ItemName	Quantity	ItemID
Register account	C	C	C	C			C																				
Login a player					R, U	U		R, U																			
Edit a player	U	U	U	U			U																				
Remove a player						U																					
Ban a player	D	D	D	D		U																					
Lougout						U																					
Lockout	R			R	R,U	U			C	R	C					U											
Make a character	R								R	R	R		R,U	C, U	U,R		R U										
Enter a game	R								R			R U	U	U	R	U	R U										
Exit a game									R			D	U	R	R U	U	R D										
Place on home tile														C,U	R U		U										
Accumulate score														U	R U												
Move tile																	R U				U						
Collect item																							R	R	R	C, U	
Use item																					U			R	R	U	
Gain health													U														
Deal damage															U				U	U	U						
Lose health													U					U									
Die												D	U	R	R U	U	D		R								
Check last attack																					R						
Check last movement																											
Mark AFK (away from keyboard)						U						D				U						U					
Attack cool down																				U,D							
invincible window																		U,D									
Return score														R U	R U												
Start a new game																											
Draw map																											
Draw grid																											
Set tiles																											
Set/Remove item on tile																											R
Set/Remove NPC on tile																											
Player moves to tile																											
See item name																											R
See item details																											
Kill game																											

Storyboards

1.0 Main menu

The first screen seen upon running the application client.

Functions

1.1 Should display the game title and three buttons.
1.2 Login into account
1.3 Create account
1.4 Exit application (closes window)

2.0 Account Sign-up

The account creation screen that allows players to enter criteria and create an account for gameplay.
Functions

Accessed through the 'sign up' button on the main menu.

2.1 – 2.5 Criteria requires:

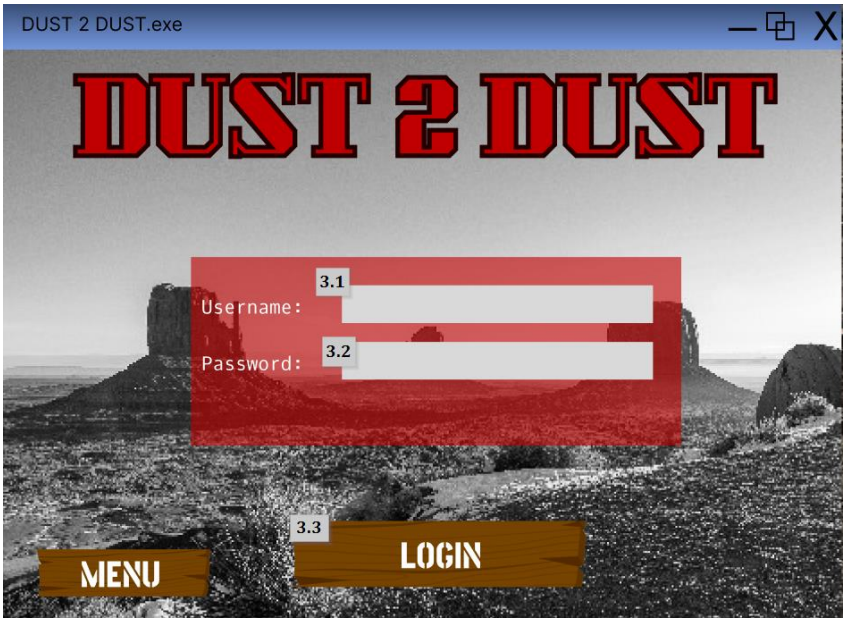
- Unique email
- Unique username
- Password

2.6 If the username the player is attempting to claim is already stored in the database, it will return this error.

2.7 If the player attempts to use an email address that is already stored in the database, it will return this error.

2.8 If the entered details meet the criteria requirements, a 'success' message will appear.

2.9 On clicking the 'sign up' button, all account details are submitted and stored in the database account page.



3.0 Login

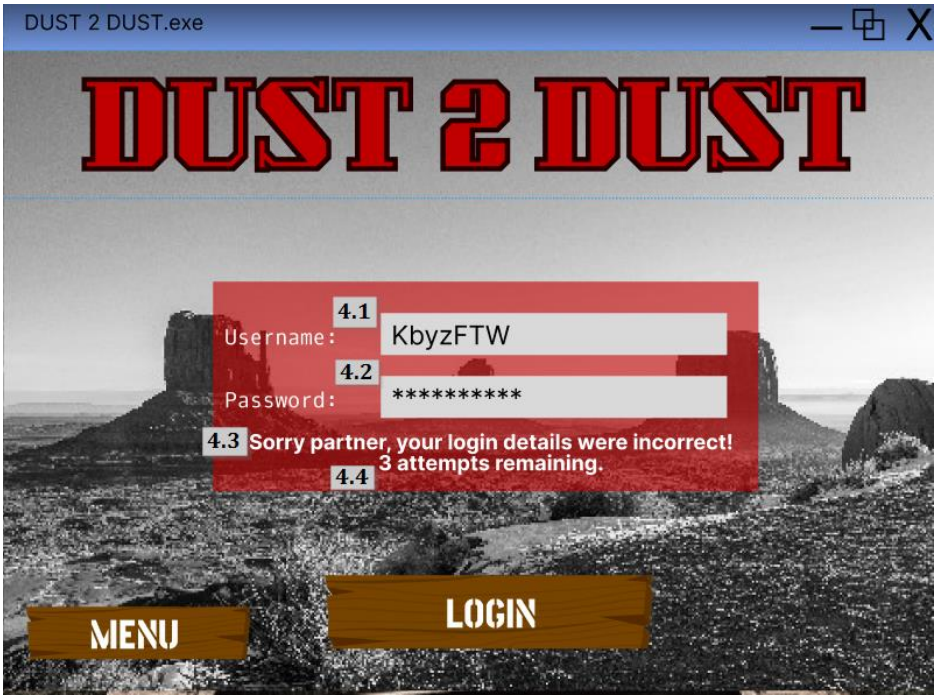
Allows players with a account stored in the database will be able to log into with their username and password.

Functions

3.1 Player will enter their username in the provided text box. This criteria will be checked against the list of usernames stored in the user accounts table in the database

3.2 Player will enter their corresponding password in the provided text box. This criteria will be checked against the password corresponding to the provided username stored in the user accounts table in the database. The password cannot work without a valid username.

3.3 When details are entered, the player can click the login button to attempt to login to their account. A login attempt will check the provided details against what is stored in the database.



4.0 Failed Login

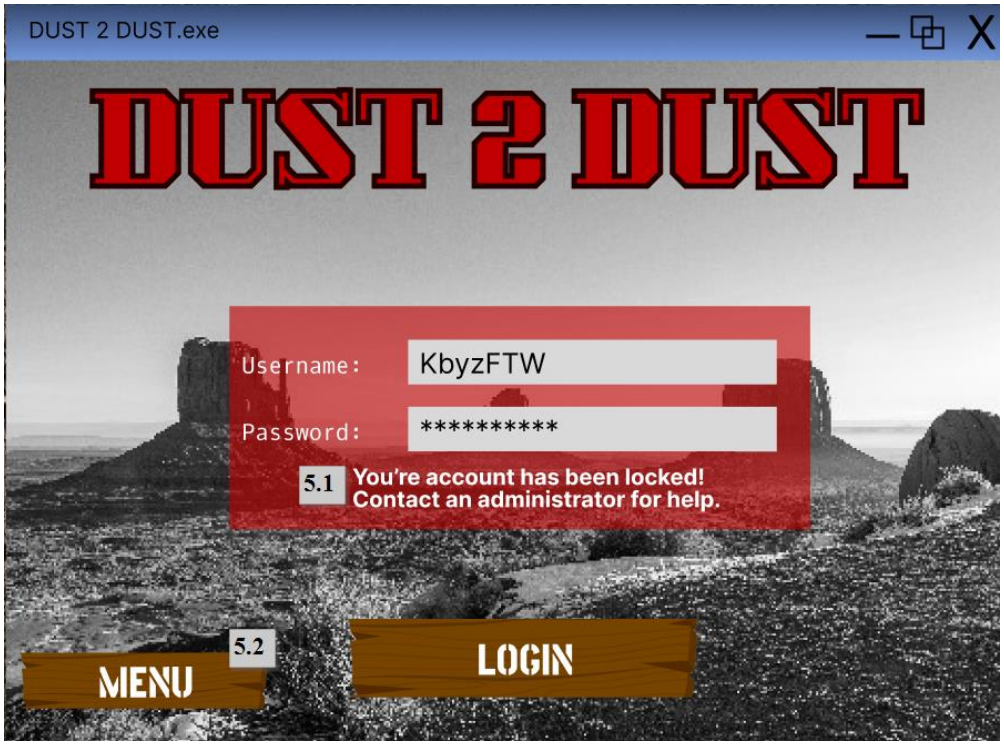
If a player attempts to login with invalid details, the login menu will produce an error message.
Functions

4.1 If the database has not stored the username provided in the textbox, it will return a login error regardless of the password (correct or incorrect).

4.2 If the database has not stored the corresponding password in the database, it will return a login error.

4.3 When a login attempt fails with the criteria in either 4.1 or 4.2, an error message will appear to notify the player of their failed attempt.

4.4 The database will store a number of login attempts, the maximum being 4 tries. Upon the next failed login attempt, the database will update the error message to display the number of remaining attempts.



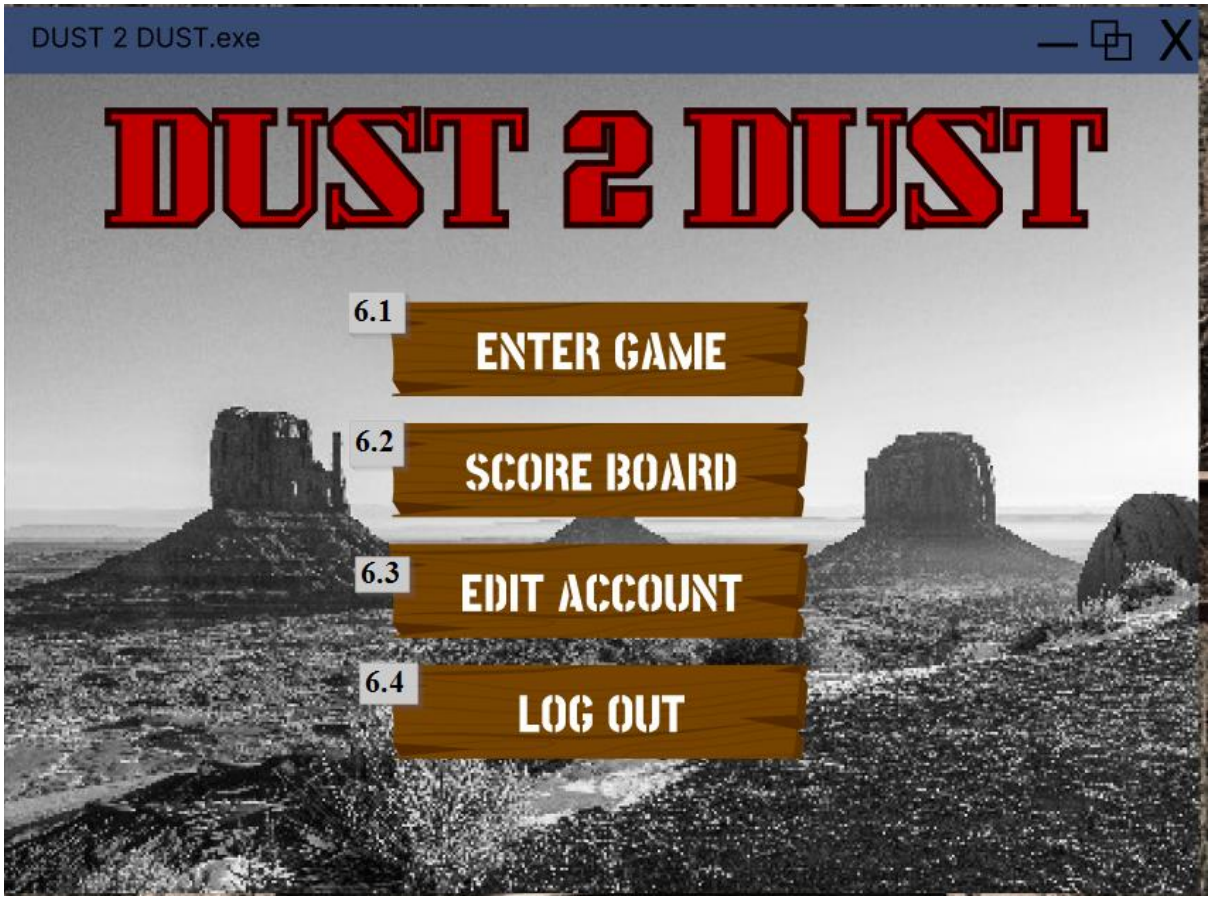
5.0 Account Locked

When a player has exceeded 4 login attempts with invalid details, their account will be locked until an administrator is contacted via the 'help' feature on the main menu.

Functions

5.1 After the fourth and final login attempt with invalid criteria, the error message will update to notify the player of their account being locked. The player will be encouraged to contact an administrator for help.

5.2 The player will be guided back to the menu via the main menu button where the 'help' option can be found.



6.0 Account Menu (logged in)

When a player successfully logs into their account, they will be taken to the account menu.

Functions

6.1 A player can enter a game. This will either create a game if there is no active players or return an active gameID to allocate the player to. All characters are created equally upon entering the game and will be identified through the account username.

6.2 View their scoreboard. This will show an aggregate of each player's highest score.

6.3 A player can edit their account details

6.4 Logout of their account. This will return the player back to the main menu as shown in fig 1.0.

7.0 Edit Account

A player can edit details in their account which will update any of the changed criteria. Functions. Accessed through the ‘edit account’’ button on the logged-in menu as seen in fig 6.

Criteria requires:

7.1 Unique username

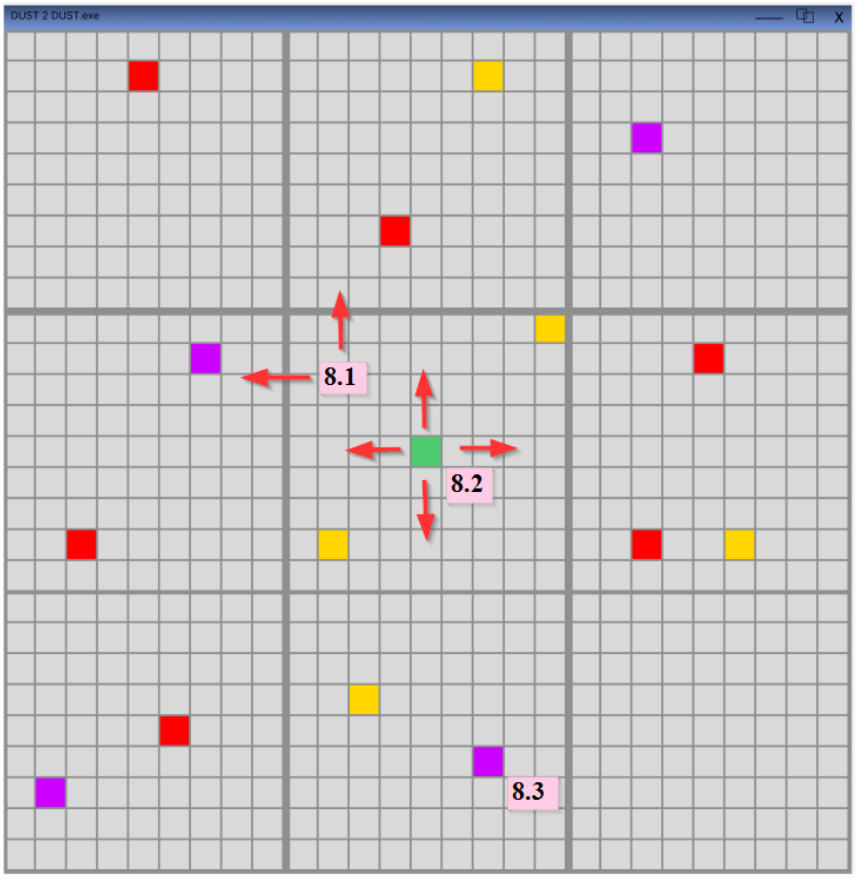
7.2 Old password to confirm the account holder is valid

7.4 New password

7.5 Confirmation of new password

Players cannot change their registered email autonomously. This will be handled through a request email to the administrator.

On clicking the ‘save’ button, all edited account details are checked for the above criteria in the database. If criteria are met, the new details are saved.



8.0 Whole Game Map - 3x3 Vector Grid

This is a visual of the entire map that exists in a game. The entire map (NOT seen by the player) is a 3x3 grid of 9x9 grids. When a player is active, they will only view the 9x9 grid they are actively on.

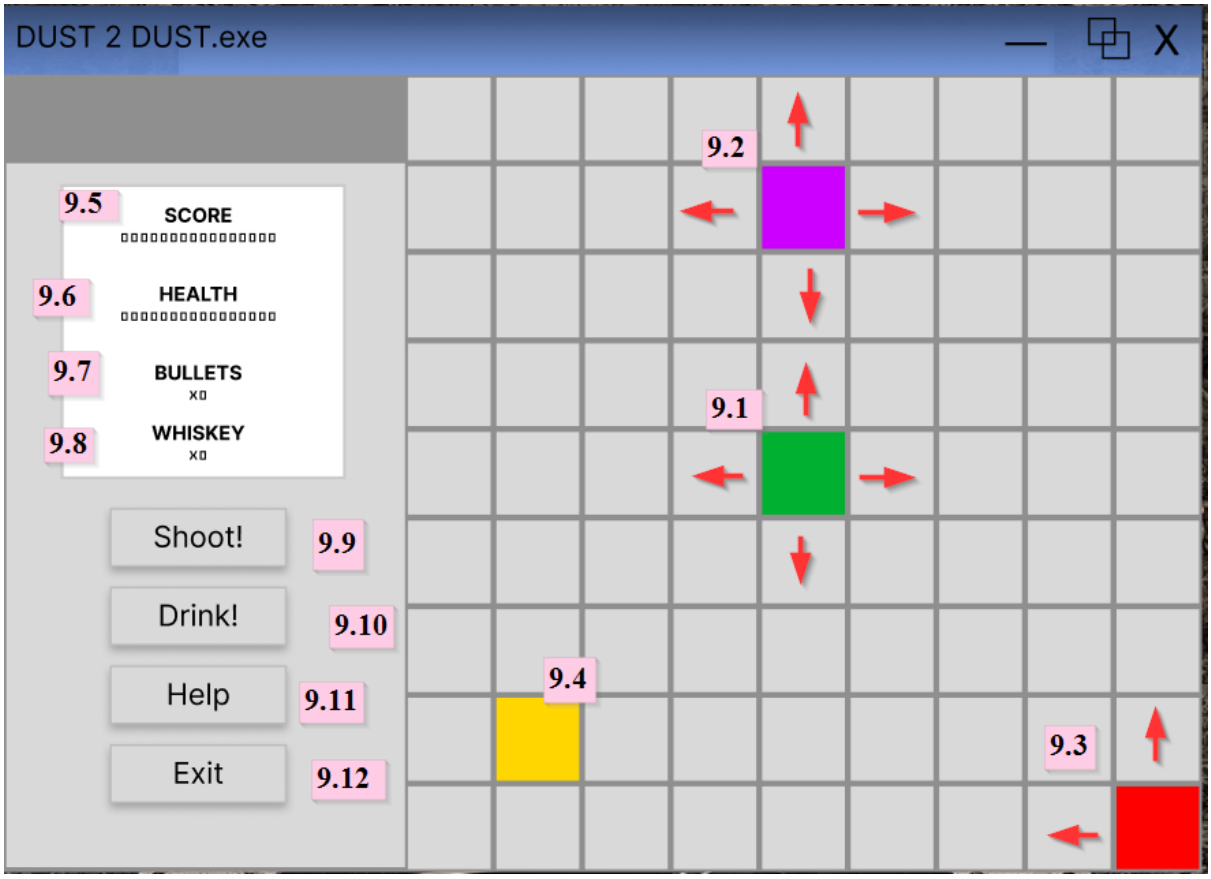
A game will have active assets across all grids. It will be programmed to randomly spawn and de-spawn items and NPCs after players have interacted with them of after allotted amounts of time.

Functions

8.1 A game exists within the bounds of the 3x3 map.

8.2 A player will be able to view and navigate one 9x9 grid within the map. The tile coordinates will be identified by each gridID within the map i.e.
gridID: ‘west2’ tileID: “-1.0, 0.2”

8.3 A new game will generate items on commencement All items and NPC will load across the map at random intervals. All players in a game can see all items, NPCs and other players.



9.0 Game Screen – 9x9 grid

As mentioned in fig 8.0, the map is a 3x3 vector of 9x9 grids. Seen in above (fig 9.0) the 9x9 gird represents the part of the 3x3 map visible to the player which they will be able to navigate to find the NPCs, items, and enemy players as shown in the figure.

Functions

9.1 A player will spawn on a random empty tile (coordinate) in a random grid (gridID) in the map. They will be able to navigate on the X and Y axis using the keyboard keys up, down, left, and right as represented by the red arrows. Upon reaching a boarder, they will move to a near by grid if available. If not, they will remain still.

9.2 When a game is created, Non-player Characters (NPCs) will spawn throughout the map which are visible to all players. NPCs are able to move via the database updating their location every few moments.

