

# **Task Manager**

**Database Management**

**308N-201**

**MaristTask**



Marist College  
School of Computer Science and Mathematics

Submitted to:  
Dr. Reza Sadeghi

9/18/2023

# Project Report of Red Fox Task

## Team Name

RedFoxTask

## Team Members:

1. Kevin Reiff [Kevin.Reiff1@marist.edu](mailto:Kevin.Reiff1@marist.edu) Team Member

## Description of Team Members

1. Kevin Reiff - I am a computer science major from Vernon NJ. I have been working as an intern for the IT Security Department at Selective Insurance since the beginning of the summer and I am still working for them throughout the semester. I wanted to work with these teammates because I knew two of them previously and we all share similar interests. We chose the team head via discussing out of class responsibilities and who was best equipped to take on the responsibility.

## Table of Contents

Project Objective .....	5
Related Work Review .....	6
Project Merits .....	7
Github Repository Address .....	9
Phase 2: ER Model and EER Model .....	10
Phase 3: Presentation and SQL Files .....	12
Phase 4: Database Development .....	16
Phase 5: Loading Data and Performance Enhancement.....	23
Phase 6: Application Development.....	35
References .....	61

## Project Objective

### **Project Title:** MaristTask

The MaristTask is a task management system that will display a calendar for the selected day, week, month or year. It organizes the specific tasks of different users each day, and users have the ability to update these tasks at any time. The calendar should update in day, week, month and year form when a task is added. The task management system will store the data of different users in distinct SQL tables.

This task management system includes multi-level user authentication, a dynamic calendar display, personalized task management, an advanced search function, a user friendly interface, data reporting, alerts and warnings, and an easy exit function.

The multi-level user authentication gives access to the system to both users and admins. Users are able to manage their own task pages, while admins have access to security pages as well as the pages of regular users. The calendar display gives users options to choose their preference when it comes to listing their tasks, whether that is in an annual, monthly, weekly, or daily view. The personalization of the task manager continues, as users can edit each part of the tasks they create, all the way down to the color it displays as. The task management system includes an advanced search function, where users can search within their tasks using a variety of parameters. All of these features are presented in a user friendly interface that is easy to maneuver and is aesthetically pleasing to the eye. For easier functionality, an alert system is integrated that will alert users if there are overlapping dates or tasks. Users have the ability to easily exit the task management system with a simple logout option.

## **Related Work Review:**

Trello - A good task manager that is used for business. It uses all-around good software, where you can keep track of tasks done, upcoming tasks, tasks to do, and tasks you are in the middle of completing<sup>[3]</sup>. However, Trello does not have a way to directly message people if something needs to be done. This would help a lot in our application.

Jira - Another good task manager which uses a timeline to create tasks that you have to do. It shows the start and end date on a type of calendar that goes throughout months<sup>[2]</sup>. However, it is a bit too confusing. You can make teams, but the tasks are confusing and awkward to look at. I think they should make it simpler and more user friendly to make people want it.

Todoist - Another great task managing software which is used by huge companies such as Disney, Microsoft, Amazon, and Netflix. This offers an easy-to-manage and visually pleasing to do list of everything you need to complete. It shows personal tasks for yourself, team tasks if you have a team company you are working with, and different workspaces to list your groceries, fitness routines, appointments, and personal goals. You can also join teams to create joint tasks<sup>[1]</sup>. Just like with Trello, what would make this better is a messaging software within it to contact your team at any time.

