Project Title: Infinite Dungeons

Team Name: Null

Team member Names: Nathan Frazier

Andrew Truong

Abstract:

The database application we plan to make is a simple game with a simple UI that allows players to fight creatures that drop weapons or gold when killed. Players will also be able to buy into dungeons, that have harder enemies and better rewards. There will also be admins that can add new creatures, new weapons, modify weapons and creatures, as well as directly give players gold/weapons. There will also be two sets of logs, admin logs and kill logs, these will store data about events that the admins can look at.

Mission Statement:

The purpose of the Infinite Dungeons database system is to maintain the data on every player, including their login information and various statistics about their character they control in game, and maintain data on admins, who can directly modify data on enemies and weapons that impact the gameplay of all players.

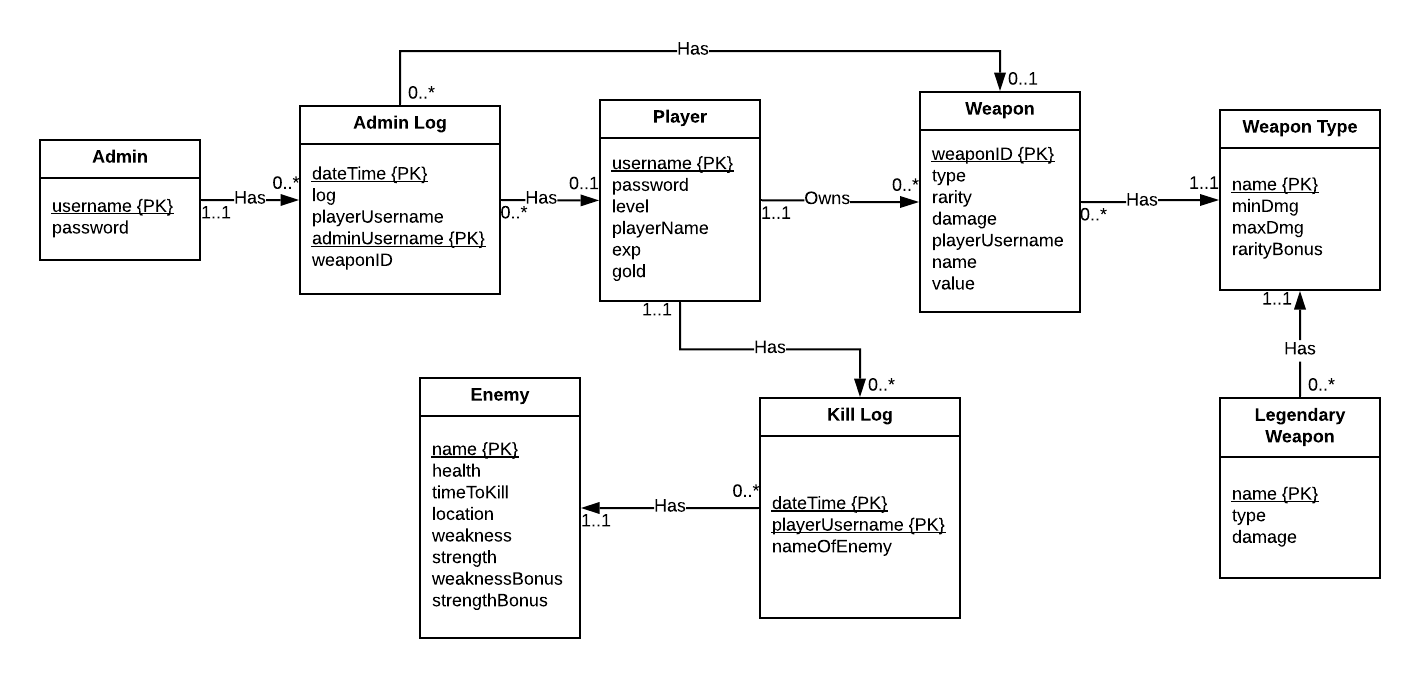
Mission Objectives: (Note: weapons and legendary weapons work completely different from eachother)

* To maintain (enter) data on admins
* To maintain (enter and update) data on players
* To maintain (enter, update, and delete) data on weapons
* To maintain (enter, update, and delete) data on enemies
* To maintain (enter, update, and delete) data on admin logs
* To maintain (enter, update, and delete) data on kill logs
* To maintain (enter, update, and delete) data on weapon types
* To maintain (enter, update, and delete) data on legendary weapons
* To perform searches on enemies
* To perform searches on kill logs
* To perform searches on admin logs
* To perform searches on weapons
* To perform searches on players
* To perform searches on weapon types
* To perform searches on legendary weapons
* To track the status of players
* To report on admins
* To report on players
* To report on weapons
* To report on enemies
* To report on admin logs
* To report on kill logs
* To report on weapon types
* To report on legendary weapons