9.3 Enemy players are represented as red tiles and have all the same functions as the player does.

9.4 Items, represented by yellow tiles, will randomly appear on the map to be interacted with by any player.

9.5 – 9.8 A side bar on the game screen will display the players current stats and items including:

> 9.5 Their current score. The score will allot 100 points for every 30 seconds the character spends in the game. The primary goal is to survive as long as possible.
>
> 9.6 Their current health. Each character is given full health displayed as 10/10 ‘points’. When a player’s health reaches 0, they will die.

9.7 The amount of bullets they have collected. All characters will start with 6 bullets but this number is limitless.

9.8 The amount of whiskey they have collected. Whiskey can be used to restore health points. Players can find whiskey throughout the game from item tiles or NPCs. Players will begin a game with no whiskey stored.

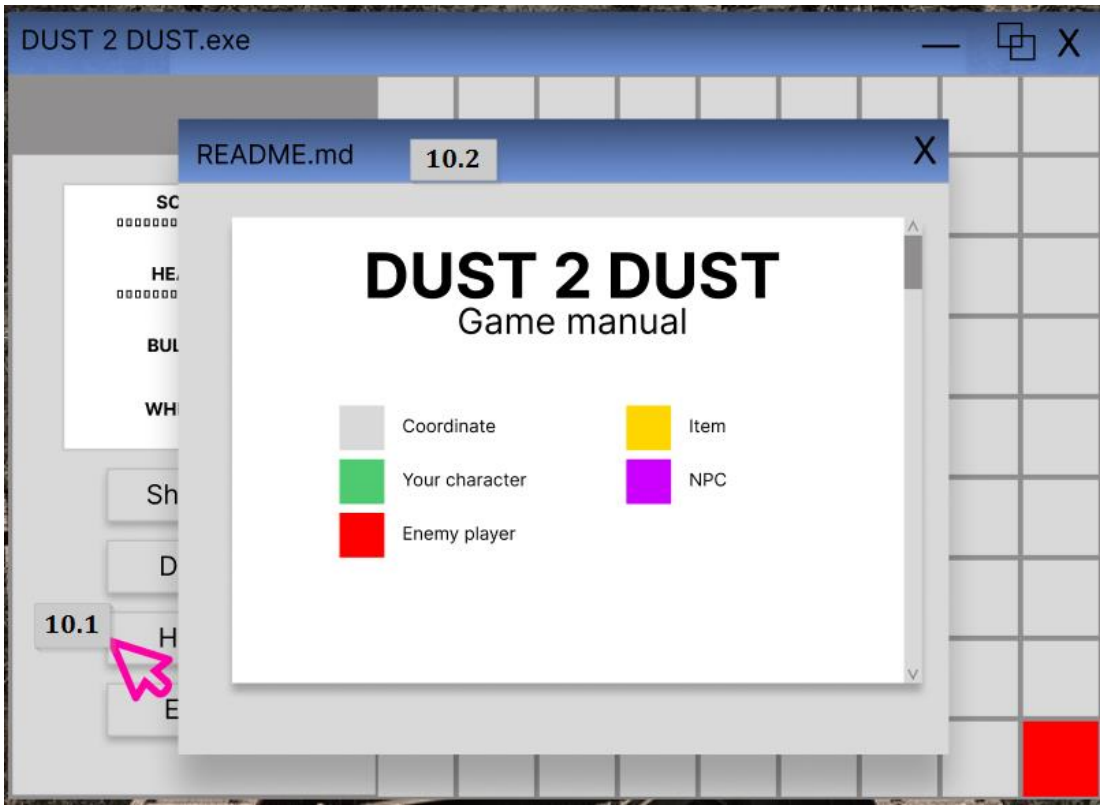
9.9 – 9.12 The player can interact with game mechanics through a collection of buttons on the side bar including:

9.9 The 'shoot!' button will allow them to attack an enemy player that is targeted through a left-click event. Clicking this button will retrieve and update the quantity of bullets the player has. The button can only function when bullet quantity is >=1.

9.10 The 'drink!' button allows a player to use a bottle of whiskey to restore health. Clicking this button will return and update the quantity of whiskey the player has. The button can only function when whiskey quantity is >=1 and health points are <10.

9.11 The player can access the game manual from the 'help' button.

9.12 The player can exit the game. This will remove their character from the active game and return them to the account menu.



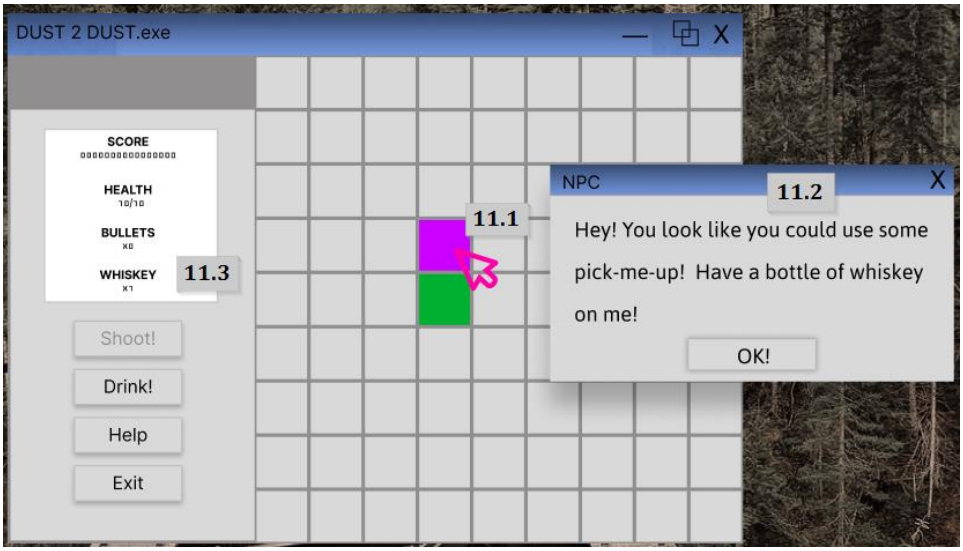
10. Game Help Menu

The player can access the player guide/game manual via the help button to read information about the game. This will not pause the game but open an independent window.

Functions

10.1 The help window is accessed via a left click on the ‘help’ button in the left hand side bar.

10.2 The help window will open a rich text file where the game guide can be viewed.



11. NPC Interaction

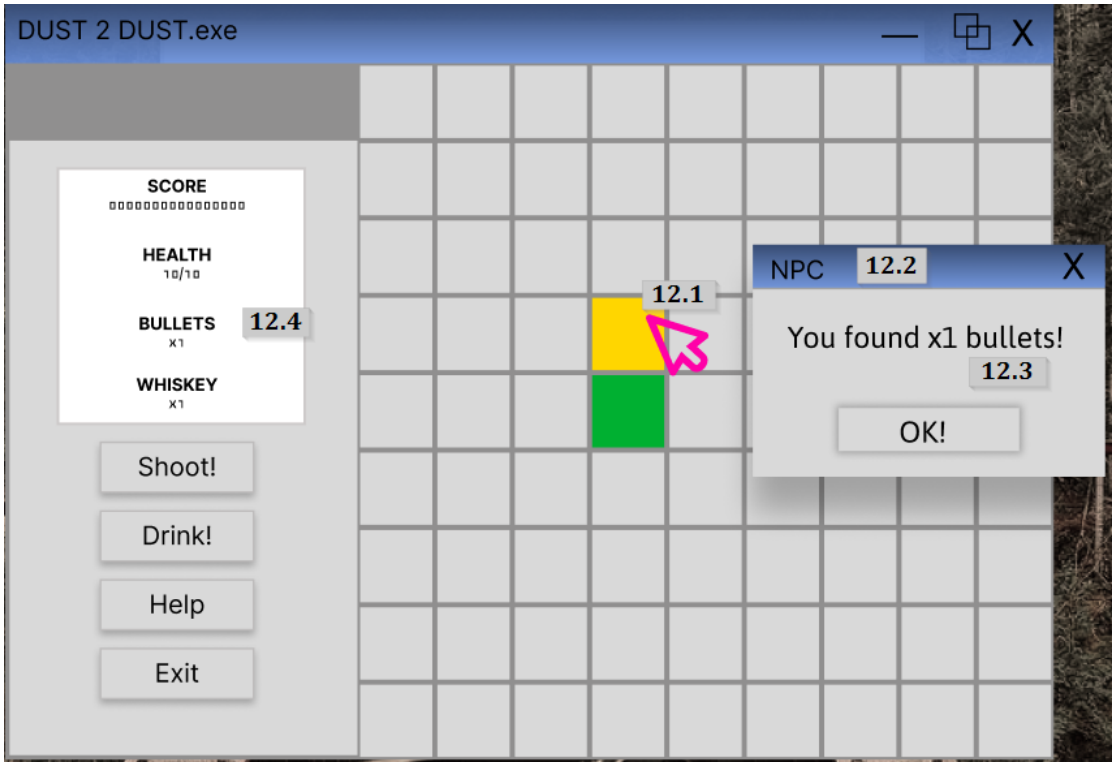
Upon approaching an NPC by standing on a bordering tile, a player can left-click the NPC to interact with them. This will open a dialogue window where the NPC may “say” something to the player or give them an item on random chance. After interacting with a player, the NPC will ‘de-spawn’, leaving their tile empty.

Functions

11.1 Once a player is standing on a tile that borders an NPC tile (purple), they can left-click the tile to interact with it.

11.2 Interacting with an NPC will open a window with dialogue text. The dialogue will be randomised and may have the chance to give the player an item.

11.3 Item given by NPCs will show in the player’s inventory.



12. Item Interaction

When a player is bordering an item tile (yellow) they can left-click to interact with it. This will open a window that will show text regarding what was found and how much. A random quantity and item name from the database. Once a player has interacted with an item tile, it will add the item(s) to their inventory and 'de-spawn' becoming an empty tile.

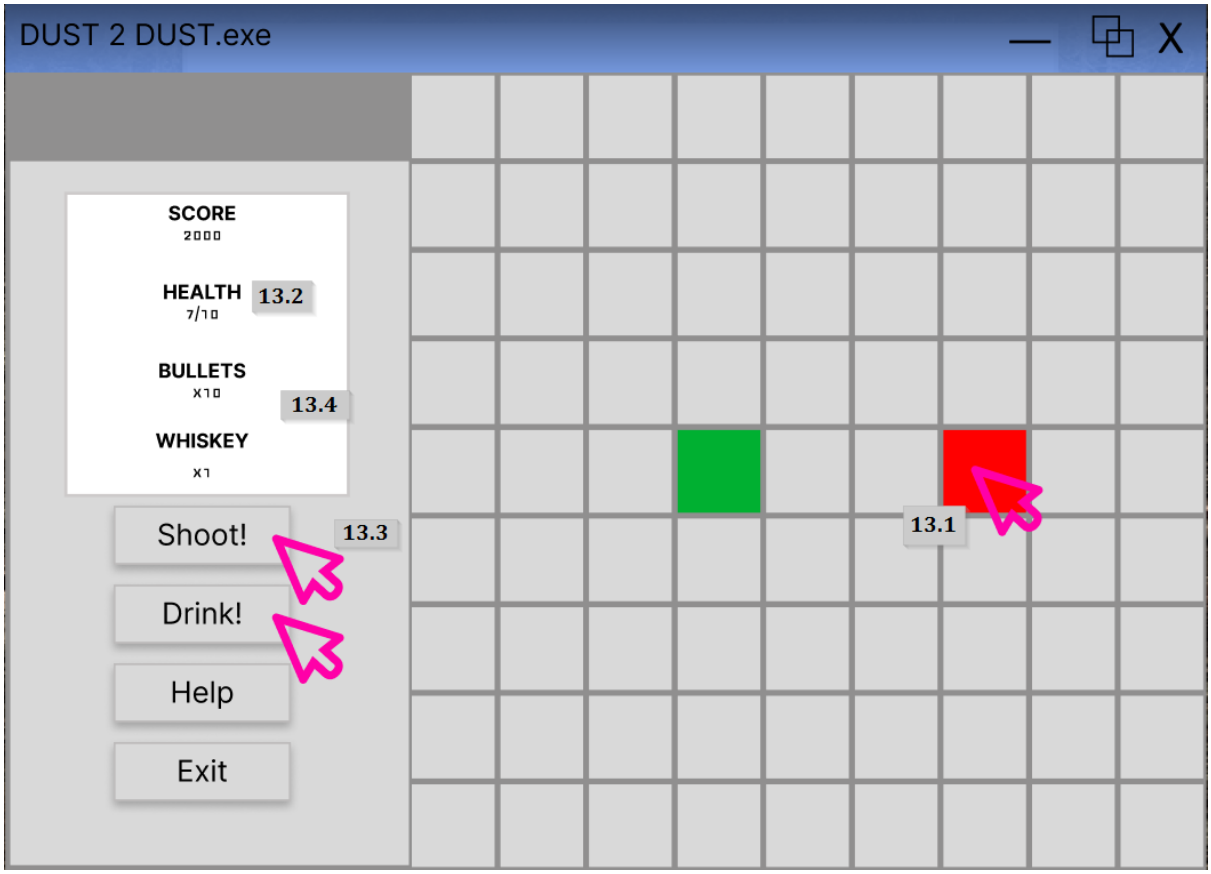
Functions

12.1 The player interacts with an item tile by bordering it then left-clicking it.

12.2 Upon interaction, a pop-up window appears with text about what the player has found.

12.3 The game will pull an item name and randomise a quantity to allot to the player.

12.4 The player's inventory will update the quantity of the item found joining the item data from the interaction window to the inventory table.



13. Dueling Enemy Players (Combat)

Other players will be considered enemies and can engage in combat. A player will be able to 'kill' another player when they at least two tiles away from the player's position.

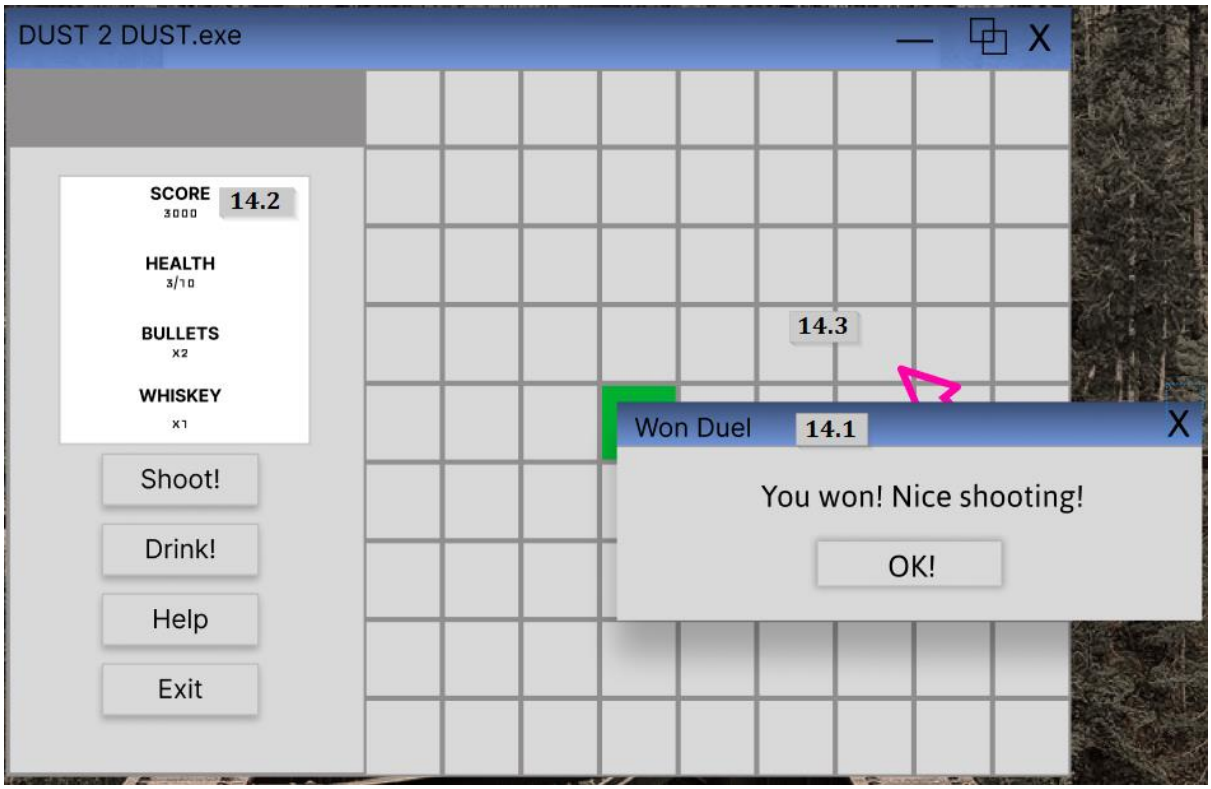
Functions

13.1 Select an enemy player (=< 2 tiles distance) with a left-click.

13.2 Taking damage will decrease the total health stored in the character table.

13.3 The player can deal damage with the 'shoot!' button or restore health with the 'drink!' button

13.4 Utilising the damage or health buttons will update the quantities of their respective items in the player inventory. Upon reaching 0, the buttons will become ineffective.



14. Combat Won

If the player is able to reduce the enemy player's health to 0/10, the enemy player will 'die', resulting in a won duel for the player.

Functions

14.1 Return window displaying 'win' message
14.2 Allot 1000 points to the winning player
14.3 Remove the enemy player from the game.

15. Combat Lost (Death)

A player loses a duel when their health is reduced to 0 through combat (enemy bullets). They will be shown a ‘death’ window and removed from the game.

Functions

15.1 Remove the player from the game, returning them to the account menu.

15.2 Display the ‘death’ window.

15.3 Return the player’s score on death of the last game and their highest score previously earned.

16. Scoreboard (Public)

The public scoreboard is an ascending list of the top 10 players with the highest scores. Players can access the public scoreboard via the ‘scoreboard’ button on the account menu.

16.1 The scoreboard joins the username to their highest score.

16.2 The scoreboard list the top 10 highest scores ascending.

17. Administrator Account Menu

Administrator accounts have a administration level functions displayed in their logged-in menu.

17.1 The administrator has access to the administrator settings which will return live-game data in an interactive window.

18. Administrative Settings

An administrative settings window allows administrative accounts to see all active game activity including active games by ID and players in each game by username. Administrative accounts have the ability to kill active games, remove players from a game, or ban a player.

18.1 The administrative window opens via the ‘admin settings’ button in the administration account login menu as seen in fig 17.

18.2 All active gameIDs are listed in the active games bar. The admin can select a game via a left-click to access more details or manage that game.

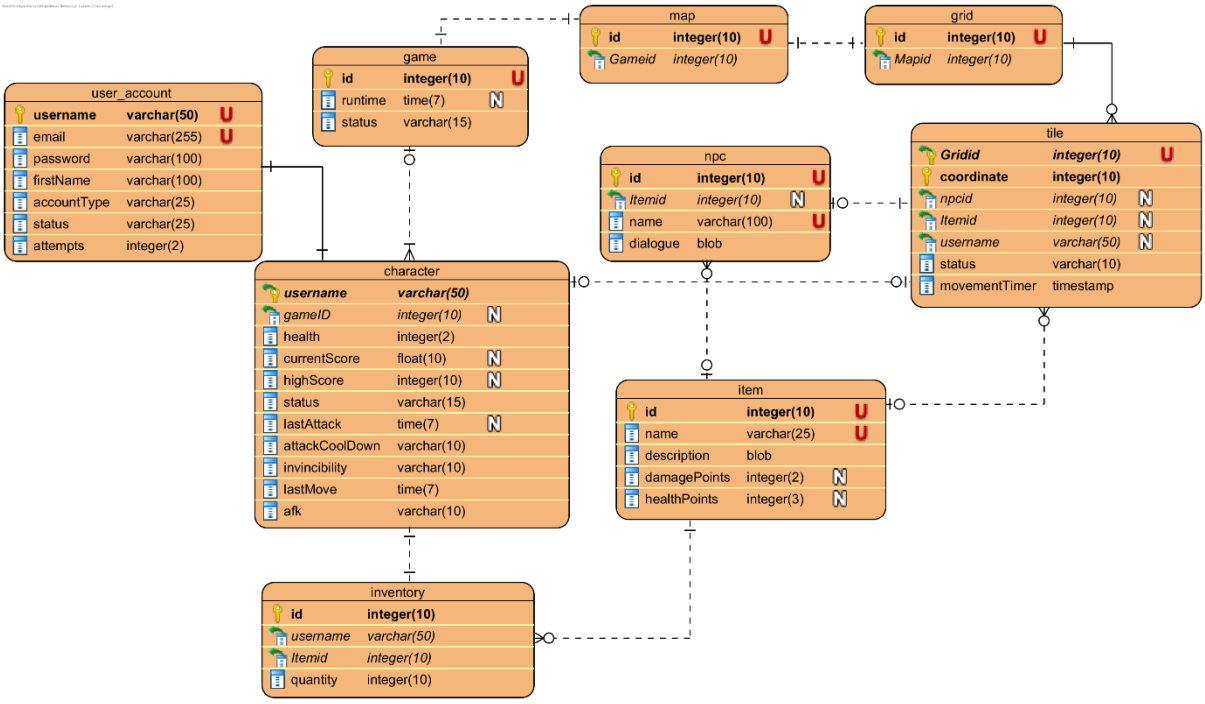
18.3 An admin can ‘kill’ a game which will remove that active gameID from the database and remove each player, sending them back to the login menu.