## Project Merits

### 1. Core Capabilities:

- a. **Multi-Level User Authentication:** Both admin and standard user roles are available in our task managing system, each with unique rights. This improves security and manageability by ensuring that the appropriate users have access to the appropriate functionality. Unlike Trello and Todoist, our task managing system includes a dedicated admin role for enhanced security and user management.
- b. **Dynamic Calendar Display:** Users may choose between weekly, monthly, and annual views, giving them the flexibility to plan and arrange work as needed. While Jira provides a timeline, our task managing system offers a more straightforward calendar display that is easier to interpret.
- c. **Personalized Task Management:** Tasks particular to each account can be added, modified, or removed by the user. Each assignment has necessary details including the title, start and end times, as well as a description. Unlike Jira, our task managing system aims for simplicity and user-friendliness in task management.
- d. **Advanced Search Functionality:** Users have the option of searching for tasks using several parameters, including duration, title, and time. This makes finding certain activities simple and quick. This search functionality exceeds the capabilities of Trello, which lacks an advanced search feature based on multiple parameters.
- e. **User-Friendly Interface:** Our task managing system will have an initial welcome page as well as a structured menu of all functions on each page, making it easy to use and intuitive. Unlike Jira, which can be confusing to new users, our task manager system focuses on user experience..
- f. **Data Reporting:** To assist users in their responsibilities in a structured fashion, the system can create tabular reports that include well-organized calendars. Compared to Todoist, this function is more comprehensive.
- g. **Alerts and Warnings:** Our system will notify users anytime they try to input contact information that already exists in the database in order to prevent repetition. This feature is not common in the other task management systems mentioned.
- h. **Graceful Exit:** The user experience is improved via a special exit function that thanks the user for using the program.

### 2. Why Choose Our Product?

- a. **Highly Customizable:** Our task managing system may be customized to meet individual needs thanks to the option to switch between calendar views and alter task information.

- b. **Secure User Management:** Our task managing system prioritizes data security and will have separate SQL databases for various user categories as well as strict password requirements.
- c. **Efficiency:** Users can better manage their time and responsibilities with the help of the sophisticated search capabilities and tabular reports that our system will have.
- d. **User-Centric Design:** Even individuals with rudimentary technological knowledge can utilize the system with ease because of its user-friendly interface.
- e. **Data integrity:** The built-in alter system that we will implement into our task managing system will prevent duplicate entries, maintaining the reliability of the stored data.
- f. **Comprehensive Solution:** Our task managing system will provide a one-stop shop for all task management requirements, covering task creation, reporting and even user administration (for admins).

## **Github Repository Address**

GitHub Repository:

[https://github.com/SamGuggino/CMPT308\\_TaskManager\\_RedFoxTask](https://github.com/SamGuggino/CMPT308_TaskManager_RedFoxTask)



## **Phase 2: ER Model and EER Model:**

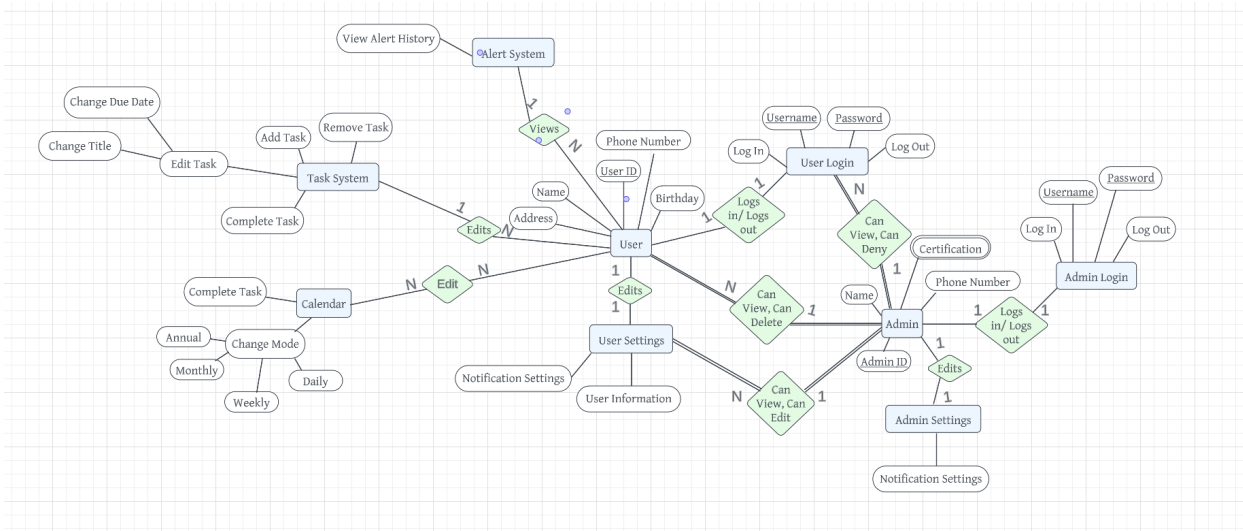
### **ER Model:**

Our Entity Relationship Model meticulously delineates the functionalities accessible to users and administrators within RedFoxTask, prioritizing clarity and ease of understanding. Designed to be intuitive, the diagram elucidates the data requisites and storage loci, employing a many-to-many cardinality to reflect the simultaneous multitasking capabilities endowed to both users and admins.