Major User Views

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Access Type | Player | Admin |
| All Players | Maintain |  | X |
|  | Query |  | X |
|  | Report |  | X |
| Own Player | Maintain | X |  |
|  | Query | X |  |
|  | Report | X |  |
| Admin | Maintain |  | X |
|  | Query |  |  |
|  | Report |  |  |
| Weapon | Maintain |  | X |
|  | Query |  | X |
|  | Report |  | X |
| Own Weapon | Maintain | X |  |
|  | Query | X |  |
|  | Report | X |  |
| Enemy | Maintain |  | X |
|  | Query | X | X |
|  | Report |  | X |
| Admin Log | Maintain |  | X |
|  | Query |  | X |
|  | Report |  | X |
| Kill Log | Maintain | X | X |
|  | Query | X | X |
|  | Report | X | X |
| Weapon Type | Maintain |  | X |
|  | Query | X |  |
|  | Report |  | X |
| Legendary Weapon | Maintain |  | X |
|  | Query | X | X |
|  | Report |  | X |

ER Diagram:



Relational Model and BCNF verification:

**DOMAIN** Username **AS VARCHAR** (20)

**CHECK** (**LEN(USERNAME)** > 2)

**DOMAIN** Password **AS VARCHAR** (30)

**CHECK** (**LEN(PASSWORD)** > 2)

**DOMAIN** Damage **AS INTEGER** (6)

**CHECK** (**VALUE** > 0)

**DOMAIN** Bonus **AS DECIMAL** (3, 2)

**TABLE** Admin

username Username **NOT NULL**,

password Password **NOT NULL**,

**PRIMARY KEY** (username)

1NF: Both Username and Password attributes can only have one value in each cell, as every Admin has only one Username and Password.

2NF: Primary Key is only one attribute, so 2NF is guaranteed.

3NF: There is only one non-primary key, thus the table must be in 3NF.

BCNF: Username does not depend on Password, as many Admins, and thus many Usernames, could, in theory, have the same password. So knowing the password does not allow you to know the username. So the admin table is in BCNF.

**TABLE** Admin Log

dateTime DATETIME **NOT NULL**,

log CHAR (120) **NOT NULL**,

playerUsername Username,

adminUsername Username **NOT NULL**,

weaponID INTEGER,

**PRIMARY KEY** (dateTime, adminUsername),

**FOREIGN KEY** (adminUsername) **REFERENCES** Admin (username),

**FOREIGN KEY** (playerUsername) **REFERENCES** Player (username),

**FOREIGN KEY** (weaponID) **REFERENCES** Weapon (weaponID),

1NF: For every admin log, there is exactly one dateTime in which the admin log was made, and only one log can be put in every row. For each log, only one admin could of made it, and only one player could have been affected and only one weapon could have been given (if one was given at all). Thus, no cell can have more than one item, and the Admin Log table is in 1NF.

2NF: The Log, playerUsername and weaponID can not be determined by the adminUsername alone, as every admin can have multiple logs, affected different players and weapons. These three attributes also can not be determined by datetime along, due to multiple different admins creating completely different logs at the same time.

3NF: The Log can not be determined by either the playerUsername or weaponID, due to the fact that you can have different logs even if you are affecting the same Player, and can have different logs even if you are affecting the same weapon. Similarly, knowing the log does not tell you which player or weapon was affected. Since any weapon can be given to any player, the playerUsername can not tell you what weaponID was given, and the weaponID cannot tell you what the playerUsername will be. Thus, AdminLog table is in 3NF.

BCNF: dateTime is not dependent on log, playerUsername or weaponID, due to all of these items being able to be in the table at different times, so simply knowing the log, playerUsername, or weaponID, cannot tell you the time they were added to the table. Similarly, The log, playerUsername, and weaponID could of also been added by different admins, so none of attributes could tell you what adminUsername should be in the table. This makes the AdminLog table in BCNF.

**TABLE** Player

username Username **NOT NULL**,

password Password **NOT NULL**,

level INTEGER **NOT NULL** **DEFAULT** ‘1’,

playerName VARCHAR (15) **NOT NULL**,

exp INTEGER **NOT NULL**,

gold INTEGER **NOT NULL**

**PRIMARY KEY** (username)

1NF: Every player can only have one username, password, level, playerName, exp, and gold count. Since each player can only have of these attributes, each row-column cell will have exactly one value. And the player table is in 1NF.

2NF: Since there is only one primary key attribute, the Player table is in 2NF.

3NF: Since each player has their own level, exp, and gold, and many players can have the same password and playerName, none of these attributes can be determined by the password or playerName, and they can not be determined by each other, as they are completely independent values. These facts make the Player table in 3NF.

BCNF: Since multiple players can have the same password, playerName, level, exp, and gold, none of these attributes alone, or even as a combination, can determine the username. Which makes the Player table in BCNF.

**TABLE** Weapon

weaponID INTEGER **NOT NULL**,

type VARCHAR (20) **NOT NULL**,

rarity VARCHAR (20) **NOT NULL**,

damage Damage **NOT NULL**,

playerUsername Username **NOT NULL**,

name varchar (20) **NOT NULL**,

value INTEGER **NOT NULL,**

**PRIMARY KEY** (weaponID),

**FOREIGN KEY** (playerUsername) **REFERENCES** Player (username)

1NF: Every weapon can have only one WeaponID, type, rarity, damage, playerUsername, name, and value, thus every row-column cell can only have one value, which makes the weapon table in 1NF.