18.4 An admin can see a list of all player usernames that are currently playing an active game selected from the active games list. They can left-click a player’s username to highlight them for further management.

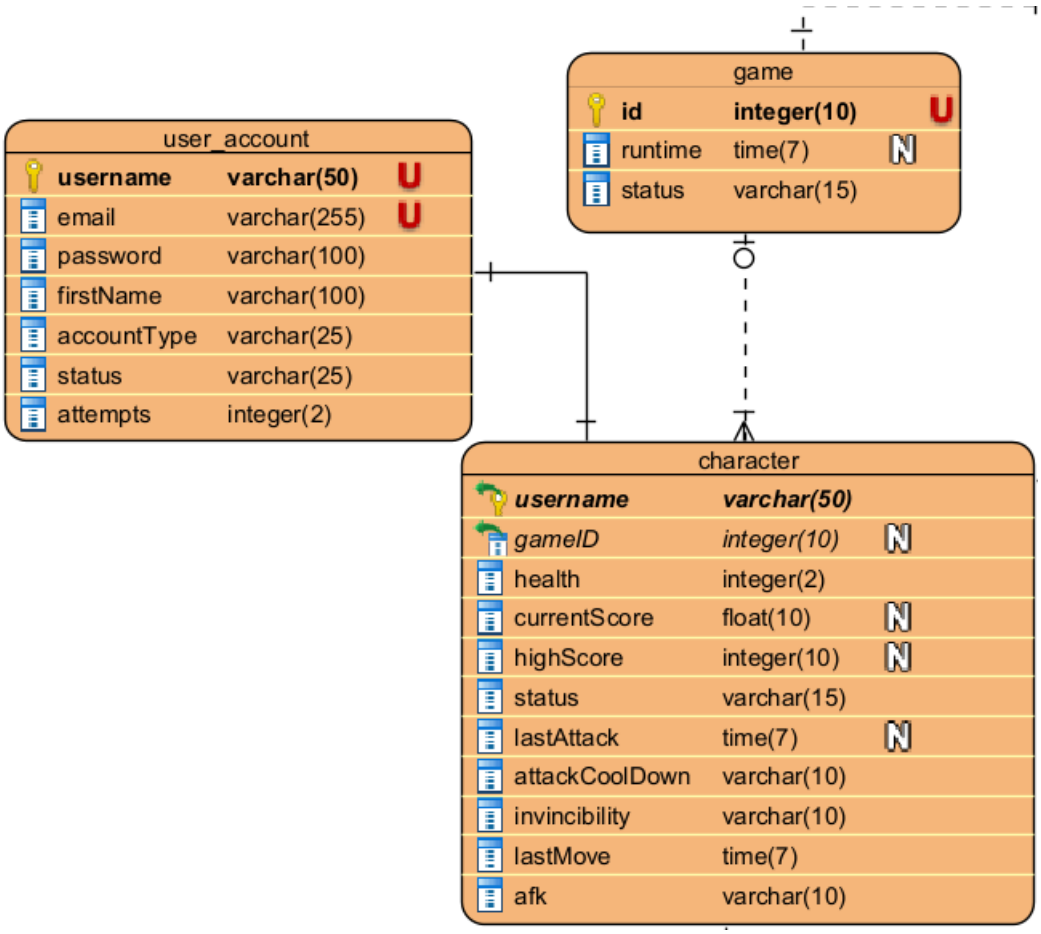
18.5 An admin can remove a highlight player from their game, returning them to the login menu.

18.6 An admin can ban a highlighted player from DUST2DUST, which will delete their account information from the database.

Entity Relationship Diagram

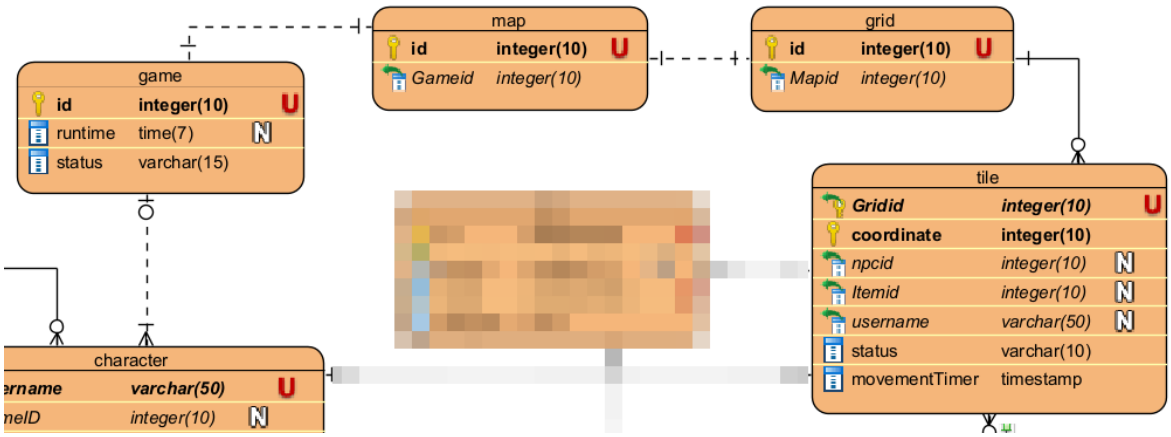


User accounts & Player Characters



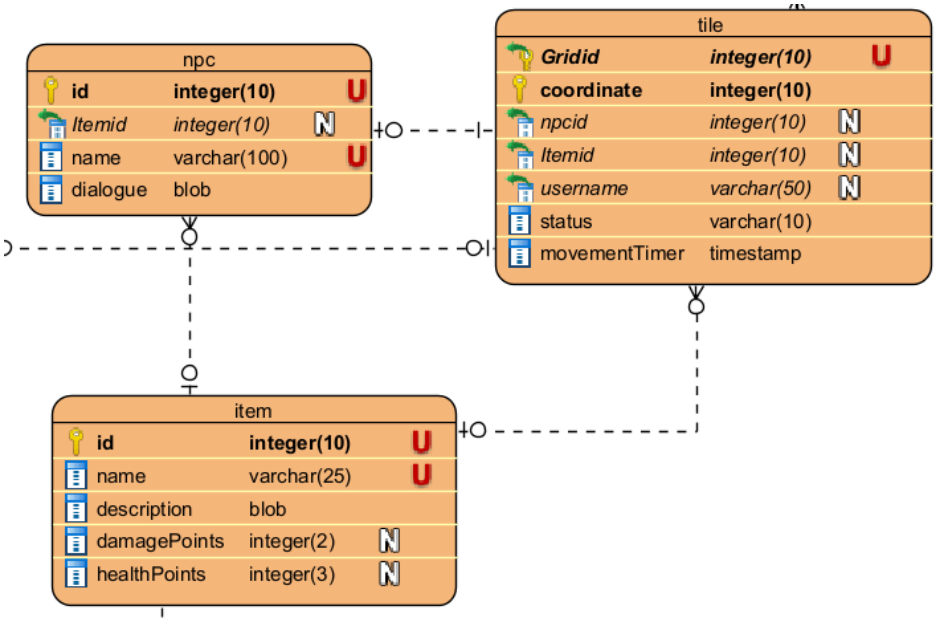
1. The user account tables stores account and account login information including:
    a. Username – The primary key identifier of a user account which users can create themselves with up to 50 characters. As a primary key, each entry of a username must be unique.
    b. Email - A unique email address supplied by the user. This will be used to contact a user from an administrator account.
    c. Password - The account's password. A string of up to 100 characters that will act with the username as a key to logging into an account.
    d. firstName – The first name or preferred name of the account user. A business rule of DUST2DUST is that only first name's are necessary for communication formalities between administrators and standard users.
    e. accountType – Set as either a standard player or administrator account. The text stored in this field will be checked against a constraint to ensure that the correct account type label is entered that will set the role and abilities the account has.
    f. Attempts – Stores the amount of login attempts the user has made. When the integer reaches === 3, the status of the account will be set to LOCKED.
2. A user account will be able to join the game on one character. This table will store active data on player characters, updating specific fields during game play.
    a. Username - This character will be known by the account username, acting as both a foreign key and primary key.
    b. gameID – This is a nullable foreign key displaying the id of a game the player character may currently be active in.
    c. health – Stores and updates an integer that represents the player's health points in the game. These will be lost and gained during action in a game.
    d. currentScore – Stores and updates the current accumulated score the player has earned by surviving in the game and winning duels.
    e. highScore – Stores the highest score a player has earned. This will be queried on the public scoreboard and the character death screen.
    f. status – A check constraint text which displays the player's status as Active or Offline (in a game or not in a game).
    g. lastAttack – When a player makes an attack move, they will be allotted a cooldown timer before they can attack again. This field stores the runtime of the game as HH:MM:SS when the player made their last attack move.
    h. attackCooldown – This is a check constraint field that checks the time of the lastAttack field and applies a momentary 'cooldown' as true or false. True will disable the player from being able to attack for an amount of time. After that time as passed from the lastAttack field, the cooldown will become false, enabling the player to attack. This is to prevent spamming of damage between players.
    i. Invincibility – A check constraint field that applies an invincibility status as true or false. When a player is attacked, this field will become true, making the player immune to damage for a short time. This is to prevent being damaged by multiple players at once too quickly.
    j. lastMove – A time check field that will update with the runtime stamp of the game every time a player makes any movement in the client.
    k. Afk – A check constraint field that works with the lastMove field to update the status as true or false. This field will default to false, but if a player does not interact with the game within a certain amount of time, the afk field will become true, removing the character from the game.

Game, Maps, Grids, & Tiles



1. The game table will store information about each game run in the server. A game cannot exist until at least one player enters. Upon first player entry, the game table will generate:
    a. GameID - A new gameID to identify that game instance as a primary key.
    b. Runtime - Begin a runtime of that game stored as a HH:MM:SS.
    c. Status - Set a default status of 'Active' which can be quired by the admin accounts to see all active games. This status will update to 'Inactive' when a game has no players or is 'killed' by an admin.
2. When a game is created, the map table will store information on the 3x3 vector grid of grids that will be generated to represent the whole map or boundaries of the game. This will store an ID for the entire map relating to a game via the gameID foreign key.
3. The mapID will foreign key into the grid table which will store information on the 9x9 grid blocks that make up the 3x3 wider map. Each 9x9 grid will be identified with a primary key.
4. The gridID will foreign key to the tile table which will store information on each tile in a grid block including:
    a. GridID/Coordinate - The gridID and coordindate which will act as composite keys to identify both the coordinate and the grid is it related to. Each tile will have a coordinate between 0.0 and 9.9 before reaching the boarder of the next grid.

NPCs, Items, & Tiles

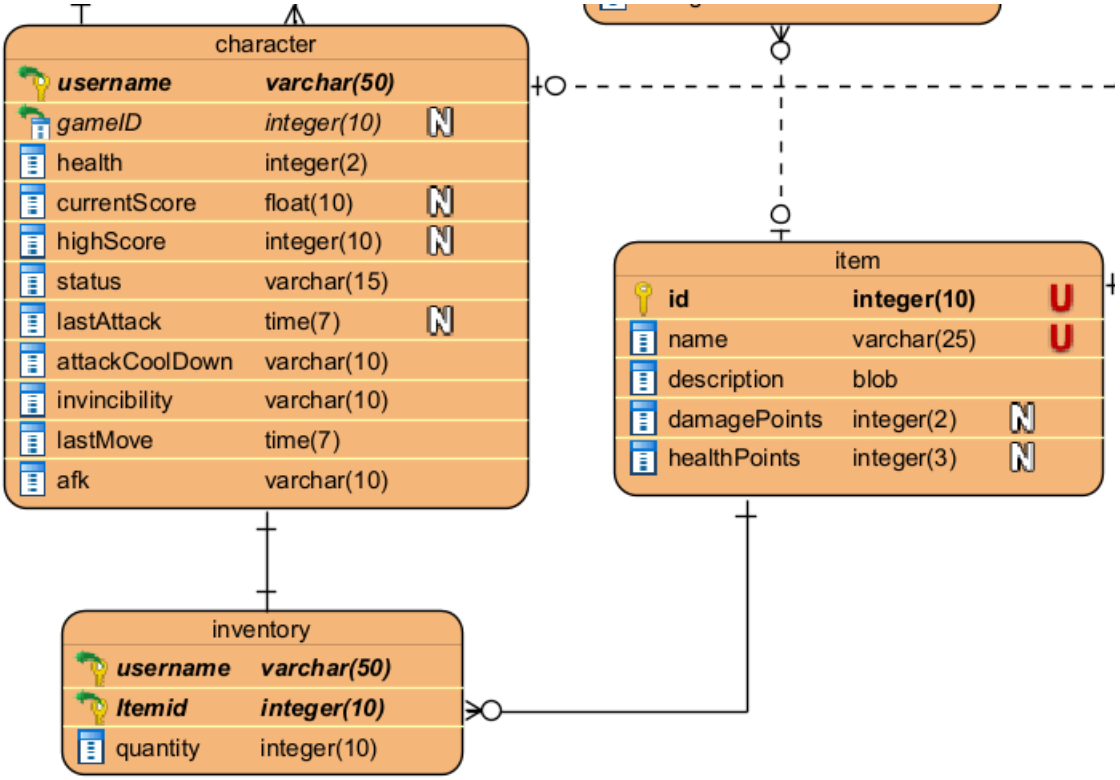


1. The tile table also stores data on the assets that could occupy a coordinate including:
    a. npcID- A nullable NPCID which will foreign key from the npc table to retrieve relevant data if a tile becomes an NPC.
    b. itemID - A nullable itemID which foreign keys from the item table to retrieve relevant data if a tile becomes an item.
    c. Username - A nullable username which retrieves data from the character table to identify both where a player is to the player themselves and others.
    d. Status - A check constraint field that will update the status of a tile as 'empty' or 'taken'. This status will determine a player's ability to move onto a tile.
    e. movementTimer – When a tile is checked as containing npc data, a timer stored at HH:MM:SS will be used to update the NPC's position every few seconds to show movement across the grid.
2. The NPC table stores pre-defined and generated information on Non-player character (NPC) assets that can be generated into a game and interacted with by players including:
    a. npcID – A unique identified for each NPC which will foreign key into the tile table.
    b. itemID – Some NPCs may give an item to the player when interacted with. The itemID is a nullable foreign key which will return the primary key of an item for the NPC to give but will be nullable as not all NPCs will give items.
    c. Name – The full name of the NPC stored as varchar text e.g. *Wyatt Earp.*
    d. Dialogue – A blob object that stores a string of long text that will be returned by the dialogue window that appears when a player interacts with a character.
3. The item table stores pre-defined data on items that can occupy tiles, be given out by NPCs, and collected/used by players. The data stored includes:
    a. itemID – A primary key to uniquely identify each item implemented in the game.
    b. Name – A meaningful name given to the item that will be displayed in the player's GUI (e.g. Bullet).
    c. Description - A short description stored as blob text which will be shown in the item window, illuding to its use in the game.
    d. damagePoints – A nullable integer which allots the amount of damage points the item can deal. This number will be subtracted from the enemy player's health field

when damage is dealt to them through player combat. Not every item can deal damage.

e. healthPoints – A nullable integer which allots the amount of health points the item can restore to the player character's health field on use. Not every item returns health.

Character Inventory & Items



1. The inventory table will store data on items that a player has collected and used throughout the gameplay.
2. A field within an inventory is identified by a composite key (Ravikiran, 2024) of the the foreign key itemID which will identify each item in the character's inventory and the username foreign key of the character who owns the inventory.
3. All items will exist in the player's inventory with a default integer of 3 bullets and 0 whiskeys. The quantity will update when items are collected or used during gameplay.

MySQL DDL

Drop Schema, Drop Tables

```
1  DROP SCHEMA IF EXISTS dust2dust;
2  CREATE SCHEMA dust2dust;
3
4  USE dust2dust;
5
6  DROP TABLE IF EXISTS `tile`;
7  DROP TABLE IF EXISTS `inventory`;
8  DROP TABLE IF EXISTS `item`;
9  DROP TABLE IF EXISTS `npc`;
10 DROP TABLE IF EXISTS `grid`;
11 DROP TABLE IF EXISTS `map`;
12 DROP TABLE IF EXISTS `game`;
13 DROP TABLE IF EXISTS `character`;
14 DROP TABLE IF EXISTS `user_account`;
15
```

1. A collection of statements that will drop the entire existing schema /database used to run the entire script from scratch.
2. Create the schema / database.
3. Use the created schema / database.
4. Drop each table created in the existing database to run the script from the beginning without the constraints of foreign keys and dropping tables in use (IBM, 2024).

Creating the Game & Map

```
23
24 CREATE TABLE `game`(
25     `gameID` INT PRIMARY KEY,
26     `runtime` TIME NULL,
27     `status` VARCHAR(10) NOT NULL
28 );
29
30
31 CREATE TABLE `map`(
32     `mapID` INT PRIMARY KEY,
33     `gameID` INT,
34     FOREIGN KEY (`gameID`) REFERENCES `game` (`gameID`)
35 );
36
37
38 CREATE TABLE `grid`(
39     `gridID` INT PRIMARY KEY,
40     `mapID` INT,
41     FOREIGN KEY (`mapID`) REFERENCES `map` (`mapID`)
42 );
```

1. Creating the game table, setting the gameID as the primary key.
2. Creating the map table, setting the mapID as the primary key and the associated gameID as a foreign key.
3. Creating the grid table, setting the gridID as the primary key and the corresponding mapID as a foreign key.

User Account

```
CREATE TABLE `user_account`(
    `username` VARCHAR(50) PRIMARY KEY UNIQUE,
    `email` VARCHAR(255) UNIQUE,
    `password` VARCHAR(100),
    `firstName` VARCHAR(100),
    `accountType` VARCHAR(25),
    `status` VARCHAR(25),
    `attempts` INT (3)
);
```

1. Creating the user account table with the username as the primary key and the associated criteria with unique constraint on the email address to prevent re-use of an existing email (W3 Schools, 2024).

Character (Player)

```
CREATE TABLE `character`(
    `username` VARCHAR(50) PRIMARY KEY,
    `gameID` INT NULL,
    `status` VARCHAR (10),
    `health` INT(4),
    `currentScore` INT(10),
    `highScore` INT(10),
    `lastAttack` TIME,
    `attackCooldown` VARCHAR (10),
    `invincibility` VARCHAR (10),
    `lastMove` TIME,
    `afk` VARCHAR (10),
    FOREIGN KEY (`username`) REFERENCES `user_account` (`username`),
    FOREIGN KEY (`gameID`) REFERENCES `game` (`gameID`)
);
```

1. Creating the character table which will use the username foreign key as the primary key to identify the player and the gameID foreign key of an active game they may be in. This nullable for when a character is not in a game.

Items & NPCs

```
91  CREATE TABLE `item`(
92      `itemID` INT PRIMARY KEY,
93      `itemName` VARCHAR(25),
94      `description` TEXT,
95      `damagePoints` INT(2) NULL,
96      `healthPoints` INT(2) NULL
97  );
98
99
100 CREATE TABLE `npc`(
101     `npcID` INT PRIMARY KEY,
102     `npcName` VARCHAR(100),
103     `dialogue` TEXT,
104     `itemID` INT NULL
105 );
```

1. Creating the item table which will identify each item by the itemID primary key.
2. Creating the npc table which will identify each npc by the npcID primary key. The itemID foreign key will provide an item to some NPCs who are design to give items to a player character.

Map Tiles

```
110 CREATE TABLE `tile`(
111     `coordinate` DECIMAL (19,0),
112     `gridID` INT,
113     `npcID` INT NULL,
114     `itemID` INT NULL,
115     `username` VARCHAR(50) NULL,
116     `status` VARCHAR (10) NOT NULL,
117     `movementTimer` TIME,
118     PRIMARY KEY (`coordinate`, `gridID`),
119     FOREIGN KEY (`gridID`) REFERENCES `grid` (`gridID`),
120     FOREIGN KEY (`npcID`) REFERENCES `npc` (`npcID`),
121     FOREIGN KEY (`itemID`) REFERENCES `item` (`itemID`),
122     FOREIGN KEY (`username`) REFERENCES `character` (`username`)
123 );
```

1. Creating the tile table which is identified through a composite key of the gridID foreign key and the tile coordinate. A tile may be an NPC, character, or item which are joined through nullable foreign keys.

Player Inventory

```
/* INVENTORY CREATE TABLE */

CREATE TABLE `inventory`(
    `username` VARCHAR(50),
    `itemID` INT,
    `quantity` INT NULL,
    PRIMARY KEY (`username`, `itemID`),
    FOREIGN KEY (`username`) REFERENCES `character` (`username`),
    FOREIGN KEY (`itemID`) REFERENCES `item` (`itemID`)
);
```

1. Creating the inventory table which will be identified through the username foreign key of the table owner and the itemID foreign key of the item stored in the inventory.

MILESTONE 2 – CRUD DEVELOPMENT

Game Procedures, Functions, & Database Access Objects

Login / Lockout

What is the procedure?

The MySQL procedure **login** takes two written parameters from the **user_account** table; **username**, and **password**. When this procedure is run, it checks first for null or a non-existing entry in the username field, returning an error message: “Invalid login!” through being run both in the MySQL workbench and handled in the C# GUI when called through the **LoginandSignup Database Access Object.**

If the username does exist and the password parameter matches the record, the user will successfully login. The login GUI in the C# application will pass the argument when the parameter have been entered into the corresponding fields and the ‘login’ button has been clicked. When successful, the GUI will take the user to the account menu where options for logged-in players will appear.

The **status** field in the **user_account** table in the database will update the user’s status to **‘**Logged-in’.

If the **username** parameter is found to exist in the **user_account** table, but the given **password** does not match that user’s record, the procedure will pass a SQL state 45000 error, preventing any updates. A message box will appear in the application GUI alerting the player, “Invalid login. Try again!”. An invalid login with a correct **username** parameter will add 1 to the **attempts** field in the **user_account** table. The **login** procedure caps the number of **attempts** to 3, meaning that =>3 **attempts** on a correct **username** will update the **user_account status** to ‘Locked’. The GUI will inform the user through a message box “Account has been locked. Please contact an admin.” Any login attempts after this, invalid or valid will throw a SQL state 45000 error, preventing any updates until the account is unbanned by an admin.

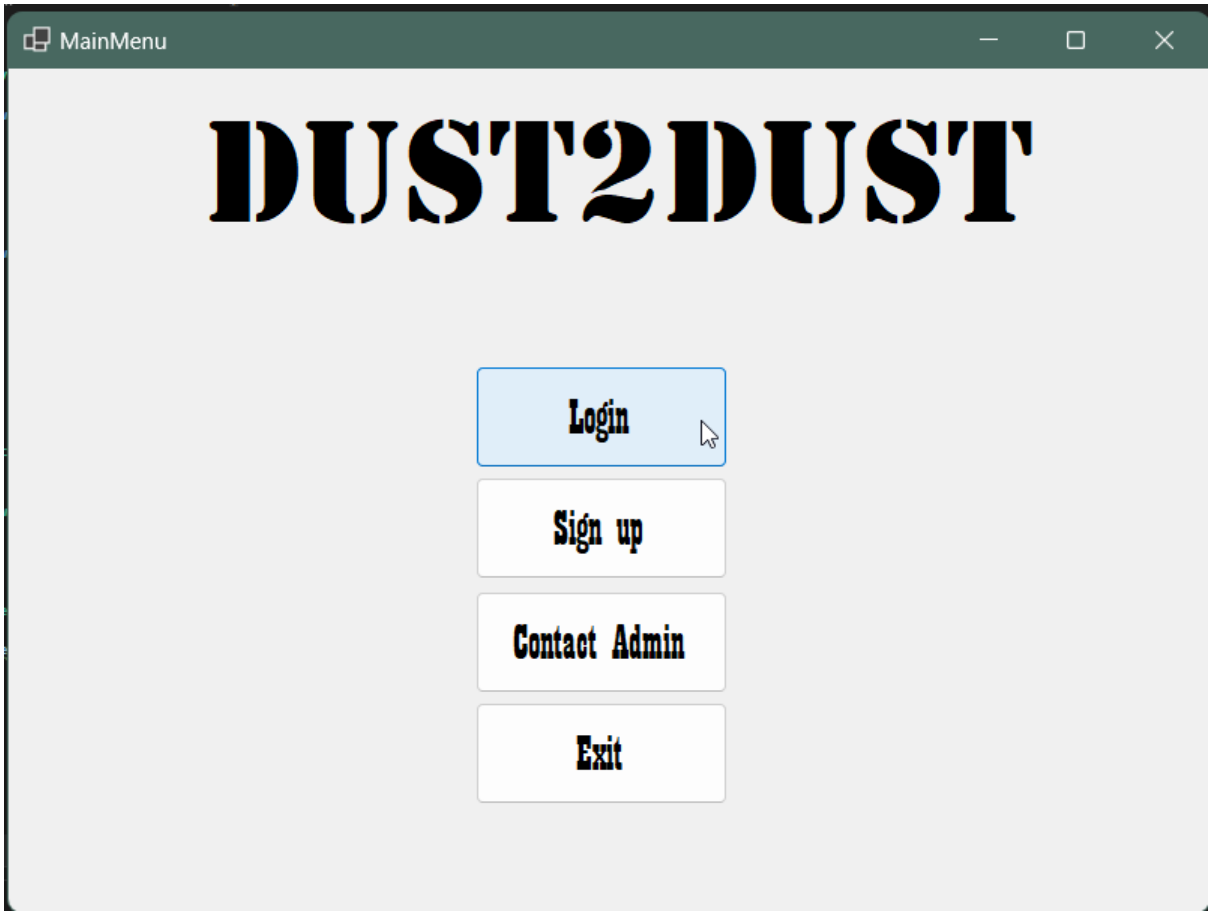
MySQL Procedure

```
DROP PROCEDURE IF EXISTS login;

DELIMITER $$

CREATE PROCEDURE login(IN `username_para` VARCHAR(50), IN `password_para` VARCHAR(100))
COMMENT 'Check login'
BEGIN
    DECLARE `status` VARCHAR(10) DEFAULT 'Logged out';
    DECLARE `attempts` INT DEFAULT 0;

    SELECT ua.`status`, ua.`attempts`
    INTO `status`, `attempts`
    FROM `user_account` ua
    WHERE ua.`username` = `username_para`;

    IF `status` = 'Locked' THEN
        SELECT 'Account Locked' AS MESSAGE;
    ELSEIF EXISTS (
        SELECT 1
        FROM `user_account` ua
        WHERE ua.`username` = `username_para`
        AND ua.`password` = `password_para`
    ) THEN
        UPDATE `user_account` ua
        SET ua.`status` = 'Online',
            ua.`attempts` = 0
        WHERE ua.`username` = `username_para`;
        SELECT 'Logged In' AS MESSAGE;
    ELSE
        UPDATE `user_account` ua
        SET ua.`attempts` = ua.`attempts` + 1
        WHERE ua.`username` = `username_para`;

        SELECT ua.`attempts`
        INTO `attempts`
        FROM `user_account` ua
        WHERE ua.`username` = `username_para`;

        IF `attempts` >= 3 THEN
            UPDATE `user_account` ua
            SET ua.`status` = 'Locked'
            WHERE ua.`username` = `username_para`;
            SELECT 'Invalid Login: Account Locked' AS MESSAGE;
        ELSE
            SELECT 'Invalid Login' AS MESSAGE;
        END IF;
    END IF;
COMMIT;

END $$

DELIMITER ;
```

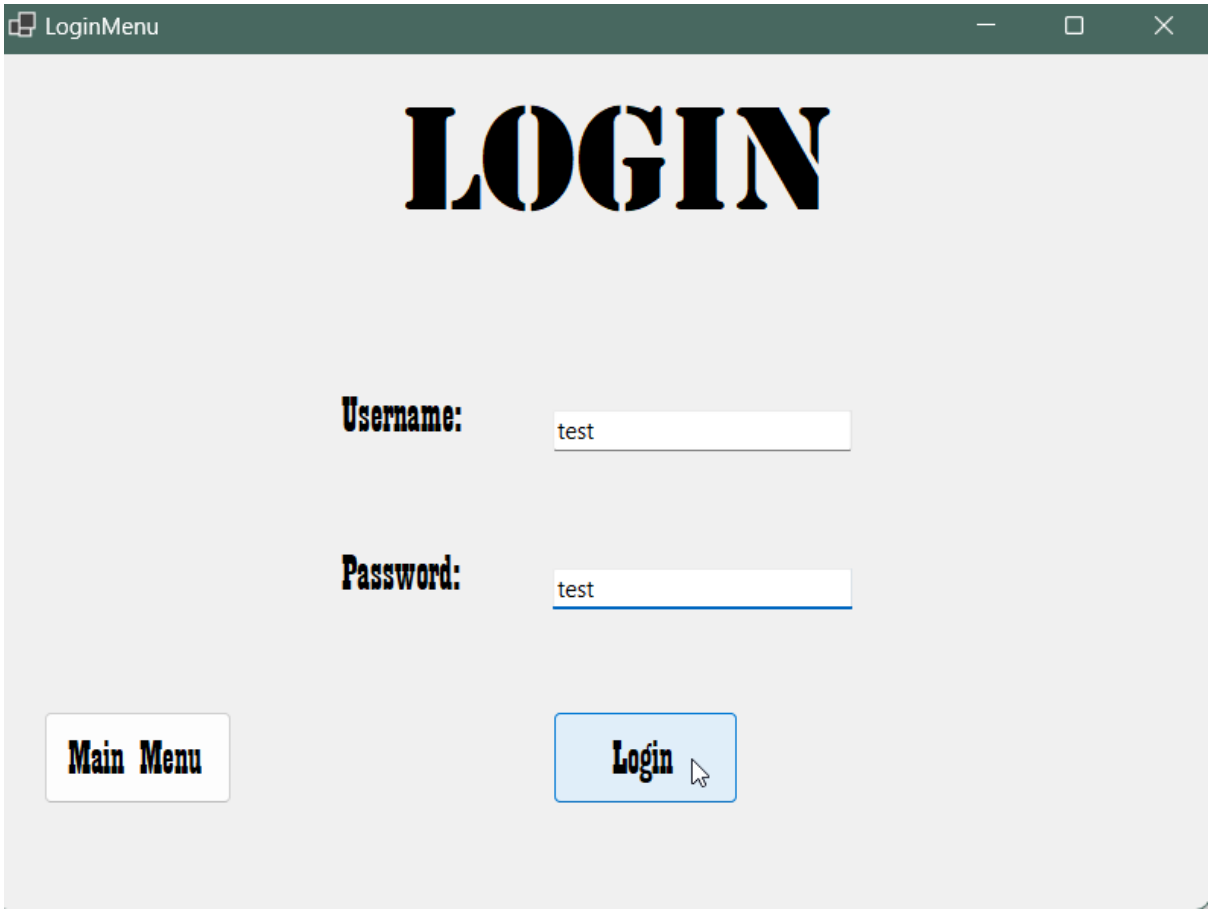
C# DAO call

```
public string login(string username_para, string password_para)
{
    try
    {
        List<MySqlParameter> procedure_params = new()
        {
            new()
            {
                ParameterName = "@username",
                MySqlDbType = MySqlDbType.VarChar,
                Size = 50,
                Value = username_para
            },
            new()
            {
                ParameterName = "@password",
                MySqlDbType = MySqlDbType.VarChar,
                Size = 100,
                Value = password_para
            }

        };

        DataSet auth_result = MySqlHelper.ExecuteDataset(DatabaseAccessObject.MySqlConnection, "call login(@username, @password)", procedure_params.ToArray());


        DataRow auth_result_row = auth_result.Tables[0].Rows[0];

        return auth_result_row.ItemArray[0].ToString();

    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

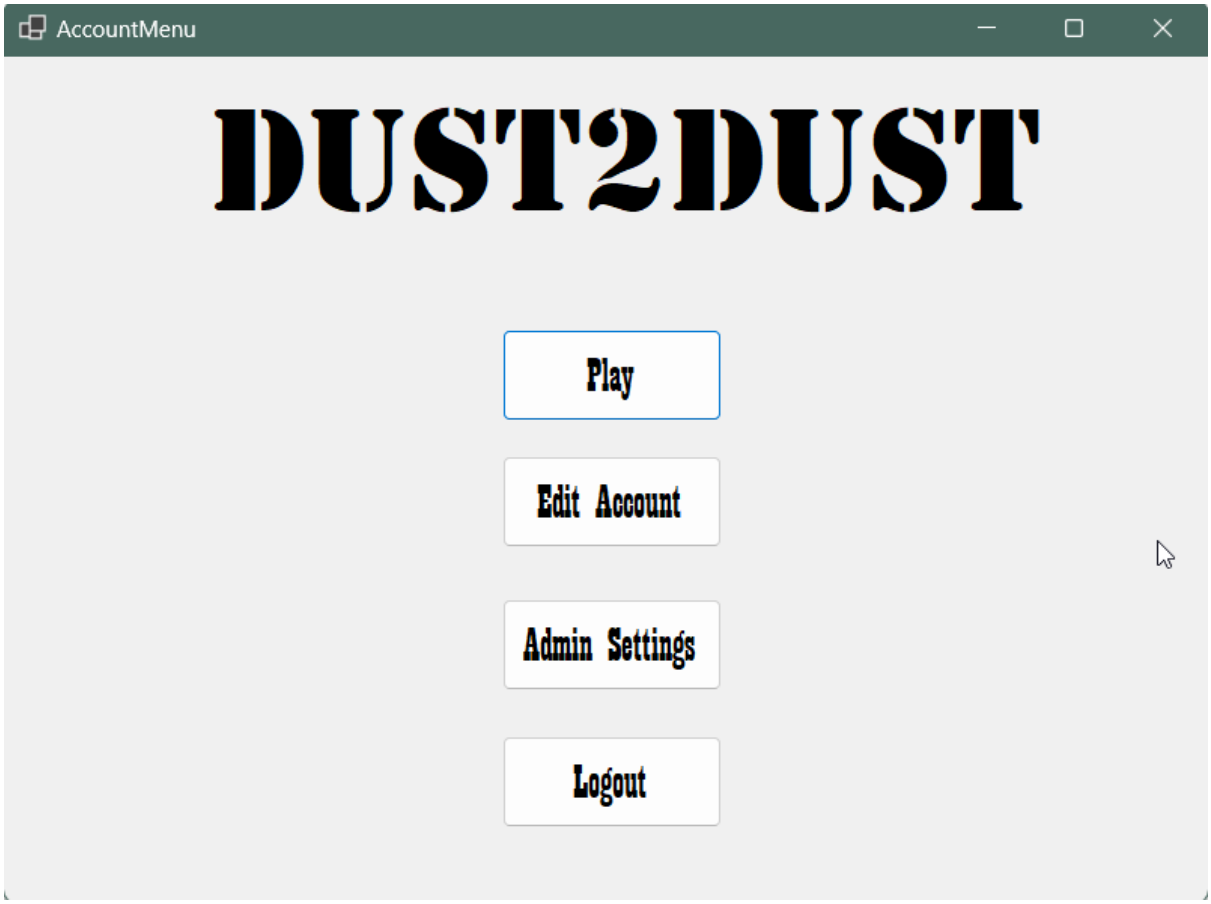
Application GUI



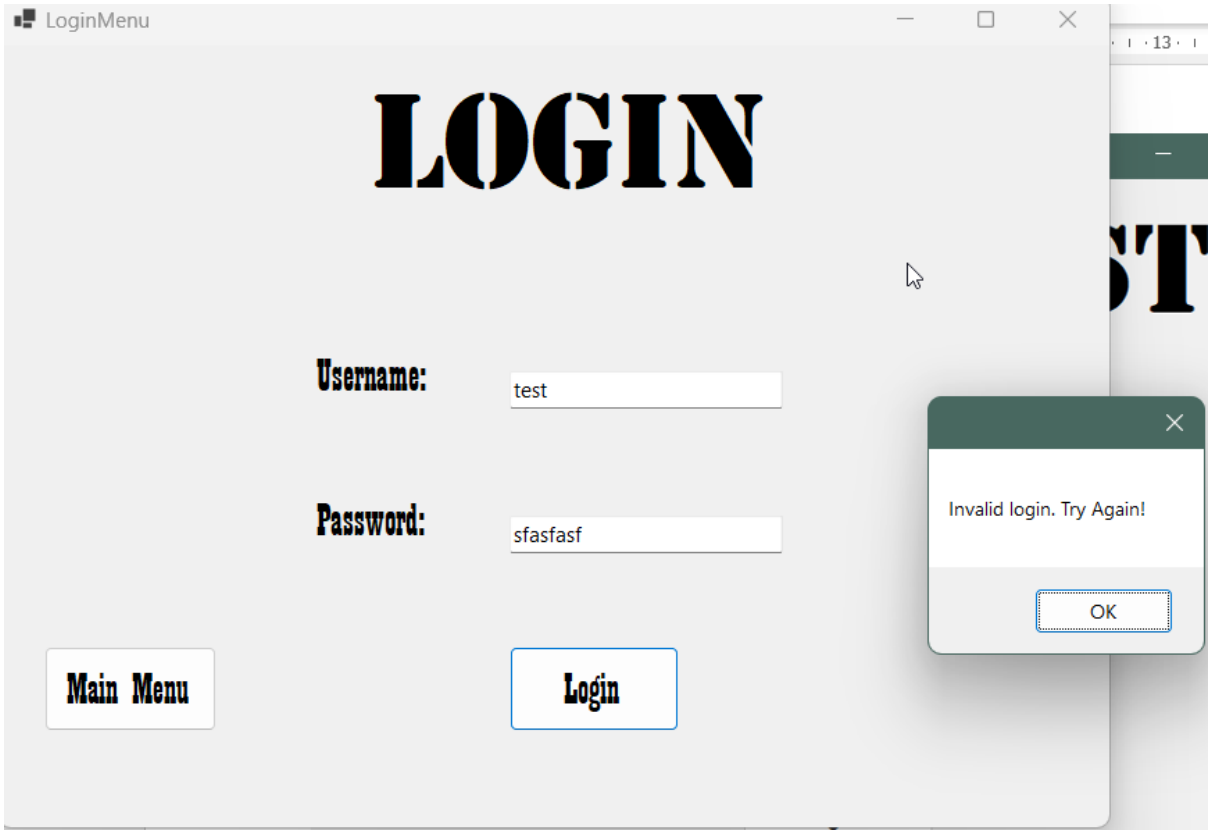
1 Accessing the login menu from the main menu.



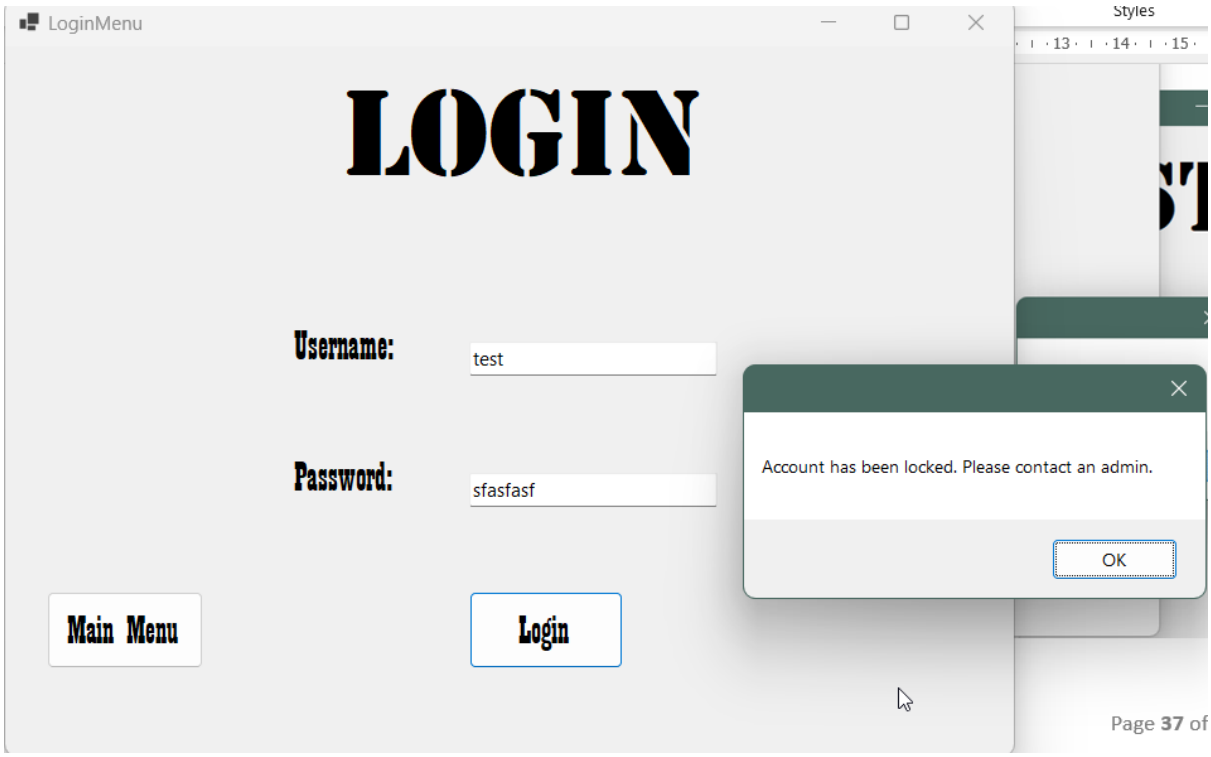
2 Inserting successful login parameters into the username and password inputs.



3 Successful login returns the logged in account menu.



4 Unsuccessful login returns error window



5 > 3 attempts at an unsuccessful login will lock the account

Logout

The **logout** procedure in the MySQL database will change the given user's **status** in the **username parameter** to 'Logged out'. This is called through the LoginandSignup database access object in the C# application when selecting the **Log out** button in the account menu GUI.

MySQL Procedure

```
DROP PROCEDURE IF EXISTS logout;

DELIMITER $$
CREATE PROCEDURE logout(IN `username_para` VARCHAR (50))
BEGIN
    DECLARE `status` VARCHAR (10);

    SELECT ua.`status`
    INTO `status`
    FROM `user_account` ua
    WHERE ua.`username` = `username_para`;

    SELECT ua.`status` FROM `user_account` ua
    WHERE ua.`username` = `username_para`;
    IF `status` = 'Logged out' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT  = "Error: That account is already logged out.";
    END IF;

    UPDATE `user_account` ua
    SET ua.`status` = 'Logged out'
    WHERE ua.`username` = `username_para`;
    SELECT 'Logged Out' AS MESSAGE;