2NF: Since there is only one primary key attribute, the weapon table is in 2NF.

3NF: Knowing the type, rarity, damage, playerUsername, name, or value of the weapon can not help to determine any of the other attributes. Each of these values are not unique and will mix and match heavily over many rows.

BCNF: Since all of the attributes can be the same for a completely different weapon, with a completely different weaponID, these attributes can not be used to determine the weaponID. This makes the weapon table in BCNF.

**TABLE** Weapon Type

name VARCHAR (20) **NOT NULL**,

minDmg Damage **NOT NULL**,

maxDmg Damage **NOT NULL**,

rarityBonus Bonus **NOT NULL DEFAULT** ‘1.50’,

**PRIMARY KEY** (name)

1NF: Each WeaponType can have only one name, and only one minDmg and maxDmg and exactly one rarityBonus. This means each cell will have only one value, which makes the WeaponType table 1NF.

2NF: Since there is only one primary key attribute for this table, it is in 2NF.

3NF: minDmg and maxDmg can not be used to determine rarityBonus, and similarly, minDmg can not be determined by maxDmg or rarityBonus, and maxDmg can not be determined by maxDmg or rarityBonus. This means the WeaponType table is in 3NF.

BCNF: the name attribute can not be determined by minDmg, maxDmg, or rarityBonus due to many different weapon types having different names with the same minDmg, maxDmg, and rarityBonus. This makes the WeaponType in BCNF.

**TABLE** Legendary Weapon

name VAR CHAR (20) **NOT NULL**,

type VAR CHAR (20) **NOT NULL**,

damage Damage **NOT NULL**,

**PRIMARY KEY** (name)

**FOREIGN KEY** (type) **REFERENCES** Weapon Type (name),

1NF: Each Legendary Weapon can only have one name, type, and damage, so each cell will only have one value in it, which makes the LegendaryWeapon table in 1NF.

2NF: Since the LegendaryWeapon table only has one primary key attribute, this table is in 2NF.

3NF: Knowing the type attribute cannot determine what the damage will be due to two weapons having the same type can have different damages. Similarly, knowing the damage cannot determine the type, due to two weapons having the same damage but different types.

BCNF: Knowing the type or damage will not tell you the name, as two weapons can have different names, but the same type and damage. This means the LegendaryWeapon table is in BCNF.

**TABLE** Enemy

name varchar (20) **NOT NULL**,

health INTEGER **NOT NULL**,

timeToKill INTEGER **NOT NULL**,

location VARCHAR (20),

weakness VARCHAR(15),

strength VARCHAR (15),

weaknessBonus Bonus,

strengthBonus Bonus,

**PRIMARY KEY** (name)

1NF: Each enemy can have only one name, health value, timeToKill value, location, weakness, strength, and weakness and strength Bonuses. This means each cell will only have one value, which makes the Enemy table in 1NF.

2NF: Since the Enemy table only has one primary key attribute, the Enemy table is in 2NF.

3NF: Health, timeToKill, location, weakness, strength, weaknessBonus, and stengthBonus are heavily mixed and matched between rows. None of these attributes can determine any of the other attributes or combination of attributes. Which makes the Enemy table 3NF.

BCNF: Health, timeToKill, location, weakness, strength, weaknessBonus, and stengthBonus can all be the same for many different names of enemies, thus the enemy attribute is not dependent on any of these attributes, or combination of these attributes. This makes the Enemy table in BCNF.

**TABLE** Kill Log

dateTime DATETIME **NOT NULL**,

playerUsername Username **NOT NULL**,

nameOfEnemy varchar (20) **NOT NULL**,

**PRIMARY KEY** (dateTime, playerUsername),

**FOREIGN KEY** (nameOfEnemy) **REFERENCES** Enemy (name),

**FOREIGN KEY** (playerUsername) **REFERENCES** Player (name)

1NF: Each Kill Log can only have one playerUsername, and nameOfEnemy, and each log can only have been made at a single dateTime, which means each cell will have exactly one value, and the KillLog table is in 1NF.

2NF: Many different enemies can be killed at the same time, and many different enemies can be killed by the same playerUsername, only if you know the playerUsername and dateTime will there be exactly one nameOfEnemy, which makes the KillLog table in 2NF.

3NF: There is only one non-primary attribute, which makes the KillLog table in 3NF.

BCNF: Neither dateTime or PlayerUsername can be determined by nameOfEnemy, due to many different players killing the same enemy at the same time or different time. This makes the KillLog table BCNF.