COMMIT;
END $$

DELIMITER ;
```

Register account

For a user to login, they must first have a registered account. This can be accessed via the main menu when the Dust2Dust C# application is run. The MySQL procedure **signup** will take parameters from the new user that correspond with key fields in the **user_account** table including **username, email, password,** and **firstName.** The **signup** procedure checks that all parameters are not null, sending a SQL state 45000 error in such case before checking that the given username and given email are both unique or not already existing within the **user_account** table.

In the C# application GUI, a field is given for each necessary criteria and is passed through the database access object to call the **signup** procedure upon clicking the 'Sign up' button where an error message box may inform the user that their fields are null, or their username or email is already taken. If all parameters are acceptable, the GUI will show a message box, letting the player know their account has successfully been created before taking them to the login menu in the GUI where the **login** procedure can be accessed.

MySQL Procedure

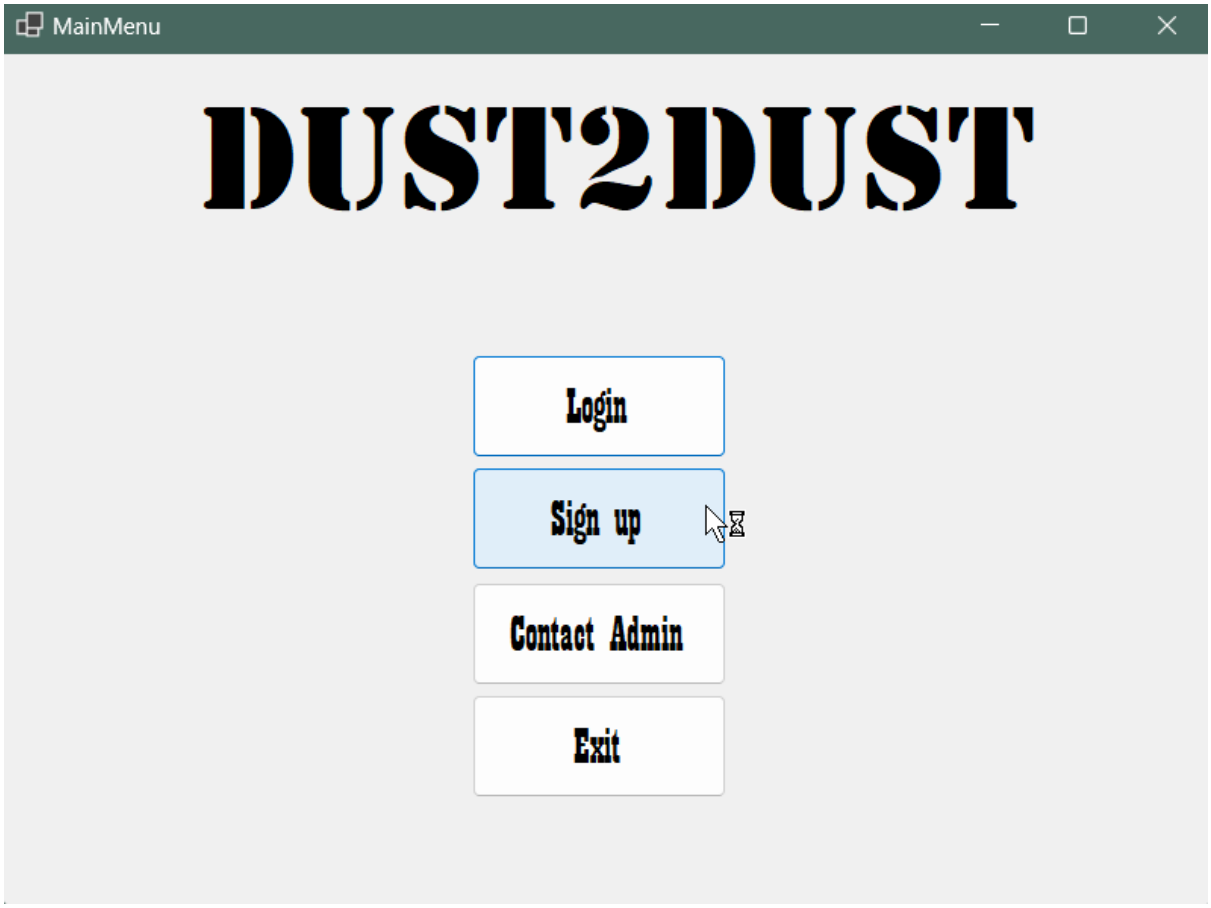
```
DROP PROCEDURE IF EXISTS signup;

DELIMITER $$

CREATE PROCEDURE signup (IN `username_para` VARCHAR(50), IN `email_para` VARCHAR(255), IN `password_para` VARCHAR(100), IN `firstName_para` VARCHAR (100))
BEGIN
    DECLARE var_exists INT;
-- Check existing usernames in user account table
    SELECT COUNT(*) INTO var_exists
    FROM user_account
    WHERE `username` = `username_para`;
    IF var_exists > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'That username is taken! Try something different.';
    END IF;
-- Check existing email in user account table
    SELECT COUNT(*) INTO var_exists
    FROM user_account
    WHERE `email` = `email_para`;
    IF var_exists > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'That email address is already in use! Try something different.';
    END IF;

 -- Reject null criteria
     IF (`username_para` IS NULL OR `username_para` = '')
         OR (`email_para` IS NULL OR `email_para` = '')
         OR (`password_para` IS NULL OR `password_para` = '')
         OR (`firstName_para` IS NULL OR `firstName_para` = '') THEN
         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Some fields are null!';
     END IF;

 -- Insert new user if critera is acceptable
     INSERT INTO user_account (`username`, `email`, `password`, `firstName`) VALUES (`username_para`, `email_para`, `password_para`, `firstName_para`);
     SELECT 'Account created!' AS MESSAGE;

 COMMIT;

 END $$

 DELIMITER ;
```

C# Database Access Object

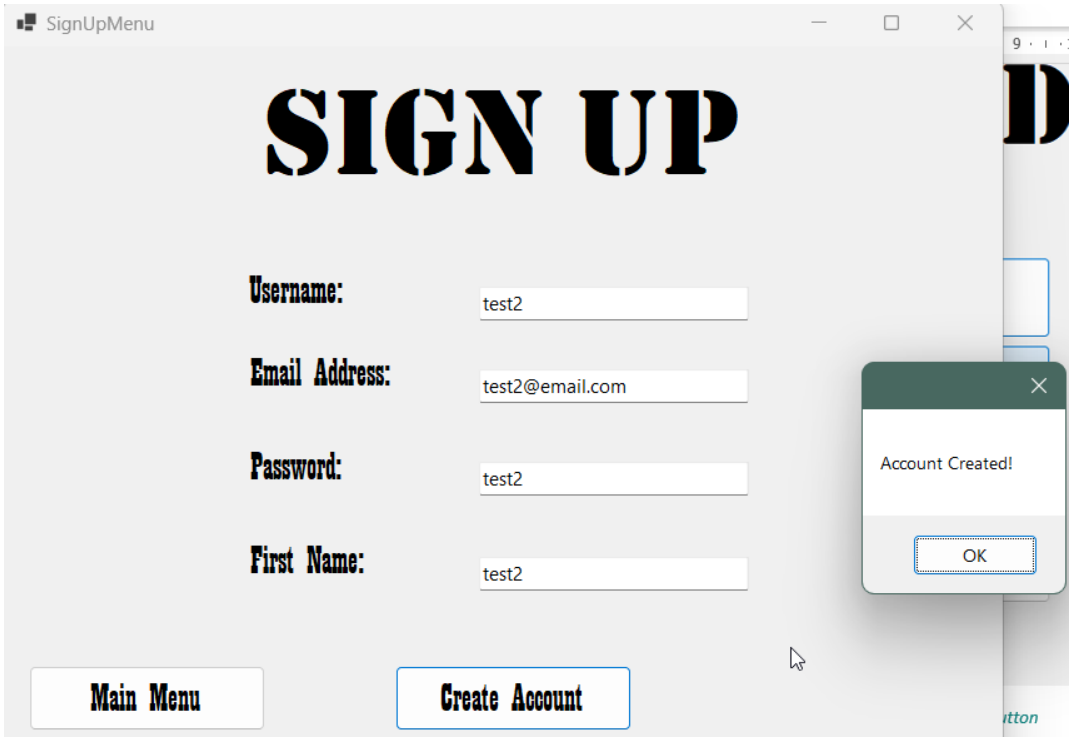
```
namespace dust2dustpart3
{
    4 references
    internal class LoginAndSignupDAO : DatabaseAccessObject
    {

        1 reference
        public string signup(string username_para, string email_para, string password_para, string firstName_para)
        {
            try
            {

                List<MySqlParameter> procedure_params = new()
                {
                    new()
                    {
                        ParameterName = "@username",
                        MySqlDbType = MySqlDbType.VarChar,
                        Size = 50,
                        Value = username_para
                    },
                    new()
                    {
                        ParameterName = "@email",
                        MySqlDbType = MySqlDbType.VarChar,
                        Size = 100,
                        Value = email_para
                    },
                    new()
                    {
                        ParameterName = "@password",
                        MySqlDbType = MySqlDbType.VarChar,
                        Size = 50,
                        Value = password_para
                    },
                    new()
                    {
                        ParameterName = "@firstName",
                        MySqlDbType = MySqlDbType.VarChar,
                        Size = 100,
                        Value = firstName_para
                    }
```

```
    };

    foreach (var param in procedure_params.ToArray())
    {
        Console.WriteLine(param.Value);
    }

    DataSet register_result = MySqlHelper.ExecuteDataset(DatabaseAccessObject.MySqlConnection, "call signup(@username, @email, @password, @firstName)", procedure_params.ToArray());


    return register_result.Tables[0].Rows[0].ItemArray[0].ToString();
}
catch (Exception ex)
{
    throw ex;
}
```

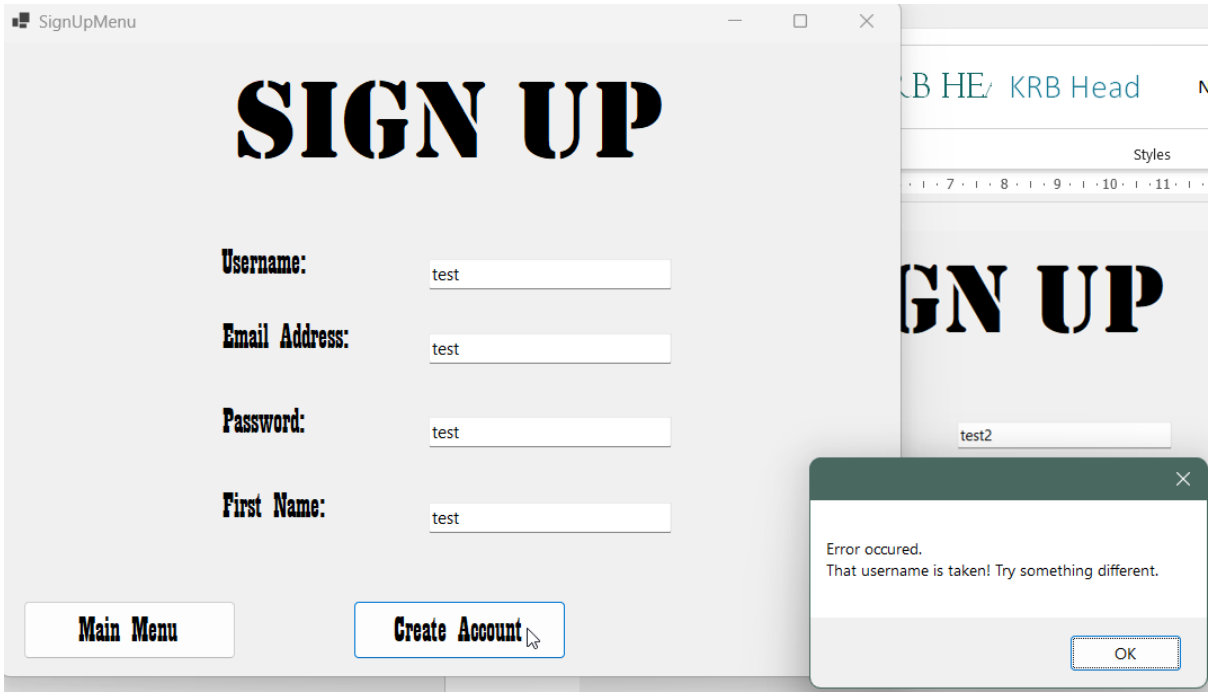
Application GUI



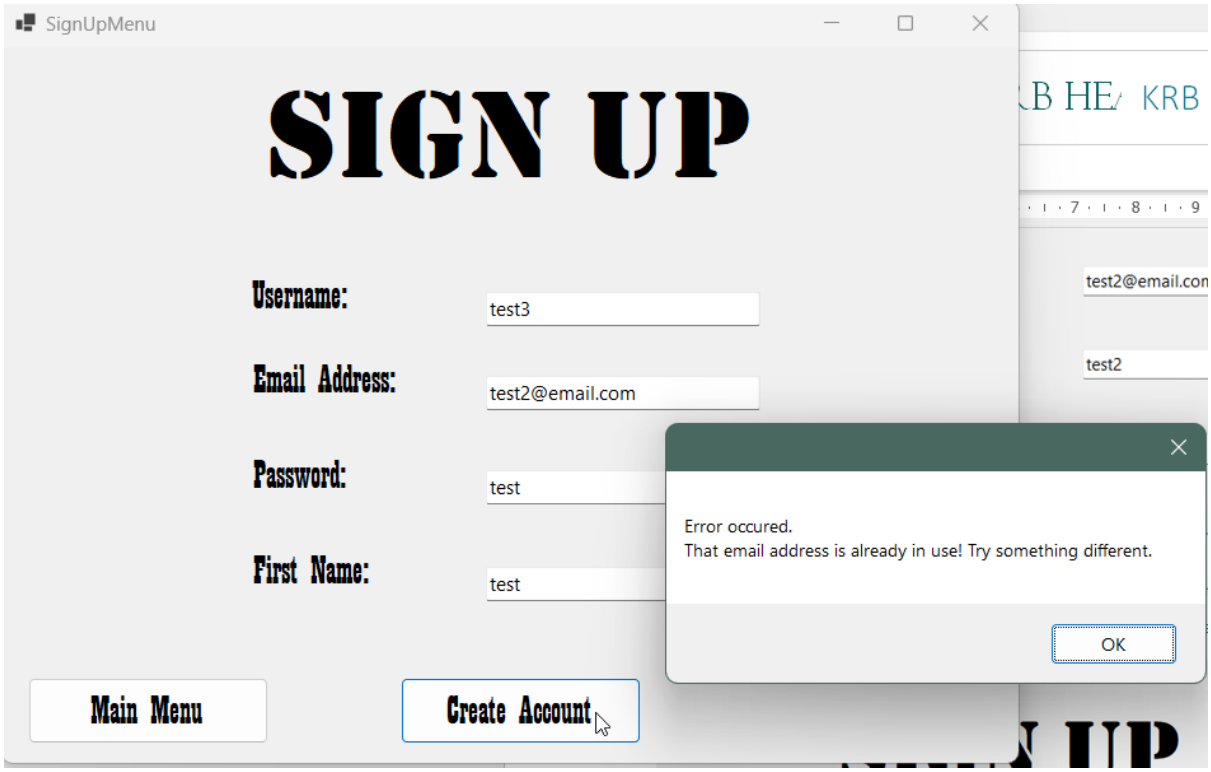
1 Access the sign up menu through the main menu via the sign up button



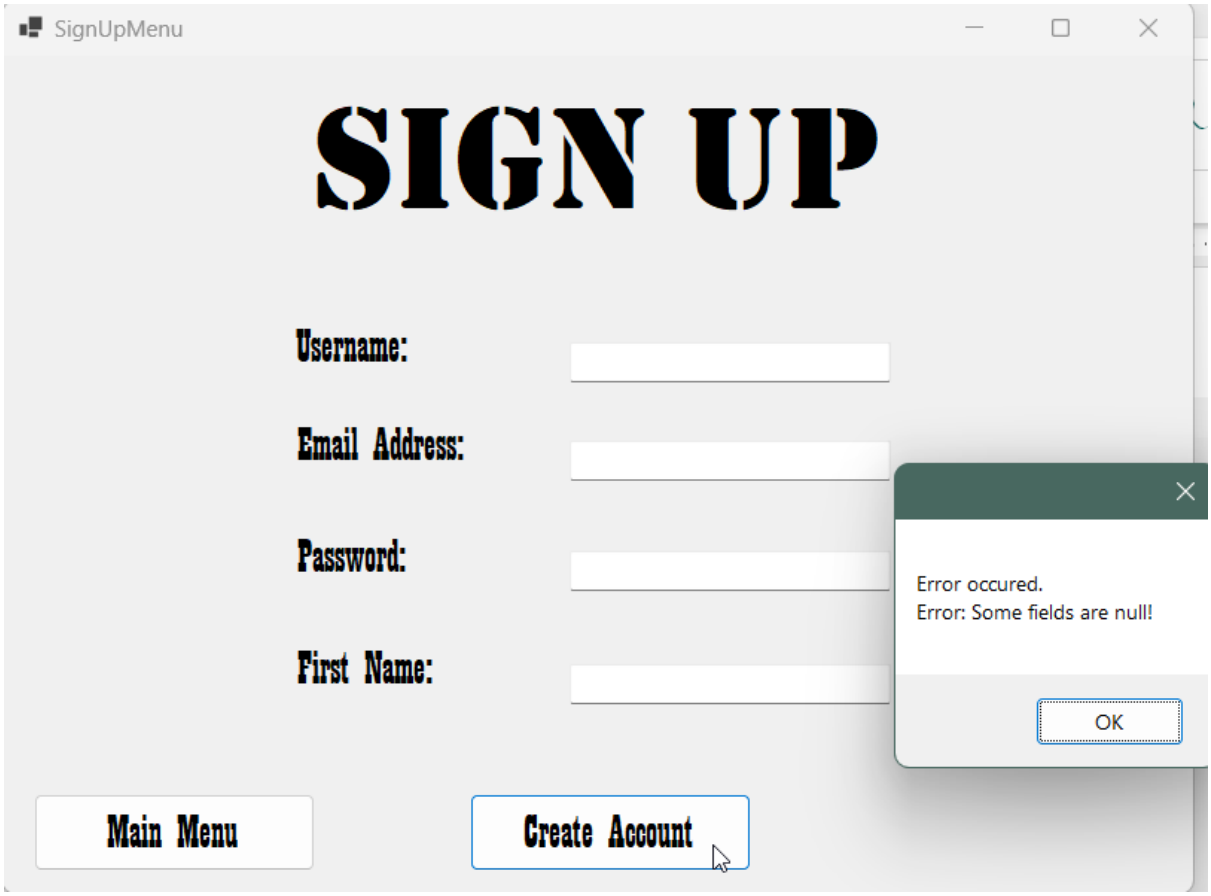
2 Successful account creation given unique username and email parameters when passed via the 'create account' button.



3 Unsuccessful account creation when passing an existing username.



4 Unsuccessful account creation when passing an existing email address.



5 Unsuccessful account creation when attempting to pass null fields.

Edit account

Once logged in, a user can edit their account details via the **edit account** button in the logged in menu. The **edit_account** procedure in MySQL takes the same parameters as the **signup** procedure, checking for null fields and/or existing usernames and emails. The user can add a new **username, email, password, and firstName** to the fields in the GUI which will pass through the **LoginandSignup** database access object, running the **Update** method which calls the **edit_account** procedure in the database, handling the same errors as in the **signup** procedure. If the account details throw no errors, a message box will appear in the GUI to inform the user that their details have been saved before taking them back to the logged-in menu.

MySQL Procedure

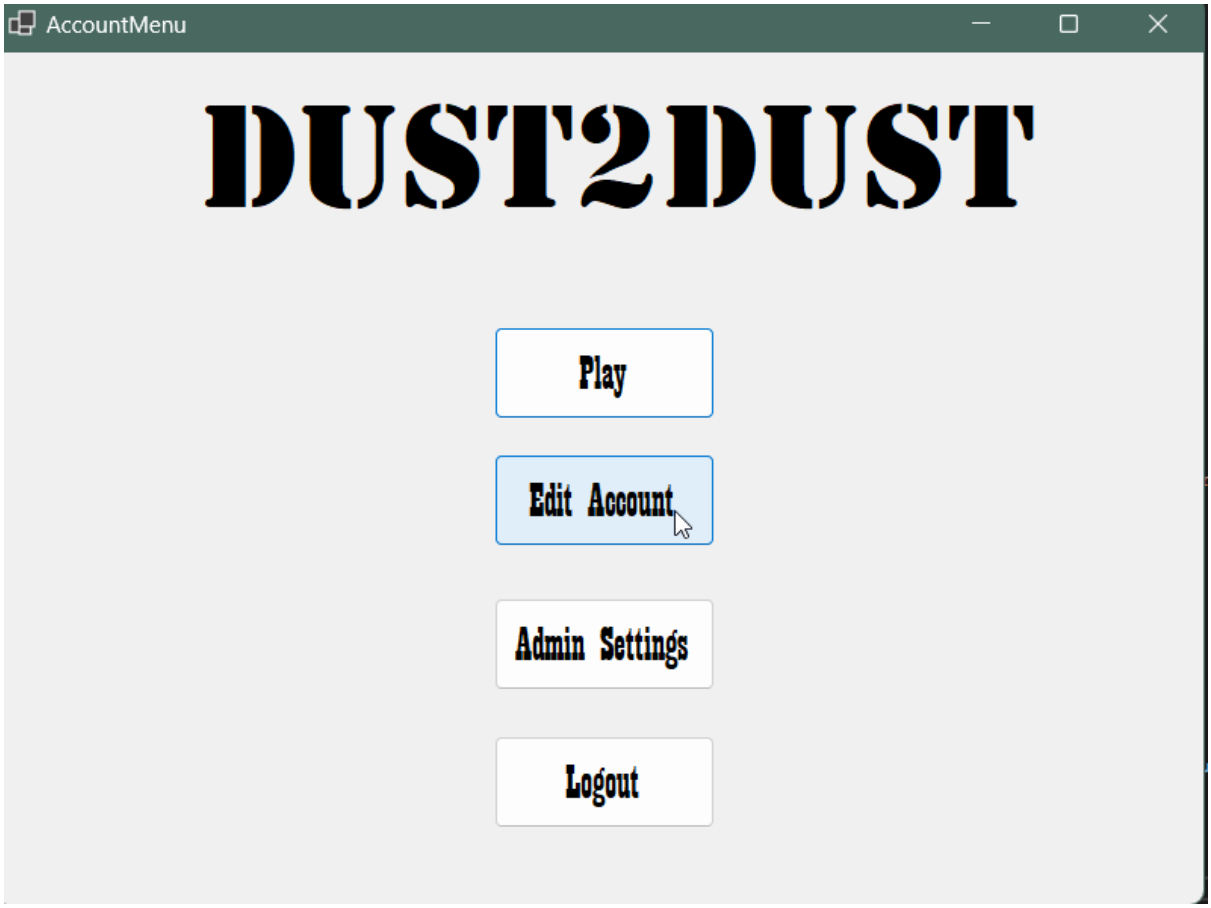
```
DROP PROCEDURE IF EXISTS edit_account;

DELIMITER $$

CREATE PROCEDURE edit_account (IN `username_para` VARCHAR(50), IN `new_username_para` VARCHAR (50), IN `new_email_para` VARCHAR(255), IN `new_password_para` VARCHAR(100), IN `new_firstName_para` VARCHAR (100))
BEGIN
    DECLARE var_exists INT;
-- Check existing usernames in user account table
    SELECT COUNT(*) INTO var_exists
    FROM user_account
    WHERE `username` = `new_username_para`;
    IF var_exists > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: That username is taken! Try something different.';
    END IF;
-- Check existing email in user account table
    SELECT COUNT(*) INTO var_exists
    FROM user_account
    WHERE `email` = `new_email_para`;
    IF var_exists > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: That email address is already in use! Try something different.';
    END IF;
-- Reject null criteria
    IF (`username_para` IS NULL OR `username_para` = '')
        OR (`new_username_para` IS NULL OR `new_username_para` = '')
        OR (`new_email_para` IS NULL OR `new_email_para` = '')
        OR (`new_password_para` IS NULL OR `new_password_para` = '')
        OR (`new_firstName_para` IS NULL OR `new_firstName_para` = '') THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Some fields are null!';
    END IF;

-- Insert new user if critera is acceptable
    UPDATE `user_account`
    SET
        `username` = `new_username_para`,
        `email` = `new_email_para`,
        `password` = `new_password_para`,
        `firstName` = `new_firstName_para`
    WHERE `username` = `username_para`;
    SELECT 'Account updated!' AS MESSAGE;

COMMIT;
```

C# DAO

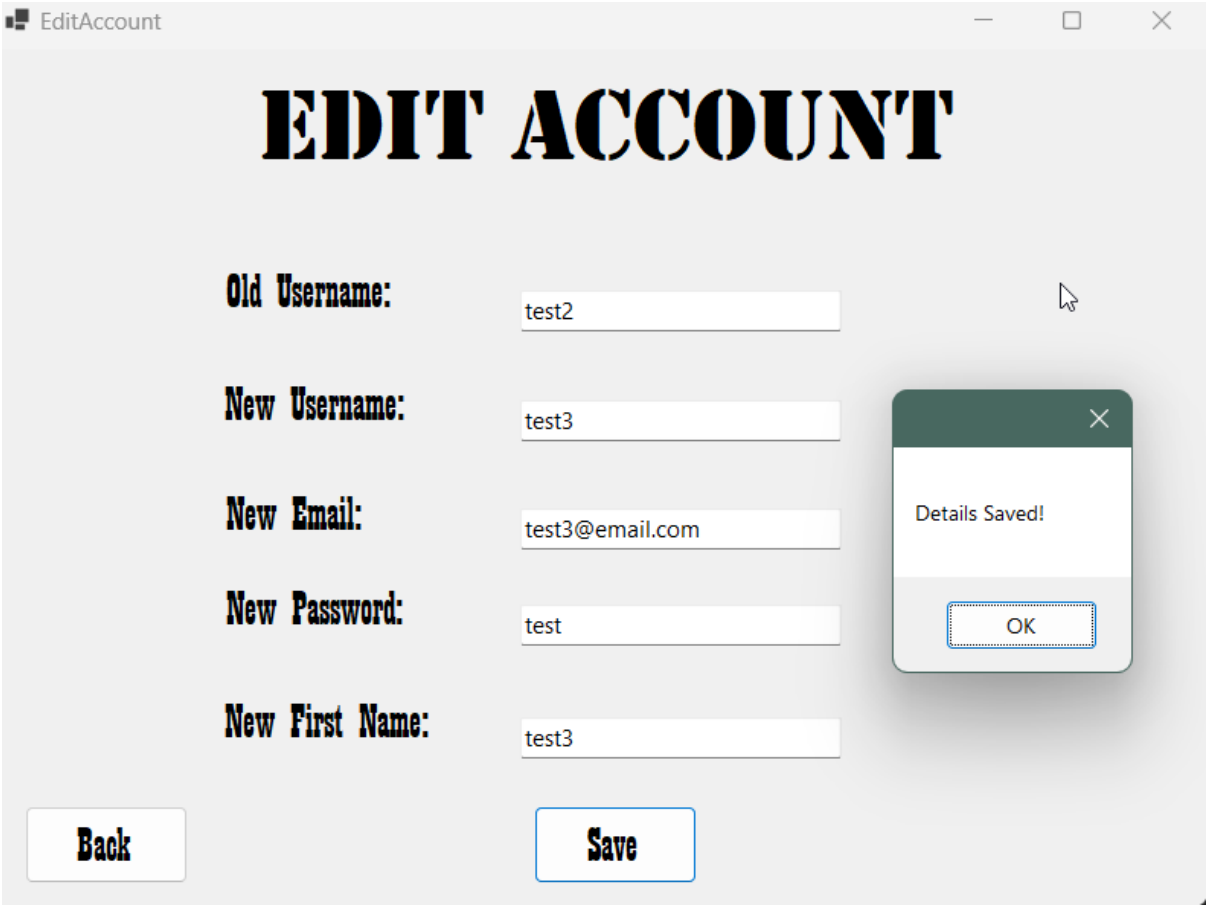
```
public string Update(string username_para, string newusername_para, string newemail_para, string newpassword_para, string newfirstName_para)
{
    try
    {
        List<MySqlParameter> procedure_params = new()
            {
            new()
            {
                ParameterName = "@username",
                MySqlDbType = MySqlDbType.VarChar,
                Size = 50,
                Value = username_para
            },
            new()
            {
                ParameterName = "@newusername",
                MySqlDbType= MySqlDbType.VarChar,
                Size = 50,
                Value = newusername_para
            },
            new()
            {
                ParameterName = "@email",
                MySqlDbType= MySqlDbType.VarChar,
                Size = 255,
                Value = newemail_para

            },
            new()
            {
                ParameterName = "@password",
                MySqlDbType = MySqlDbType.VarChar,
                Size = 100,
                Value = newpassword_para
            },
            new()
            {
                ParameterName = "@firstName",
                MySqlDbType= MySqlDbType.VarChar,
                Size = 100,
                Value = newfirstName_para
            }
```

```
        };
        foreach (var param in procedure_params.ToArray())
        {
            Console.WriteLine(param.Value);
        }

        DataSet register_result = MySqlHelper.ExecuteDataset(DatabaseAccessObject.MySqlConnection, "call edit_account(@username, @newusername, @email, @password, @firstName)", procedure_params.ToArray());


        return register_result.Tables[0].Rows[0].ItemArray[0].ToString();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

1 reference
internal void Update(string username, string? email, string password, string? firstName, string? accountType, string? status, int attempts)
{
    throw new NotImplementedException();
}
```

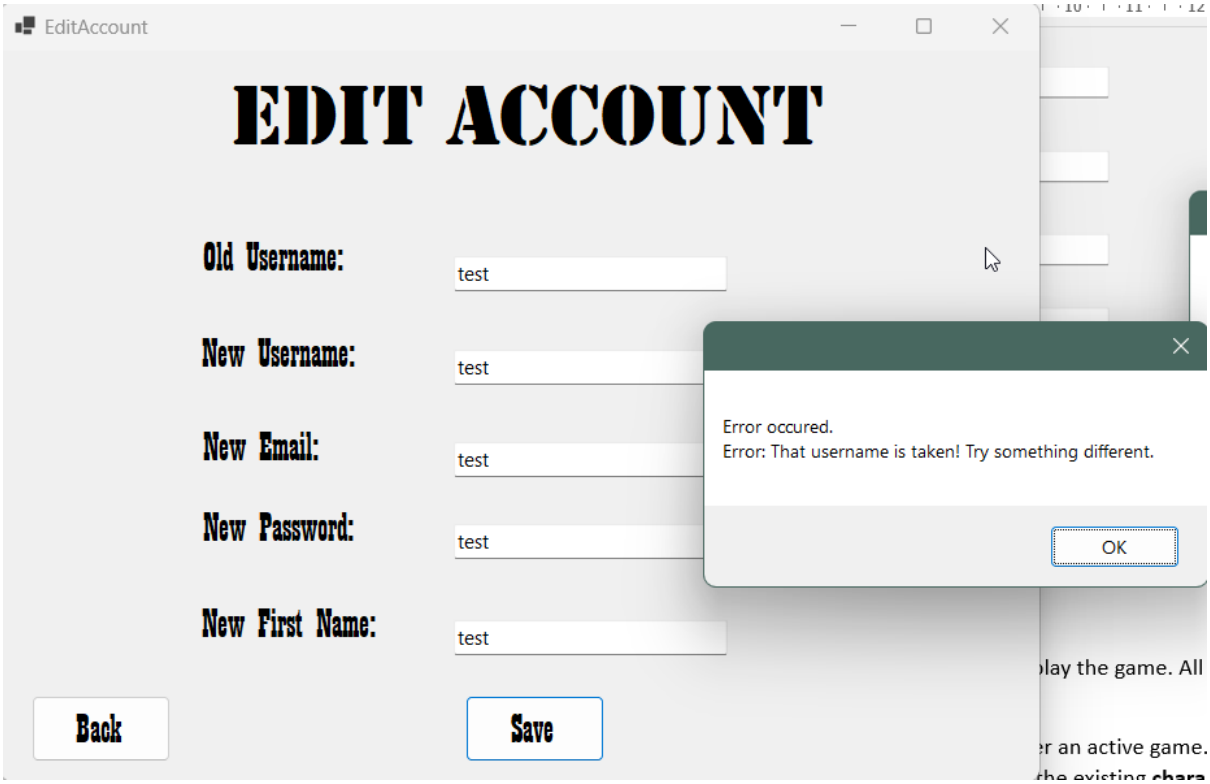
Application GUI



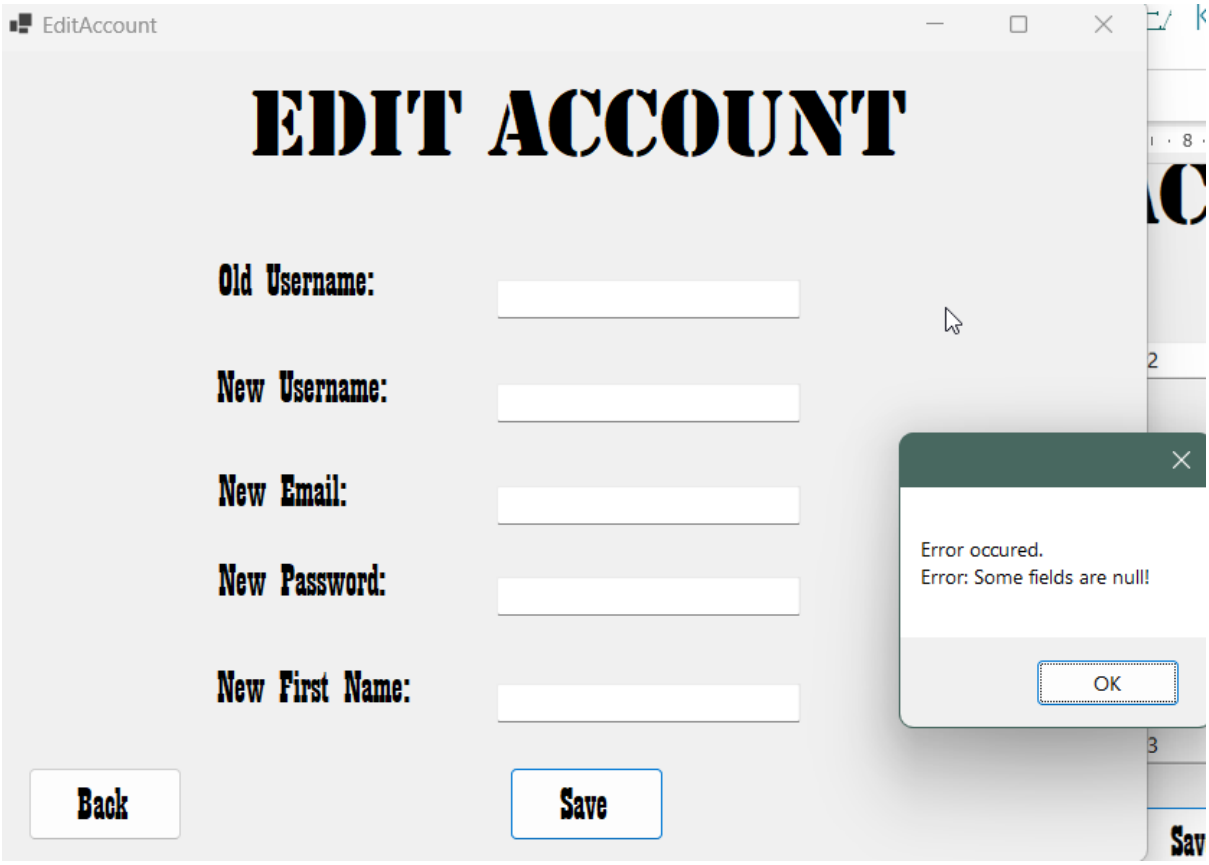
1 Access the edit menu via the 'edit account' button on the logged in account menu



2 Successful call of editing an account with valid input parameters.



3 Unsuccessful call of editing an account with input parameters that already exist in the database.



4 Unsuccessful call of editing an account with null fields

Setting tile types (Items and NPCs)

The function **get_tile_type** in the MySQL script is made to be called within the **draw_gameboard** procedure. **Get_tile_type** fills the **tileType** column when new tiles are generated to assign a 1 (being an item tile) for every two tiles or a 2 for every three tiles (being an NPC tile) when the grid of the gameboard is created. For every tile that is not a 1 or a 2, it will be a default of 0, being an open tile for a player character to move to.

MySQL Function

```
DROP FUNCTION IF EXISTS get_tile_type;

DELIMITER $$

CREATE FUNCTION get_tile_type() RETURNS INT
COMMENT 'Function to generate different tile types when called in the draw_gameboard procedure'
DETERMINISTIC
BEGIN
    -- Create tileType 1 to represent an item on a tile
    IF ROUND(RAND() * 3) = 2 THEN
        RETURN 1;
    -- Create tileType 2 to represent an NPC on a tile
    ELSEIF ROUND(RAND() * 2) = 1 THEN
        RETURN 2;
    ELSE
    -- Create tileType 0 to represent an empty tile
        RETURN 0;
    END IF;

END $$

DELIMITER ;
```

Drawing the gameboard

The procedure **draw_gameboard** is designed to create a new active game of Dust2Dust, generating a new gameID at +1 from the last inserted gameID which will foreign key into a new mapID also a +1 of the last inserted ID, and a grid of tiles, each with a composite key of the mapID, the row number, and column number of that tile. The parameters passed when calling this procedure are the maximum row and columns for the grid which will be inserted into the **map** table at the maxRow and maxCol and the **tile** table as each tile number by row and col (e.g. CALL **draw_gameboard (9,9)** would create a 9x9 grid of 81 tiles).

MySQL Procedure

```
DROP PROCEDURE IF EXISTS draw_gameboard;

DELIMITER $$

CREATE PROCEDURE draw_gameboard(IN `maxRow_para` INT, IN `maxCol_para` INT)
BEGIN
    DECLARE new_game_id INT;
    DECLARE new_map_id INT;
    DECLARE current_row INT DEFAULT 0;
    DECLARE current_col INT DEFAULT 0;
    DECLARE tile_type INT DEFAULT 0;

-- Create a new gameID based on the next numerical order from the last insterted gameID
INSERT INTO `game` (`status`) VALUES ('Active');
SET new_game_id = LAST_INSERT_ID();

-- Create a new mapID based on the next numerical order from the last insterted mapID and related to the last inserted gameID (above)
INSERT INTO `map` (`gameID`) VALUES (new_game_id);
SET new_map_id = LAST_INSERT_ID();
```

```
-- Create a new mapID based on the next numerical order from the last insterted mapID and related to the last inserted gameID (above)
INSERT INTO `map` (`gameID`) VALUES (new_game_id);
SET new_map_id = LAST_INSERT_ID();
-- Create maxium rows and columns that end at a given border, set the tile type with the tile type function to place an item (1) or npc (2) at random
    WHILE current_row < `maxRow_para` DO
        WHILE current_col < `maxCol_para` DO
            SET tile_type = get_tile_type();

             -- Insert an id for each row and column which will identify the tile one by one until the parameter is reached
                INSERT INTO `tile` (`mapID`, `row`, `col`, `tileType`)
                VALUES (new_map_id, current_row, current_col, tile_type);

            SET current_col = current_col + 1;
        END WHILE;

        SET current_col = 0;
        SET current_row = current_row + 1;

    END WHILE;

    SELECT 'New gameboard created!' AS MESSAGE;

COMMIT;

END $$

DELIMITER ;
```

Testing

```
707 ●    -- TEST GAMEBOARD PROCEDURE
708      CALL draw_gameboard(10,10);
709
710      -- View the new game created
```

Result Grid | Filter Rows: | Export: | Wrap

MESSAGE
New gameboard created!

1 New gameID is generated

```
712    -- View the new map created
713 •  SELECT * FROM `map`;
```

Result Grid | Filter Rows:

mapID	gameID	maxRow	maxCol
1	2	10	10
3	4	10	10
NULL	NULL	NULL	NULL

2 New mapID generated for new gameID (4)

```
716    -- View information for all tiles
717 •  SELECT * FROM `tile` WHERE `mapID` = 3;
718
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell

tileID	mapID	row	col	tileType	npcID	itemID	username	movementTimer
182	3	0	0	2	NULL	NULL	NULL	NULL
183	3	0	1	2	NULL	NULL	NULL	NULL
184	3	0	2	1	NULL	NULL	NULL	NULL
185	3	0	3	1	NULL	NULL	NULL	NULL
186	3	0	4	0	NULL	NULL	NULL	NULL
187	3	0	5	0	NULL	NULL	NULL	NULL
188	3	0	6	0	NULL	NULL	NULL	NULL
189	3	0	7	1	NULL	NULL	NULL	NULL
190	3	0	8	2	NULL	NULL	NULL	NULL

3 Tile table called by corresponding mapID to see generated tiles on the gameboard

Add New Players to Game

In Dust2Dust, all users have one character which is used to play the game. All characters are created equal and are recognised by their associated username.

The MySQL procedure **enter_character** allows a user to enter an active game. The procedure first checks to see if the **character** exists already. If so, it returns the existing **character** and sets their stat columns to the defaults listed in the CREATE TABLE statement and inserts the **gameID** foreign key. For character that do not exist, the procedure inserts a new character into the character table with the **username parameter**, updating their stats to the defaults, inserting the **gameID** parameter into the foreign key column, and updating their **status to 'Active'.** Once a character username has been passed, the procedure then sets them on the home_tile in the map of that gameID. This updates the **tile** table to insert the **username_parameter** into the **username** column.