**Actors:** Admin, Player

**Use Cases:**

**Entity: Admin**

Use case name: Create Admin

Actor: Admin

Steps:

1. User clicks “Create Admin” button
2. Prompt user to enter username and password
3. User clicks “Register” button

SQL statement:

INSERT INTO Admin (username, password)

VALUES (@username, @password);

Explanation: Adds admin login information into the Admin table

Use case name: Delete Admin

Actor: Admin

Steps:

1. User clicks “View Admins” button
2. Each admin username is displayed
3. User types in an admin username
4. User presses the “Delete” Button
5. The admin whose username matches the field is removed from database

SQL statement:

DELETE FROM Admin

WHERE username = @username;

Explanation: Removes a specific admin from the Admin table

Use case name: View Admin Count

Actor: Admin

Steps:

1. User clicks “View Admins” button
2. Each admin username is displayed as well as the number of admins being displayed

SQL statement:

SELECT COUNT(\*)

FROM Admin;

Explanation: Shows the number of admins in the Admin table

Use case name: Update Admin Password

Actor: Admin

Steps:

1. User clicks “Change Password” button
2. Prompt user to type in new password
3. User types in new password
4. User clicks “Update Password” button
5. Password is changed for that admin account in the database

SQL statement:

UPDATE Admin

SET password = @password

WHERE username = @username;

Explanation: Changes the password attribute for the current user in the Admin table

User case name: Show Complete Admin Logs

Actor: Admin

Steps:

1. User clicks “View Admin Logs” button
2. User is shown admin logs
3. User clicks “View Complete Admin Logs” button
4. User is shown all logs, with all their info, along with admin’s password and player name in log and the weapons name (if applicable)

SQL statement:

SELECT Admin.username adminUsername, Admin.[password] adminPassword, dateTime, log, Player.username playerUsername, playerName, AdminLog.weaponID, Weapon.name

FROM Admin, AdminLog, Player, Weapon

WHERE Admin.username = AdminLog.adminUsername AND Player.username = AdminLog.playerUsername AND Weapon.weaponID = AdminLog.WeaponID;

Explanation: Joint Query between Admin and AdminLog tables to show more info in the logs

**Entity: AdminLog**

Use case name: Create Admin Log

Actor: Admin

Steps:

1. User clicks “Player Options” button
2. User clicks “Give Weapon” button
3. Prompt user to enter weapon name, type, damage, rarity and a Player username
4. That weapon is added to the database with the username entered as the owner
5. An Admin log is created to keep track of this action

SQL statement:

INSERT INTO AdminLog (dateTime, adminUsername, playerUsername, log, weaponID)

VALUES (CURRENT\_TIMESTAMP, @adminUsername, @playerUsername,

@log(ex. ‘exampleAdmin gave examplePlayer a weapon with id @weaponID.’), @weaponID);

Explanation: Creates a log of an admins Action and stores it inside of AdminLog table

Use case name: Delete logs of certain Admin

Actor: Admin

Steps:

1. User clicks “View Admin Logs” button
2. User is shown admin logs
3. User clicks “Delete Logs” button
4. User is prompted to enter a username
5. User enters an admin’s username
6. User clicks “Delete” Button
7. All admin logs related to that username are deleted

SQL statement:

DELETE FROM AdminLog

WHERE adminUsername = @adminUsername;

Explanation: Deletes all logs related to a specific adminUsername in AdminLog table

Use case name: View AdminLog Count

Actor: Admin

Steps:

1. User clicks “View Admins Logs” button
2. User is shown admin logs as well as the total number of admin logs in database

SQL statement:

SELECT COUNT(\*)

FROM AdminLog;

Explanation: Shows the number of logs in the AdminLog table

Use case name: Remove Player from an AdminLog

Actor: Admin

Steps:

1. User clicks “View Admin Logs” button
2. User is shown admin logs
3. User clicks “Remove Player” button
4. User types in a username
5. User clicks “Remove” button
6. That playerUsername is removed from all logs it is in.

SQL statement:

UPDATE AdminLog

SET playerUsername = NULL

WHERE playerUsername = @playerUsername;

Explanation: Removes Player from all AdminLogs where he is referenced

**Entity: Player**

Use case name: Create Player

Actor: Player

Steps:

1. User clicks “Register” button
2. Prompt user to enter username and password and a player name
3. User clicks new “Register” button
4. New player is added to the database

SQL statement:

INSERT INTO Player (username, password, level, playerName, exp, gold)

VALUES (@username, @password, 1, @playerName, 0, 100);

Explanation: Creates a new Player and adds them to the Player table

Use case name: Get number of Players and total amount of gold and average level

Actor: Admin

Steps:

1. User clicks “Player Information” button
2. User clicks “View All Player Statistics” button
3. User is shown total number of players and their combined gold amount and average level

SQL statement:

SELECT COUNT(\*), SUM(gold), AVG(level)

FROM Player;

Explanation: Returns the total number of players, the sum of all gold, and the average level of players from Player table

Use case name: Player receives gold and experience

Actor: Player

Steps:

1. User clicks “Attack” button
2. Enemy’s health bar goes down
3. Repeat step 1 and 2 until enemy’s health bar is empty
4. User is informed of rewards from the kill
5. Gold and experience are added to the player’s total gold and experience

SQL statement:

UPDATE Player

SET gold = gold + @goldInc, experience = experience = @experienceInc

WHERE username = @username;

Explanation: Adds gold and experience to a player in the Player table when a monster is killed

Use case name: Delete Player

Actor: Admin

Steps:

1. User clicks “Player Options” button
2. User clicks “Delete Player” button
3. User enters a username
4. User is prompted with a confirmation
5. User clicks “Yes” to confirmation
6. Player is deleted from Player table

SQL statement:

DELETE FROM Player

WHERE username = @username;

Explanation: Lets admin delete a Player from the Player table

Use case name: View Weapons in Game

Actor: Admin

Steps:

1. User clicks “Player Information” button
2. User clicks “All Owned Weapons” button
3. User is shown all owned weapons by players

SQL statement:

SELECT playerName, Weapon.name, damage, rarity, type, value, minDmg, maxDmg, rarityBonus

FROM Player, Weapon, WeaponType

WHERE Player.username = Weapon.playerUsername AND Weapon.type = WeaponType.name;

Explanation: Uses join operation to join Player and Weapon table to show relevant information about weapons that are in the current game

**Entity: KillLog**

Use case name: Create Kill Log

Actor: Player

Steps:

1. User clicks “Attack” button
2. Enemy’s health bar goes down
3. Repeat step 1 and 2 until enemy’s health bar is empty
4. User is informed of rewards from the kill
5. Kill log is created

SQL statement:

INSERT INTO KillLog (dateTime, playerUsername, nameOfEnemy)

VALUES (CURRENT\_TIMESTAMP, @playerUsername, @nameOfEnemy);

Explanation: Adds a Kill Log to the KillLog table whenever a player kills a monster

Use case name: Delete Kill Logs of Player

Actor: Admin

Steps:

1. User clicks “Player Information” button
2. User clicks “Kill Logs” button
3. User clicks “Delete Kill Logs” button
4. User enters a username
5. Kill logs associated with that username are deleted from database

SQL statement:

DELETE FROM KillLog

WHERE playerUsername = @playerUsername;

Explanation: Deletes all the kill logs of a certain player from KillLog table

Use case name: Change playerUsername in KillLog

Actor: Admin

Steps:

1. User clicks “Player Information” button
2. User clicks “Kill Logs” button
3. User clicks “Change Kill Logs” button
4. User enters a username to change, and a username to change it to
5. Kill logs associated with that username are changed to the new username in the database

SQL statement:

UPDATE KillLog

SET playerUsername = @newPlayerUsername

WHERE playerUsername = @oldPlayerUsername;

Explanation: Changes a username to a different username in KillLog table

Use case name: Get the number of times each enemy was killed and their location

Actor: Player

Steps:

1. User clicks “Statistics” button
2. User clicks “Enemy Statistics” button
3. User is shown total number of times they have killed each enemy

SQL statement:

SELECT playerName, nameOfEnemy, location, COUNT(nameOfEnemy) kills

FROM Player, KillLog, Enemy

WHERE Player.username = KillLog. playerUsername AND  
 KillLog.nameOfEnemy = Enemy.name

GROUP BY playerName, nameOfEnemy, location

ORDER BY COUNT(nameOfEnemy) DESC;

Explanation: Shows the player the number of times each enemy shows up in the KillLog table as well as showing their location.

**Entity: Enemy**

Use case name: Create new Enemy

Actor: Admin

Steps:

1. User clicks “Enemies” button
2. User is shown all current enemies in the database
3. User clicks “Add Enemy” button
4. User inserts name, health, timeToKill (in seconds), location, weakness, strength, weaknessBonus, and strengthBonus of the new enemy.
5. User clicks “Add” button
6. New enemy is added to the database

SQL statement:

INSERT INTO Enemy (name, health, timeToKill, location, weakness, strength, weaknessBonus. strengthBonus)

VALUES (@name,@ health, @timeToKill, @location, @weakness, @strength, @weaknessBonus, @strengthBonus);

Explanation: Adds a new enemy to the Enemy table

Use case name: Delete Enemy

Actor: Admin

Steps:

1. User clicks “Enemies” button
2. User is shown all current enemies in the database
3. User clicks “Delete Enemy” button
4. User enters the name of an enemy
5. User clicks “Delete” Button
6. Enemy is removed from the database

SQL statement:

DELETE FROM Enemy

WHERE name = @name;

Explanation: Removes a specific enemy from the Enemy table

Use case name: View Enemy Count

Actor: Admin

Steps:

1. User clicks “Enemies” button
2. User is shown all current enemies in the database and the number of enemies

SQL statement:

SELECT COUNT(\*)

FROM Enemy;

Explanation: Shows the number of enemies in the Enemy table

Use case name: Update Enemy information

Actor: Admin

Steps:

1. User clicks “Enemies” button
2. User is shown all current enemies in the database
3. User clicks “Update Enemy” button
4. User enters the name of an enemy
5. Users is shown fields with all the enemy information
6. User changes one or more fields
7. User clicks “Update” button

SQL statement:

UPDATE Enemy

SET health = @health, timeToKill = @timeToKill, location = @location, weakness = @weakness, strength = @strength, weaknessBonus = @weaknessBonus, strengthBonus = @strengthBonus;

WHERE name = @name;

Explanation: Changes one or more properties of an enemy

**Entity: Weapon**

Use case name: Weapon drop

Actor: Player

Steps:

1. User clicks “Attack” button
2. Enemy’s health bar goes down
3. Repeat step 1 and 2 until enemy’s health bar is empty
4. User is informed of a weapon being dropped from the enemy
5. Weapon is added to the database

SQL statement:

INSERT INTO Weapon

VALUES (@WeaponID, @type, @rarity, @damage, @playerUsername, @name, @value);

Explanation: Adds a new weapon to the Weapon table when a weapon is dropped from an enemy

Use case name: Sell Weapon

Actor: Player

Steps:

1. User clicks “View Weapons” button
2. User is shown all current weapons they own
3. User types in a weaponID to sell
4. User clicks “Sell”
5. Weapon is removed from the database

SQL statement:

DELETE FROM Weapon

WHERE weaponID = @weaponID;

Explanation: Removes a specific weapon from the Weapon table

Use case name: View Weapon Count and total worth

Actor: Player

Steps:

1. User clicks “View Weapons” button
2. User is shown all weapons they own as well as the number of weapons they own and their total worth

SQL statement:

SELECT COUNT(\*), SUM(value)

FROM Weapon;

Explanation: Shows the number of weapons associated with a player in the Weapon table as well as the sum of the value of those weapons

Use case name: Reroll weapon damage

Actor: Player

Steps:

1. User clicks “View Weapons” button
2. User is shown all current weapons they own
3. User clicks types in a WeaponID to reroll damage of
4. User clicks “Reroll Damage” button
5. Damage of the weapon is changed based on an algorithm inside program

SQL statement:

UPDATE Weapon

SET damage = @damage

WHERE weaponID = @weaponID;

Explanation: Changes the damage of a weapon in the Weapon table

**Entity: WeaponType**

Use case name: View Weapon Type

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database, including how many legendary weapons of each type there are

SQL statement:

SELECT WeaponType.name, minDmg, maxDmg, rarityBonus, COUNT (LegendaryWeapon.name) numberOfLegendaries

FROM WeaponType, LegendaryWeapon

WHERE WeaponType.name = LegendaryWeapon.type

GROUP BY WeaponType.name, minDmg, maxDmg, rarityBonus;

Explanation: Uses Joint operation to display all data in the WeaponType table, as well as the amount of legendary weapons of each type from the LegenadryWeapon table

Use case name: Create new Weapon Type

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Add Weapon Type” button
4. User inserts name, minDmg, maxDmg, and rarityBonus for the new weapon type
5. User clicks “Add” button
6. New weapon type is added to the database

SQL statement:

INSERT INTO WeaponType (name, minDmg, maxDmg, rarityBonus)

VALUES (@name, @minDmg, @maxDmg, @rarityBonus);

Explanation: Adds a new weapon type to the WeaponType table

Use case name: Delete Weapon Type

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Delete Weapon Type” button
4. User enters the name of the weapon type
5. User clicks “Delete” Button
6. Weapon type is removed from the database

SQL statement:

DELETE FROM WeaponType

WHERE name = @name;

Explanation: Removes a specific weapon type from the WeaponType table

Use case name: View the number of weapon types and the average min and max damage

Actor: Admin

Steps:

1. User clicks “WeaponType” button
2. User is shown all current weapon types in the database and the number of weapon types as well as the min and max damage averages

SQL statement:

SELECT COUNT(\*), AVG(minDmg), AVG(maxDmg)

FROM WeaponType;

Explanation: Shows the number of weapon types in the WeaponType table as well as averaging the minDmg and maxDmg attributes

Use case name: Update Weapon Type information

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Update Weapon Type” button
4. User enters the name of the weapon type to update
5. Users is shown fields with all the weapon type information
6. User changes one or more fields
7. User clicks “Update” button

SQL statement:

UPDATE WeaponType

SET minDmg = @minDmg, maxDmg = @maxDmg, rarityBonus = @rarityBonus;

WHERE name = @name;

Explanation: Changes one or more properties of a weapon type

**Entity: Legendary Weapon**

Use case name: Create new Legendary Weapon

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Legendary Weapons” button
4. User is shown all current legendary weapons in the database
5. User clicks on “Add Legendary Weapon” button
6. User inserts name, type, and damage for the new legendary weapon
7. User clicks “Add” button
8. New legendary weapon is added to the database

SQL statement:

INSERT INTO LegendaryWeapon (name, type, damage)

VALUES (@name, @type, @damage);

Explanation: Adds a new legendary weapon to the LegendaryWeapon table

Use case name: Delete Legendary Weapon

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Legendary Weapons” button
4. User is shown all current legendary weapons in the database
5. User clicks “Delete Legendary Weapon” button
6. User enters the name of the legendary weapon
7. User clicks “Delete” Button
8. Legendary weapon is removed from the database

SQL statement:

DELETE FROM LegendaryWeapon

WHERE name = @name;

Explanation: Removes a specific legendary weapon from the LegendaryWeapon table

Use case name: View the number of Legendary weapons and the average damage

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Legendary Weapons” button
4. User is shown all current legendary weapons in the database and the number of legendary weapons as well as the average damage