MySQL Procedure

```
DROP PROCEDURE IF EXISTS enter_character;

DELIMITER $$

CREATE PROCEDURE enter_character (IN `username_para` VARCHAR(50), `gameID_para` INT)
BEGIN
    DECLARE home_tile INT;
    DECLARE character_exists INT;

    -- Select an unoccupied tile as the hometile
    SELECT `tileID`
        INTO home_tile
    FROM `tile`
        WHERE `tileType` = 0
        ORDER BY RAND()
        LIMIT 1;
-- Check to see if the player already has a character in the database
    SELECT COUNT(*) INTO character_exists
    FROM `character`
    WHERE `username` = `username_para`;

IF character_exists > 0 THEN
UPDATE `character`
SET  `gameID` = `gameID_para`,
    `status` = 'Active',
    `health` = 10,
    `currentScore` = 0,
    `attackCooldown` = 'Off',
    `invincibility` = 'Off',
    `lastMove` = NOW()
WHERE `username` = `username_para`;
ELSE
-- Insert new character details with the given username parameter, setting all stat columns to defaults
    INSERT INTO `character` (`username`, `gameID`, `status`, `health`, `currentScore`, `attackCooldown`, `invincibility`, `lastMove`)
    VALUES (`username_para`, `gameID_para`, 'Active', 10, 0, 'Off', 'Off', NOW());
END IF;
```

```
    SET autocommit = OFF;
    START TRANSACTION;
    IF EXISTS (
        SELECT  ua.`username`
        FROM `user_account` ua
        WHERE ua.`username` = `username_para`
        )
    THEN
        UPDATE `tile`
        SET `username` = `username_para`
        WHERE `tileID` = home_tile;

        UPDATE `user_account` ua
        SET ua.`status` = 'Active'
        WHERE ua.`username` = `username_para`;

        COMMIT;
    ELSE
        ROLLBACK;
    END IF;

END $$

DELIMITER ;
```

Testing

```
858     -- Call procedure with a new character
859 •   CALL enter_character ('Verga', 4);
860
861     -- See the character on the map
862 •   SELECT * FROM `tile` WHERE `username` = 'Verga';
863
864
865     /*
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell C

tileID	mapID	row	col	tileType	npcID	itemID	username	movementTimer
86	1	8	5	0	NULL	NULL	Verga	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Player movement

Two procedures work to update a player's fields in the **character** table. The first procedure **player_movement** take the parameters **username, direction,** and **mapID** and declares the current row and col the character is on i.e. their tile. When a player moves up, a declaration of new row and new col subtracts 1 from each, updating the new tile. The same works for moving down, adding 1 to row and tile, moving left subtracting 1 and moving right adding 1. When the player has reached the maximum of a given row or column and attempts to move beyond its limits, a SQL state 45000 error is thrown with a message alerting "You can't move that way!", preventing any updates to the player's location. If a tile is occupied by another player, the procedure checks if a **username** exists on that tile. If so, an SQL state 45000 error refused the update and alerts the player that the tile is occupied.

Within this procedure, the function **update_lastMove** is called when a player's tile is updated. This function updates the **lastMove** time to NOW() in the **character** table which is implemented to check that the player is active in the game. This will later relate to an AFK check which will logout a player that has not made a move in three minutes.

MySQL Procedure

```
DROP PROCEDURE IF EXISTS update_lastMove;

DELIMITER $$

CREATE PROCEDURE update_lastMove (IN `username_para` VARCHAR(50))
COMMENT 'Updates the players lastMove column when an action is made. To be called recursively in movement-based procedures'
BEGIN
    UPDATE `character`
        SET `lastMove` = NOW()
        WHERE `username` = `username_para`;
COMMIT;

END $$

DELIMITER ;
```

1 Updating the player's last move. To be called in the player_movement procedure

```
DROP PROCEDURE IF EXISTS player_movement;

DELIMITER $$

CREATE PROCEDURE player_movement(IN `username_para` VARCHAR(50), IN `direction` VARCHAR(10), IN `mapID_para` INT)
BEGIN
    DECLARE current_row INT;
    DECLARE current_col INT;
    DECLARE new_row INT;
    DECLARE new_col INT;

    -- Find the player's current tile
    SELECT `row`, `col` INTO current_row, current_col
    FROM `tile`
    WHERE `username` = `username_para`
    AND `mapID` = `mapID_para`;

    -- Determine the new row and column based on the direction
    CASE `direction`
        WHEN 'up' THEN
            SET new_row = current_row - 1;
            SET new_col = current_col;
        WHEN 'down' THEN
            SET new_row = current_row + 1;
            SET new_col = current_col;
        WHEN 'left' THEN
            SET new_row = current_row;
            SET new_col = current_col - 1;
        WHEN 'right' THEN
            SET new_row = current_row;
            SET new_col = current_col + 1;
        ELSE
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cant move that way!';
    END CASE;

    -- Check if the target tile is no occupied by another player
    IF EXISTS (SELECT 1 FROM `tile` WHERE `row` = new_row AND `col` = new_col AND `username` IS NOT NULL AND `username` != `username_para`) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'That tile is occupied!';
    END IF;
```

```
    -- Update the player's current tile and the target tile
    UPDATE `tile`
    SET `username` = NULL, `tileType` = 0
    WHERE `row` = current_row
    AND `col` = current_col
    AND `username` = `username_para`
    AND `mapID` = `mapID_para`;

    -- Update the username of the occupied tile to the current player
    UPDATE `tile`
    SET `username` = `username_para`
    WHERE `row` = new_row AND `col` = new_col
    AND `mapID` = `mapID_para`;

    CALL update_lastMove (`username_para`);
COMMIT;

END $$

DELIMITER ;
```

Testing

```
967     -- See where the character is in the map
968 •   SELECT * FROM `tile` WHERE `username` = 'test';
969
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cel

tileID	mapID	row	col	tileType	npcID	itemID	username	movementTimer
38	1	3	7	0	NULL	NULL	test	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

1 Select where the user is on the gameboard by tileID

```
964     -- Call the movement procedure, make sure the mapID is correct
965 •   CALL player_movement ('test', 'left', 1);
966
967     -- See where the character is in the map
968 •   SELECT * FROM `tile` WHERE `username` = 'test';
969
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap C

tileID	mapID	row	col	tileType	npcID	itemID	username	movementTimer
37	1	3	6	1	NULL	NULL	test	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

2 Call the player_movement procedure with a given user name, direction, and mapID then select the tile table again to see updates in the tileID column

NPC movement

This was not completed in time for the milestone 2 deadline.

Setting items on tiles

This was attempted in MySQL with a procedure, but could not be completed in time.

```
DROP FUNCTION IF EXISTS place_item;

DELIMITER $$

CREATE PROCEDURE place_items()
BEGIN
    UPDATE `tile` t
    JOIN `item` i ON t.`itemID` = i.`itemID`
    SET t.`itemID` = i.`itemID`
    WHERE t.`tileType` = 1;
END $$

DELIMITER ;
```

Inventory items

This was not completed in time for the milestone 2 deadline.

Scoring points

The **score_points** procedure is called when a player picks up an item or defeats another player. The procedure takes the **username** as a parameter and updates the **currentScore** field in the **character** table, adding 100 to the current integer. This can be called within the inventory procedure when an item quantity increases.

MySQL Procedure

```
DROP PROCEDURE IF EXISTS score_points;

DELIMITER $$

CREATE PROCEDURE score_points (IN `username_para` VARCHAR (50))
COMMENT 'Add 100 to the interger in the currentScore field in the character table'
BEGIN
    UPDATE `character`
    SET `currentScore` = `currentScore` + 100
    WHERE `username` = `username_para`;

COMMIT;

END $$

DELIMITER ;
```

Testing

```
1032    -- See updated currentScore column in the character table
1033    SELECT `username`, `currentScore`
1034    FROM `character`
1035    WHERE `username` = 'test';
1036
```

Result Grid | Filter Rows: | Edit: | Export/Import:

username	currentScore
test	100
NULL	NULL

1 Select currentScore column by username to see their current score

```
1029    -- TEST SCORE POINTS
1030    CALL score_points ('test');
1031
1032    -- See updated currentScore column in the character table
1033    SELECT `username`, `currentScore`
1034    FROM `character`
1035    WHERE `username` = 'test';
1036
```

Result Grid | Filter Rows: | Edit: | Export/Import:

username	currentScore
test	200
NULL	NULL

2 Call the procedure then select the currentScore again to see updates, adding 100 to last inserted integer

Seeing all accounts

At an administrator level, a user account can access admin settings accessed through the admin settings button on the logged-in menu in the GUI. In the database exist admin-level procedures that are called through the Admin and User class Database Access Objects in C#.

The **all_accounts** procedure is called through the UserDAO when selecting the 'player accounts' button in the admin menu. This will return a list of all accounts in the **user_account** table in the database in a data grid view in the GUI. From here the admin can select from the admin settings which call the following actions.

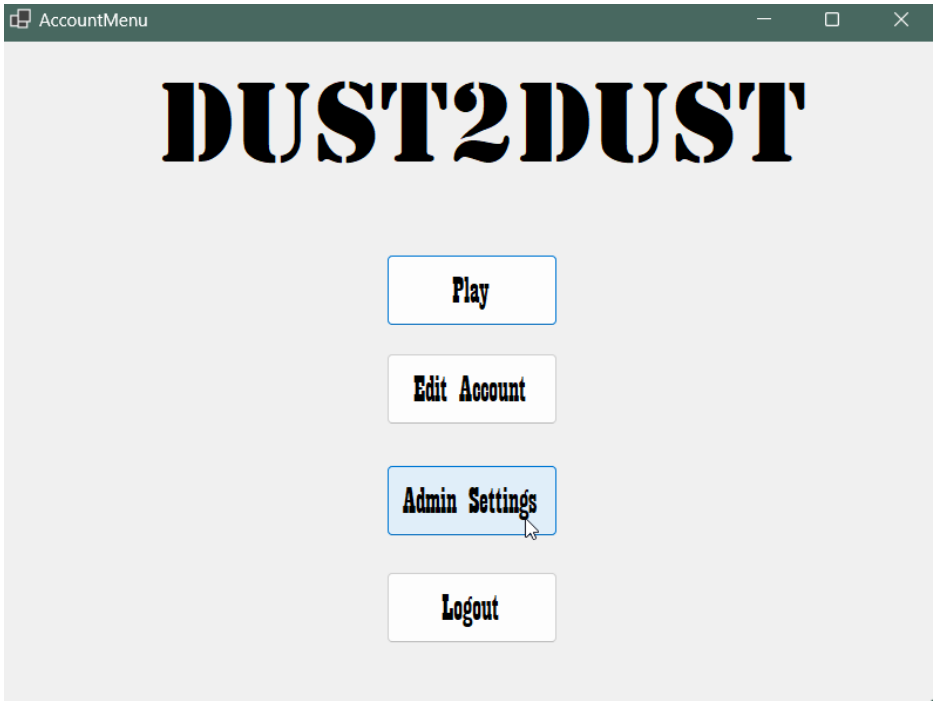
MySQL Procedure

```
DROP PROCEDURE IF EXISTS all_accounts;

DELIMITER $$

CREATE PROCEDURE all_accounts()
BEGIN
    SELECT *
    FROM user_account;

COMMIT;

END $$

DELIMITER ;
```

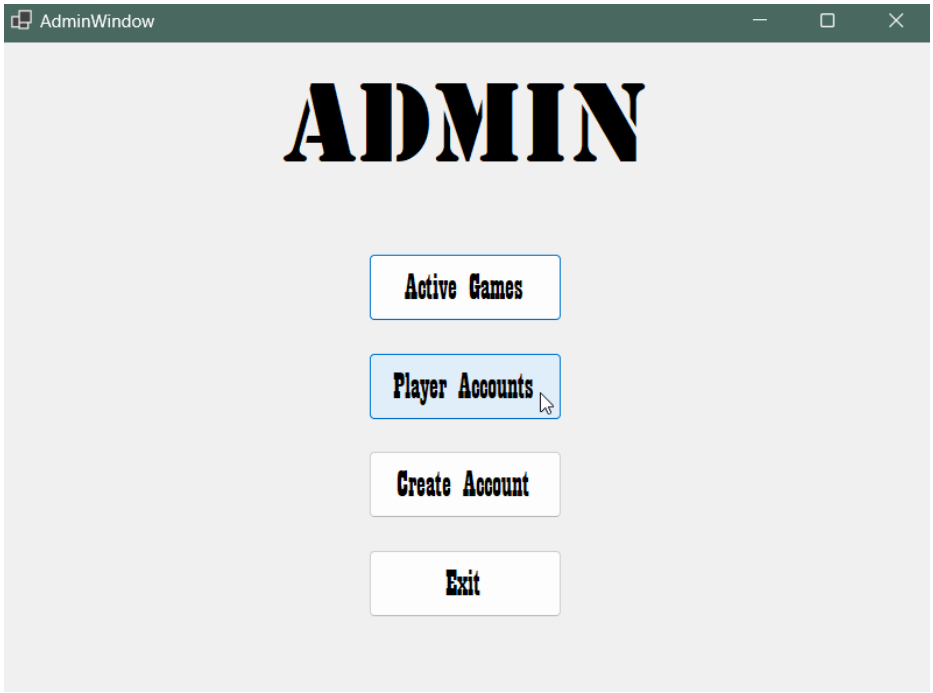
C# DAO

```
namespace dust2dustpart3
{
    8 references
    internal class clsUserDAO : DatabaseAccessObject
    {
        2 references
        public List<User> GetAllUsers()
        {
            List<User> lcUsers = new List<User>();

            var all_accounts = MySqlHelper.ExecuteDataset(MySqlConnection, "call all_accounts()");
            lcUsers = (from aResult in System.Data.DataTableExtensions.AsEnumerable(all_accounts.Tables[0])
            select
                new User
                {
                username = Convert.ToString(aResult["username"]),
                email = Convert.ToString(aResult["email"]),
                password = Convert.ToString(aResult["password"]),
                firstName = Convert.ToString(aResult["firstName"]),
                accountType = Convert.ToString(aResult["accountType"]),
                status = Convert.ToString(aResult["status"]),
                attempts = Convert.ToInt32(aResult["attempts"])
                }).ToList();

            return lcUsers;
        }
```

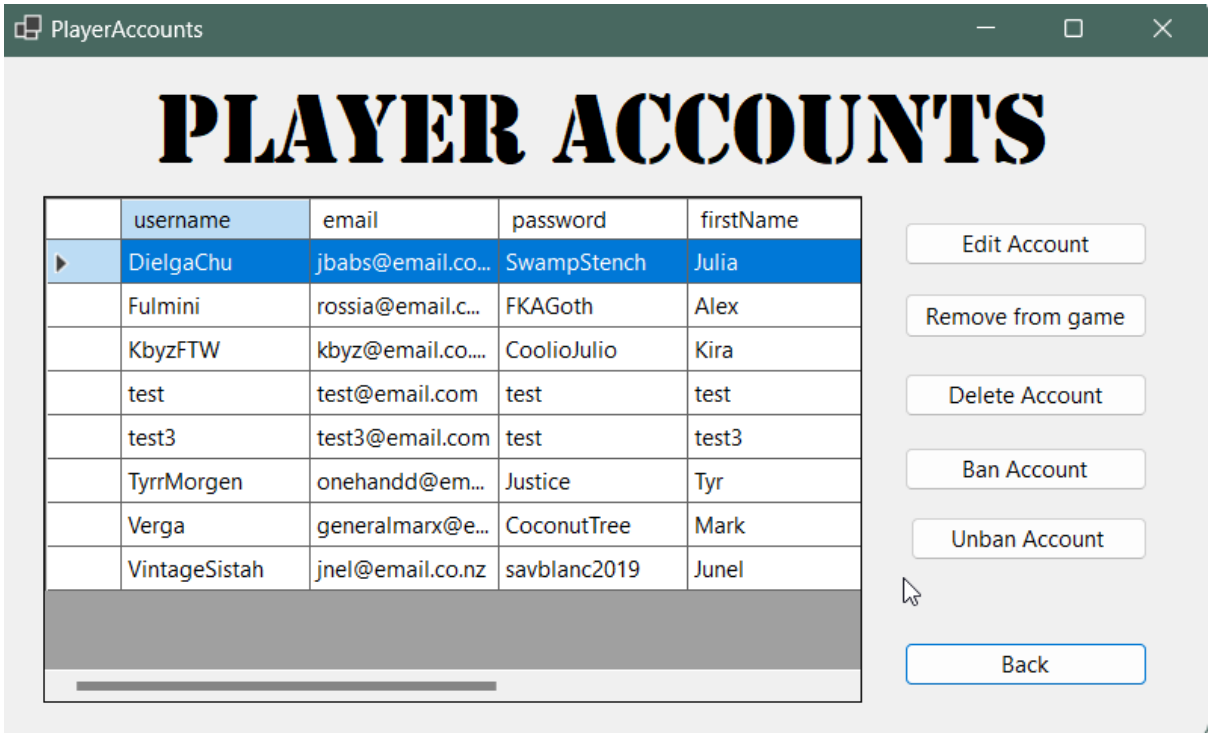
Application GUI



1 Administrator access is accessed through the 'admin settings' button in the logged in account menu



2 All player accounts can be viewed via the 'player accounts' button in the admin menu



3 The user_account table in the database is displayed in a data grid view in the GUI with row selection abilities

Ban/Unban Accounts

In the accounts data grid view GUI, the admin can select a row to manipulate it. An account can be banned with the 'ban account' button which will call the MySQL **ban_account** procedure through the AdminDAO. This will set take the **username** as the parameter and update the **status** field in the **user_account** table to 'Locked' meaning that they will not be able to log in until the account is unbanned.

Unbanning an account with the 'unban account' button calls the **unban_account** procedure through the AdminDAO which takes the **username** as a parameter and updates the **status** field in the **user_account** table to 'Offline' meaning they will be able to attempt logging in again.

MySQL Procedures

```
DROP PROCEDURE IF EXISTS ban_account;

DELIMITER $$

CREATE PROCEDURE ban_account (IN `username_para` VARCHAR(50))
COMMENT 'Lock player account.'
BEGIN
    -- Update the status field to  'Locked' preventing any form of login attempt
    UPDATE user_account ua
    SET ua.status = 'Locked'
    WHERE ua.username = `username_para`;

    SELECT 'Account banned!' AS MESSAGE_TEXT;

COMMIT;

END $$

DELIMITER ;
```

1 Ban Account procedure

```
DROP PROCEDURE IF EXISTS unban_account;

DELIMITER $$

CREATE PROCEDURE unban_account(IN `username_para` VARCHAR (50))
BEGIN
    -- Update account status to 'Logged out' to allow login attempts
    UPDATE `user_account`
    SET `status` = 'Logged Out', `attempts` = 0
    WHERE `username` = `username_para`;

    SELECT 'Account unbanned!' AS MESSAGE_TEXT;
COMMIT;

END $$

DELIMITER ;
```

2 Unban account procedure

C# DAO

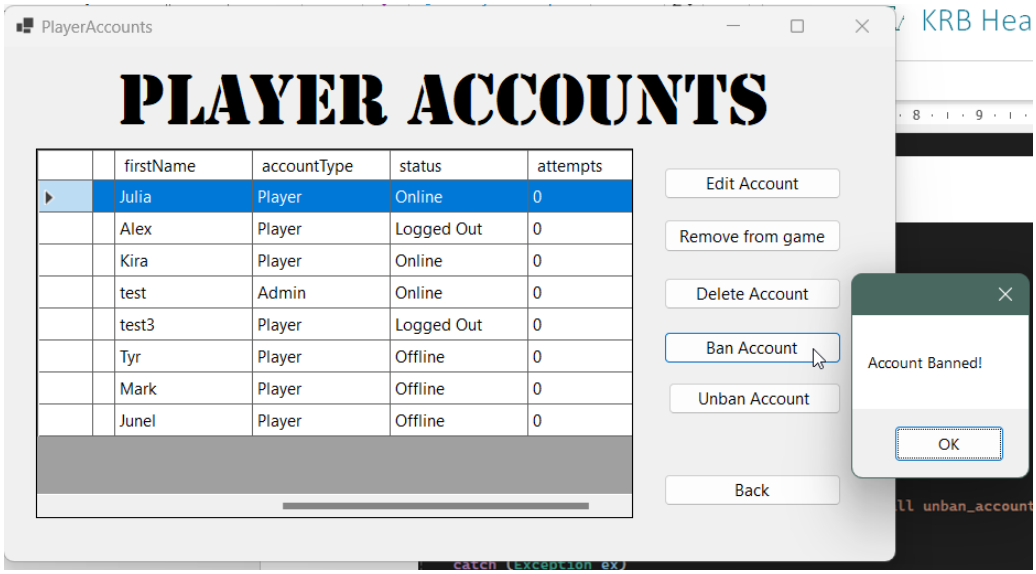
```
public void BanAccount(string username)
{
    try
    {
        List<MySqlParameter> procedure_params = new()
        {
            new()
            {

                ParameterName = "@username",
                MySqlDbType = MySqlDbType.VarChar,
                Value = username
            }
        };
        MySqlHelper.ExecuteDataset(MySqlConnection, "call ban_account(@username)", procedure_params.ToArray());

    }
    catch (Exception ex)
    {
        throw ex;
    }

}
```

3 Ban account DAO

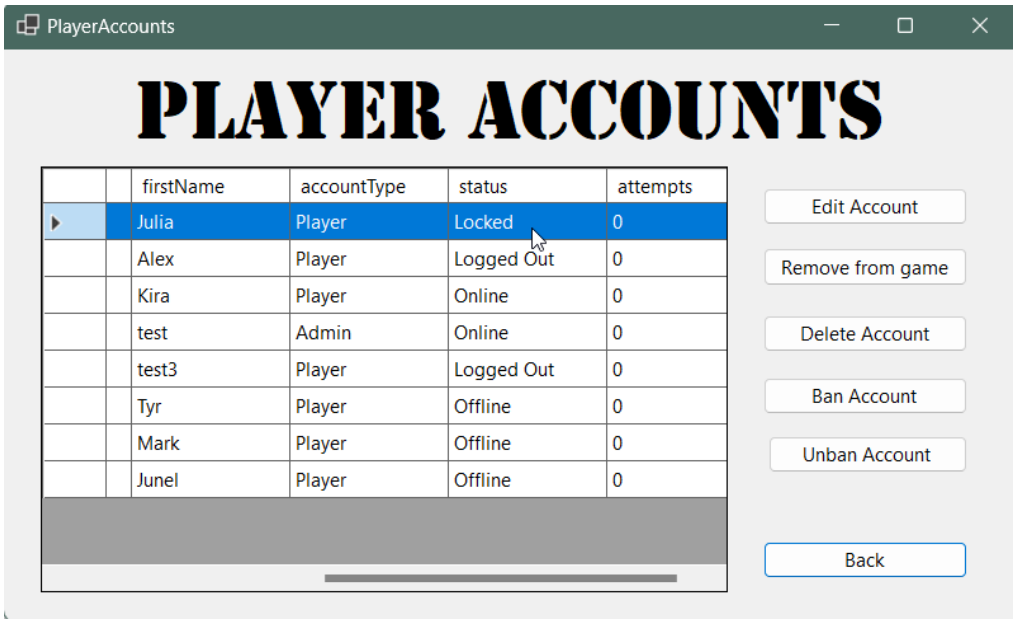
```
public void UnbanAccount(string username)
{
    try
    {
        List<MySqlParameter> procedure_params = new()
        {
            new()
            {