SQL statement:

SELECT COUNT(\*), AVG(damage)

FROM LegendaryWeapon;

Explanation: Shows the number of legendary weapons in the LegendaryWeapon table as well as averaging the damage attribute

Use case name: Update Legendary weapon damage

Actor: Admin

Steps:

1. User clicks “Weapon Types” button
2. User is shown all current weapon types in the database
3. User clicks “Legendary Weapons” button
4. User is shown all current legendary weapons in the database
5. User clicks “Update Legendary” button
6. User enters the name of the weapon type to update
7. Users is shown fields consisting of the Legendary weapon’s damage and type
8. User changes one or both fields
9. User clicks “Update” button

SQL statement:

UPDATE LegendaryWeapon

SET damage = @damage, type = @type

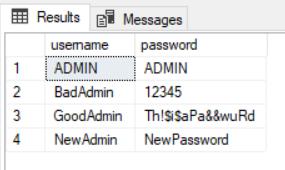
WHERE name = @name;

Explanation: Changes one or more properties of a legendary weapon

Test Records:

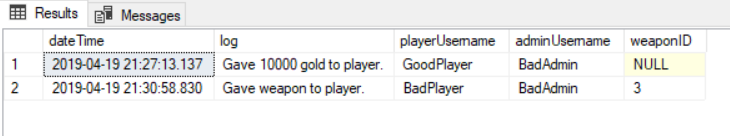
// Admin table information

SELECT \* FROM Admin;



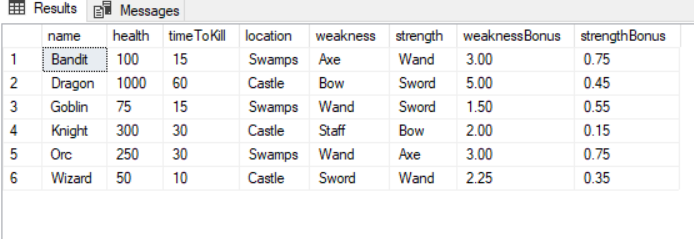
// AdminLog table information

SELECT \* FROM AdminLog;



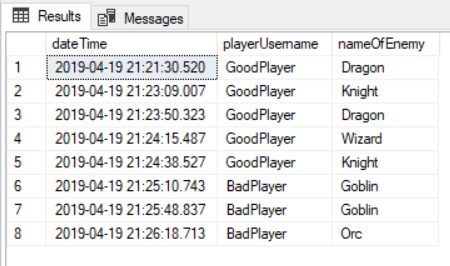
// Enemy table information

SELECT \* FROM Enemy;



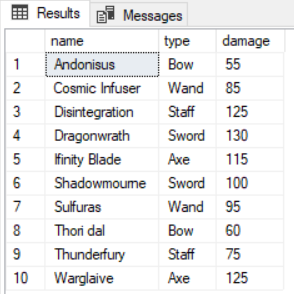
// KillLog table information

SELECT \* FROM KillLog;



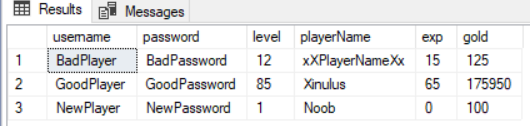
// LegendaryWeapon table information

SELECT \* FROM LegendaryWeapon;



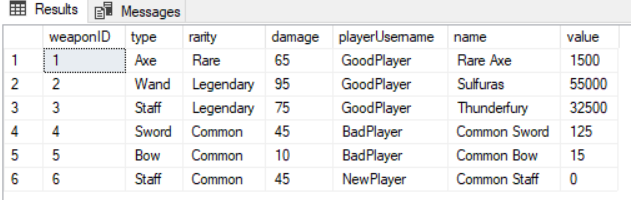
// Player table information

SELECT \* FROM Player;



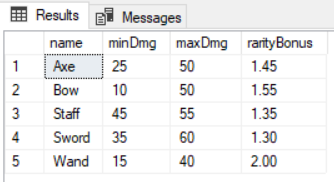
// Weapon table information

SELECT \* FROM Weapon;



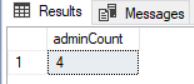
// WeaponType table information

SELECT \* FROM WeaponType;



// Shows number of Admins

SELECT COUNT(\*) AS adminCount FROM Admin;



// Shows number of admin logs of each admin

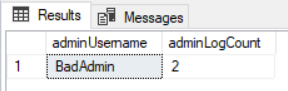
SELECT adminUsername,

COUNT(playerUsername) AS adminLogCount

FROM AdminLog

GROUP BY adminUsername

ORDER BY adminUsername;



// Shows number of enemies, the minimum health, maximum health, and average kill time

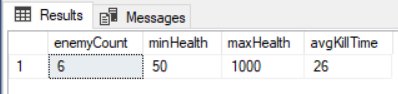
SELECT COUNT(\*) AS enemyCount,

MIN(health) AS minHealth,

MAX(health) AS maxHealth,

AVG(timeToKill) AS avgKillTime

FROM Enemy;



// Shows the number of kill logs for each player

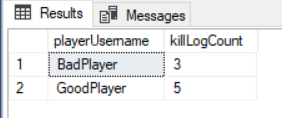
SELECT playerUsername,

COUNT(\*) AS killLogCount

FROM KillLog

GROUP BY playerUsername

ORDER BY playerUsername;



// Shows the number of each different types of legendary weapons and the average damage of them

SELECT type,

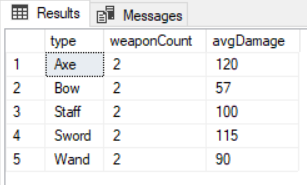
COUNT(name) AS weaponCount,

AVG(damage) AS avgDamage

FROM LegendaryWeapon

GROUP BY type

ORDER BY type;



// Shows the number of players, the minimum level, maximum level, minimum exp, max exp, minimum gold, and maximum gold present

SELECT COUNT(\*) AS playerCount,

MIN(level) AS minLevel,

Max(level) AS maxLevel,

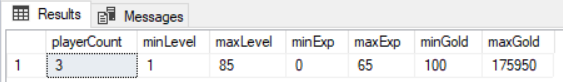
MIN(exp) AS minExp,

Max(exp) AS maxExp,

MIN(gold) AS minGold,

MAX(gold) AS maxGold

FROM Player;



// Shows the number of weapons each player has and the average damage of them

SELECT playerUsername,

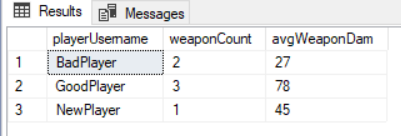
COUNT(\*) AS weaponCount,

AVG(damage) AS avgWeaponDam

FROM Weapon

GROUP BY playerUsername

ORDER BY playerUsername;



// Shows the number of weapons, the minimum damage, the maximum damage, and the average bonus of them

SELECT COUNT(\*) AS weaponCount,

MIN(minDmg) AS minDamage,

MAX(maxDmg) AS maxDamage,

AVG(rarityBonus) AS avgBonus

FROM WeaponType;

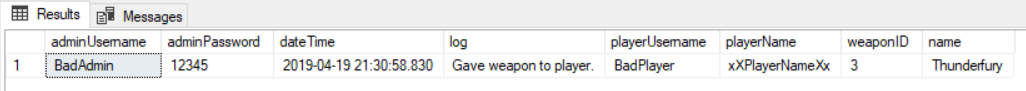


// Returns admin logs with weapons given, including the admins password and weapon's name

SELECT Admin.username adminUsername, Admin.[password] adminPassword, dateTime, log, Player.username playerUsername, playerName, AdminLog.weaponID, Weapon.name

FROM Admin, AdminLog, Player, Weapon

WHERE Admin.username = AdminLog.adminUsername AND Player.username = AdminLog.playerUsername AND Weapon.weaponID = AdminLog.WeaponID;

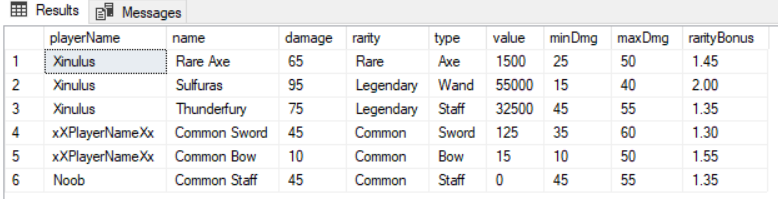


// Shows more information about all weapons in the game

SELECT playerName, Weapon.name, damage, rarity, type, value, minDmg, maxDmg, rarityBonus

FROM Player, Weapon, WeaponType

WHERE Player.username = Weapon.playerUsername AND Weapon.type = WeaponType.name;



// Shows the amount of enemies each player has killed, including their location

SELECT playerName, nameOfEnemy, location, COUNT(nameOfEnemy) kills

FROM Player, KillLog, Enemy

WHERE Player.username = KillLog. playerUsername AND

KillLog.nameOfEnemy = Enemy.name

GROUP BY playerName, nameOfEnemy, location

ORDER BY COUNT(nameOfEnemy) DESC;



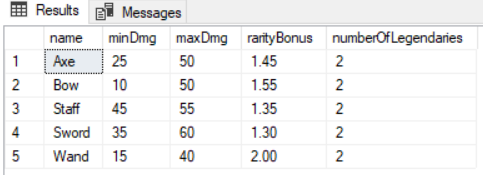
// Shows information on each weapon type, including the number of legendaries

SELECT WeaponType.name, minDmg, maxDmg, rarityBonus, COUNT (LegendaryWeapon.name) numberOfLegendaries

FROM WeaponType, LegendaryWeapon

WHERE WeaponType.name = LegendaryWeapon.type

GROUP BY WeaponType.name, minDmg, maxDmg, rarityBonus;



**Conclusion:**

This project has been extremely helpful in learning the ways to use a database, and we have learned a lot about how game companies might use a database to store data from their game.

**Reference:**

We used the Microsoft SQL server tutorial to set up the Database.

https://www.quackit.com/sql\_server/tutorial/