                ParameterName = "@username",
                MySqlDbType = MySqlDbType.VarChar,
                Value = username
            }
        };
        MySqlHelper.ExecuteDataset(MySqlConnection, "call unban_account(@username)", procedure_params.ToArray());

    }
    catch (Exception ex)
    {
        throw ex;
    }

}
```

4 Unban account DAO

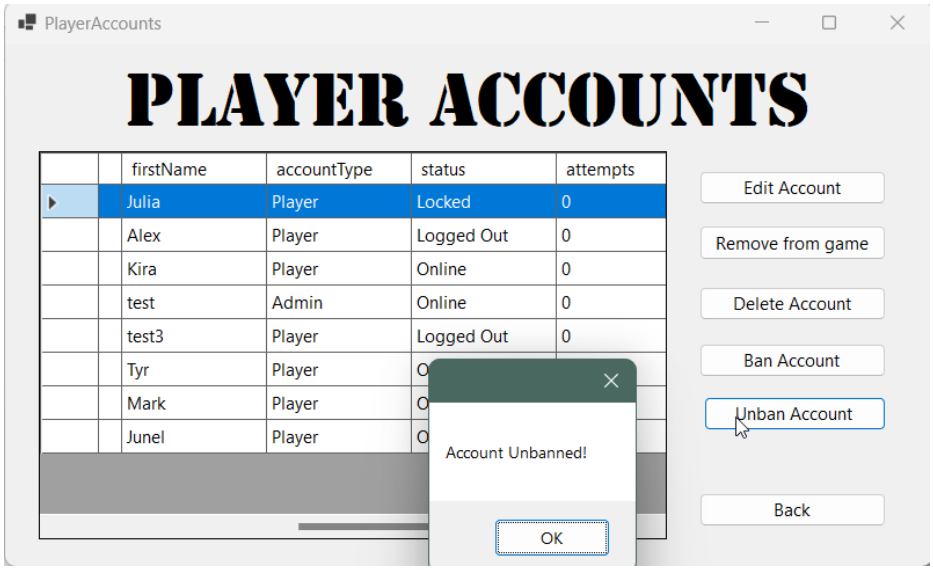
Application GUI



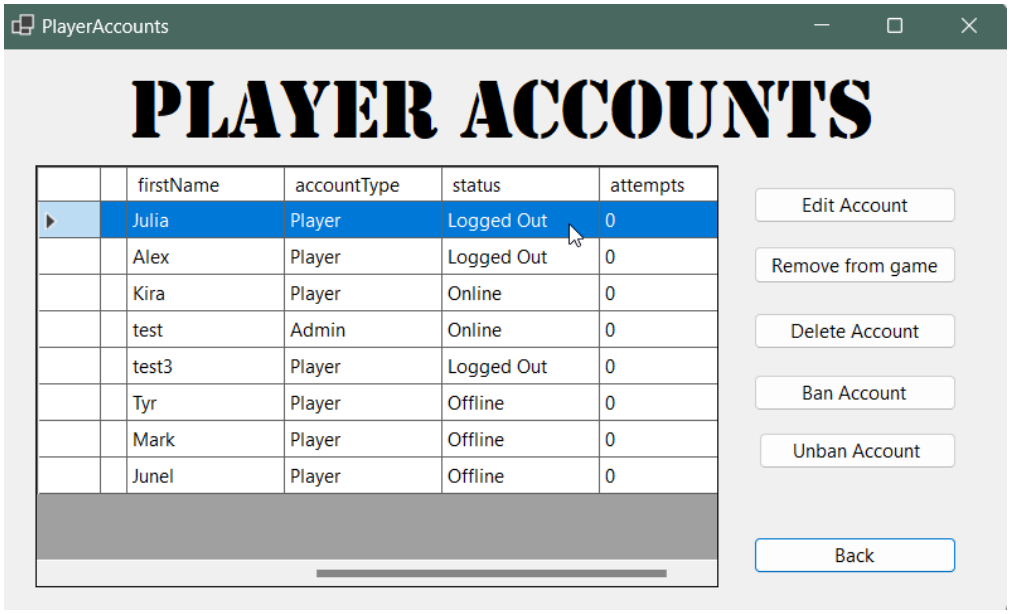
5 BanAccount method called via the Ban account button



6 Player status is updated to 'Locked' in the data grid view and database



7 UnbanAccount method called via the 'unban account' button in the GUI



8 Account status updated to 'Logged out'

Deleting Accounts

Within the accounts data grid view in the C# application GUI, an admin can select the row containing a user account and delete the account using the 'delete account' button. This button will pass the selected **username** as a parameter through the **AminDAO** which calls the MySQL procedure **delete_account** that will delete a whole record from the **user_account** table in the database.

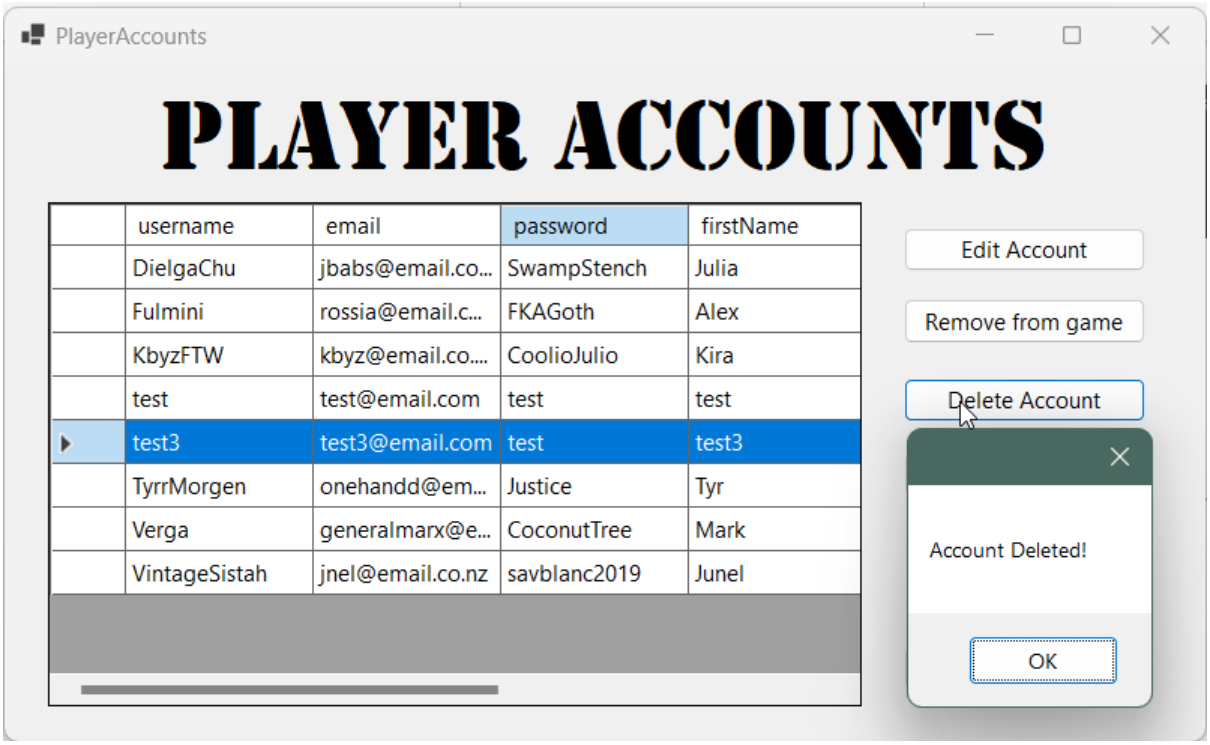
MySQL Procedure

```
DROP PROCEDURE IF EXISTS delete_account;

DELIMITER $$

CREATE PROCEDURE delete_account(IN `username_para` VARCHAR(50))
COMMENT 'Delete user account.'

BEGIN

    DELETE
    FROM user_account ua
    WHERE ua.username = `username_para`;

    SELECT 'Account deleted!' AS MESSAGE_TEXT;

COMMIT;

END $$

DELIMITER ;
```

C# DAO

```
namespace dust2dustpart3
{
    8 references
    internal class AdminDAO : DatabaseAccessObject
    {
        1 reference
        public void DeleteAccount(string username)
        {
            try
            {
                List<MySqlParameter> procedure_params = new()
                {
                    new()
                    {

                        ParameterName = "@username",
                        MySqlDbType = MySqlDbType.VarChar,
                        Value = username
                    }
                };

                MySqlHelper.ExecuteDataset(MySqlConnection, "call delete_account(@username)", procedure_params.ToArray());
            }
            catch (Exception ex)
            {
                throw ex;
            }
```

Application GUI



1 User account is selected by row in the data grid view. The DeleteAccount method is called via clicking the 'delete account' button, calling the delete_account procedure from the database.

Seeing Active Games & Players

From the admin menu GUI, the admin can select the 'Active Games' button to view the active games data grid view. This data grid view in the GUI called the **active_games_and_players** procedure through the **GamesDAO**. This procedure selects all **gameID**s from the **game** table where the **status** field is 'Active' and joins the corresponding **usernames** from the **character** table where the **gameID** exists as a foreign key.

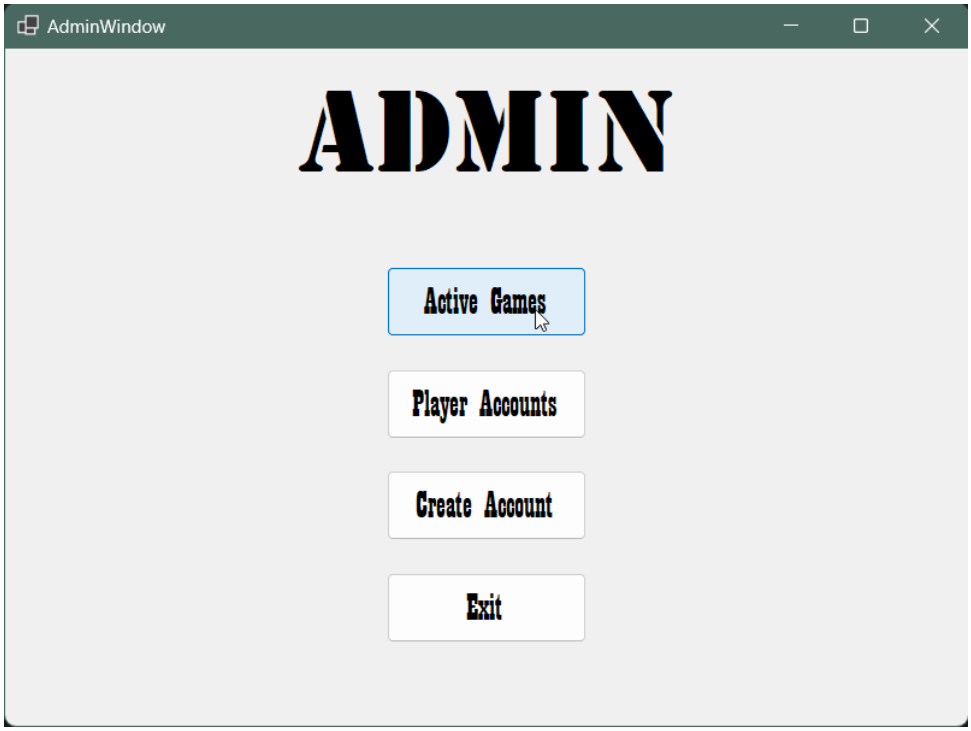
MySQL Procedure

```
DROP PROCEDURE IF EXISTS get_game_and_players;

DELIMITER $$

CREATE PROCEDURE get_game_and_players()
COMMENT 'Return all active games with associated active players.'
BEGIN
    SELECT g.gameID, c.username
    FROM game g
        INNER JOIN `character` c
        ON g.gameID = c.gameID
        AND c.gameID IS NOT NULL
        AND g.status = 'Active';

COMMIT;

END $$

DELIMITER ;
```

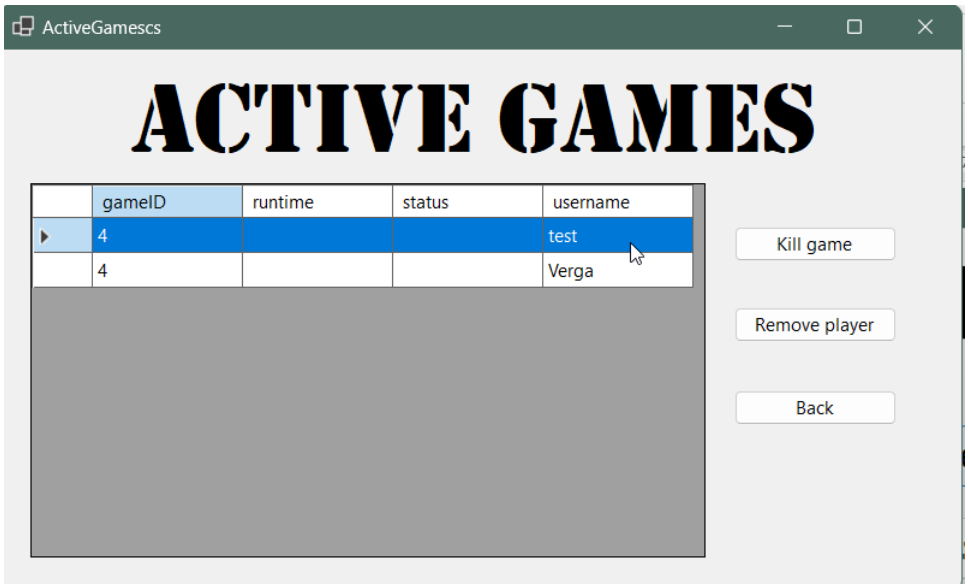
C# DAO

```
2 references
public static List<Game> GamesAndPlayers()
{
    List<Game> lcGames = new List<Game>();

    try
    {

        var games_and_players = MySqlHelper.ExecuteDataset(DatabaseAccessObject.MySqlConnection, "call get_game_and_players()");

        if (games_and_players.Tables.Count > 0 && games_and_players.Tables[0].Rows.Count > 0)
        {
            lcGames = (from aResult in System.Data.DataTableExtensions.AsEnumerable(games_and_players.Tables[0])
                       select new Game
                       {
                           gameID = Convert.ToInt32(aResult["gameID"]),
                           username = Convert.ToString(aResult["username"])
                       }).ToList();

        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Database Error!: " + ex.Message);
        throw;
    }
    return lcGames;
}
```

Application GUI



1 Active games and players list accessed via the 'active games' button in the admin menu



2 The game table and character table are joined and filtered by 'active' status input displayed in a data grid view

Killing active games

An admin can select a row in the data grid view in the C# application GUI and 'kill' the game via the 'Kill game' button. This button will call the **kill_game** procedure through the AdminDAO. This procedure takes the **gameID** as the parameter and deletes it. The **gameID** is created to delete on cascade, meaning all tiles and maps associated with it in the **tile** and **map** tables will be deleted also. The procedure then does an update on the joining **character** table that updates the **status** field of all

players that had the deleted **gameID** in their **gameID** field from ‘In-game’ to ‘Logged-in’ meaning they are no longer playing the game.

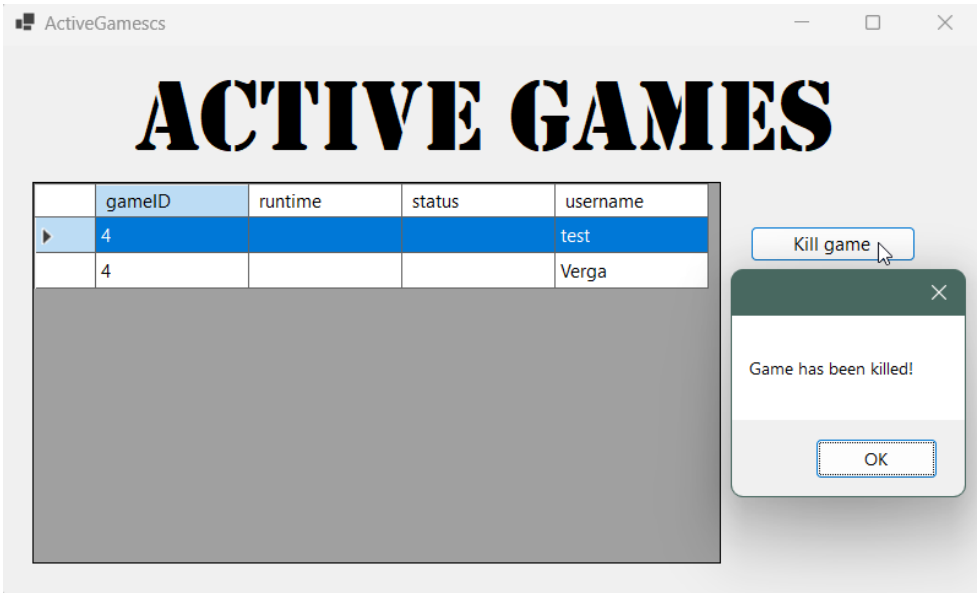
MySQL Procedure

```
DELIMITER $$

CREATE PROCEDURE kill_game (IN `gameID_para` INT)
BEGIN
    -- Deleting the gameID from the associated mapID
    DELETE FROM `map`
    WHERE `gameID` = `gameID_para`;

    -- Update the status of the game to 'offline' to prevent any active players from joining
    UPDATE `game`
    SET `status` = 'Offline'
    WHERE `gameID` = `gameID_para`;

    -- Update all associated characters in that game to disassociate them with the gameID and set their status to 'Logged in'
    UPDATE `character` c
    JOIN `game` g
        ON c.`gameID` = g.`gameID`
    SET c.`status` = 'Logged in', c.`gameID` = NULL
    WHERE g.`status` = 'Offline'
    AND g.`gameID` = `gameID_para`;

    SELECT 'Game killed!' AS MESSAGE_TEXT;

COMMIT;

END $$
```

C# DAO

```
1 reference
public void KillGame(int? gameID_para)
{
    try
    {
        List<MySqlParameter> procedure_params = new()
        {
            new()
            {

                ParameterName = "@gameID",
                MySqlDbType = MySqlDbType.Int32,
                Value = gameID_para
            }
        };

        MySqlHelper.ExecuteDataset(MySqlConnection, "call kill_game(@gameID)", procedure_params.ToArray());
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Application GUI



1 GameID selected by row in data grid view. KillGame method called via the 'kill game' button in the GUI, passing the kill_game call from the database

Removing a Player from a Game

The MySQL procedure **remove_player** updates the record of a given username within the **username** parameter to nullify any associated **gameID** and update their **status** to 'Logged in', stating that they are online in the application but not in an active game.

MySQL

```
DROP PROCEDURE IF EXISTS remove_player;

DELIMITER $$

CREATE PROCEDURE remove_player(IN `username_para` VARCHAR (50))
COMMENT 'Removes an active player from an active game at an admin level'
BEGIN

    IF
    (`username_para` = NULL OR `username_para` = '') THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Fields were null!';
    END IF;

    UPDATE `tile`
    SET `username` = NULL
    WHERE `username` = `username_para`;

    UPDATE `character`
    SET `gameID` = NULL, `status` = 'Offline'
    WHERE `gameID` IS NOT NULL
    AND `username` = `username_para`;

    UPDATE `user_account`
    SET `status` = 'Logged-in'

        SELECT 'Player removed!' AS MESSAGE;

    COMMIT;

    END $$

    DELIMITER ;
```

Test

```
1112 •  -- TEST REMOVE PLAYER FROM GAME PROCEDURE
1113    -- Reutrn an active game and list of active players
1114    CALL get_game_and_players;
1115
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

gameID	username
5	Fulmini
5	test
5	Verga

1 Call all active games and players to select an active player to remove

```
1116    -- Remove an active player
1117 •  CALL remove_player('test');
1118
```

Result Grid | Filter Rows:

	gameID	username
▶	5	Fulmini
	5	Verga

2 Call remove_player procedure with a selected username for the parameter. Call back the active games list procedure to see the update.

REFERENCES

IBM. (2024). *DROP TABLE Statement*. https://www.ibm.com/docs/en/informix-servers/12.10?topic=statements-drop-table-statement

Ravikiran, A. S. (2022, July 8). *Composite key in SQL: Your ultimate guide to mastery [Updated]*. Simplilearn. https://www.simplilearn.com/tutorials/sql-tutorial/composite-key-in-sql

W3 Schools. (2024). *SQL UNIQUE constraint*. W3Schools. https://www.w3schools.com/sql/sql_unique.asp