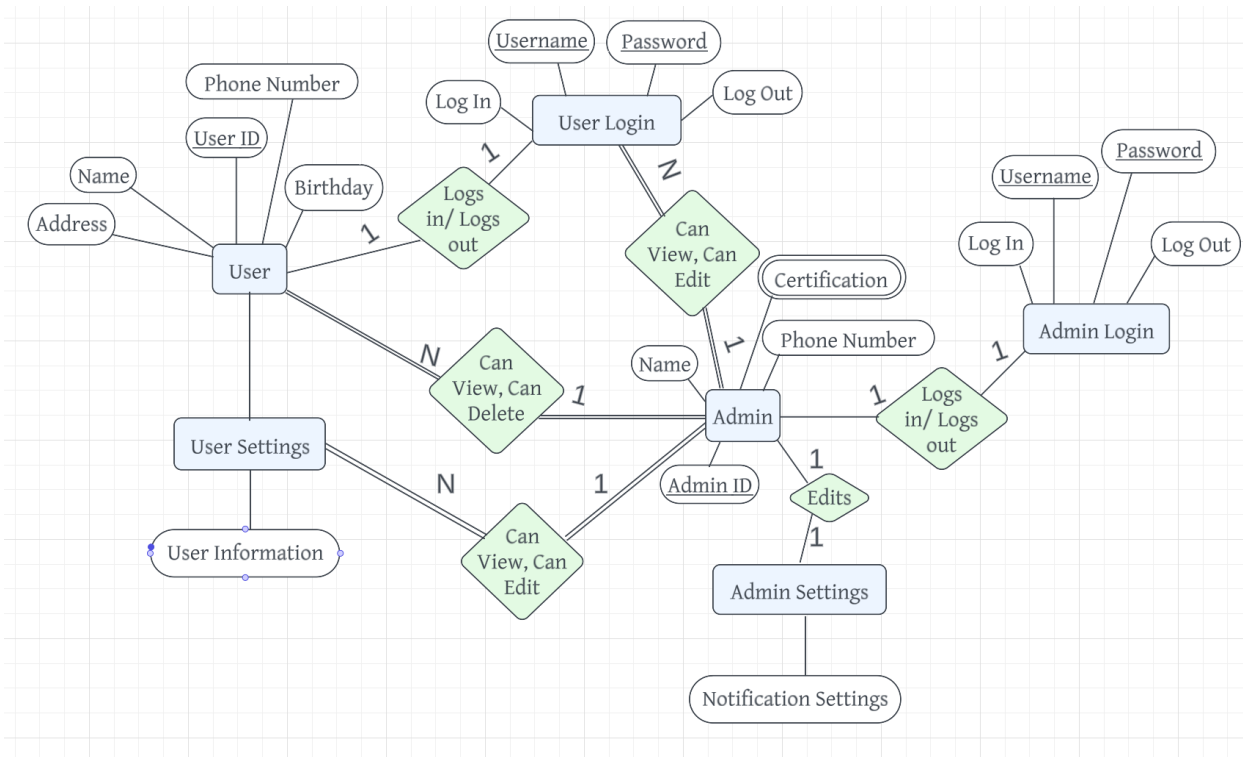
Users are furnished with a comprehensive suite of basic information fields, including user ID, name, birthday, address, and phone number, all of which establish a robust user profile. Beyond these fundamentals, users are empowered to tailor their settings, manage notification preferences, and modify personal information. The system's alert module is a versatile tool, enabling users to sanction, purge, or retrieve historical alerts. The task management framework is equally dynamic, offering functionalities for task completion, amendment, addition, and stylistic customization. Moreover, it boasts an integrated calendar feature, offering daily, weekly, monthly, and yearly views, aiding users in tracking and accomplishing their objectives.

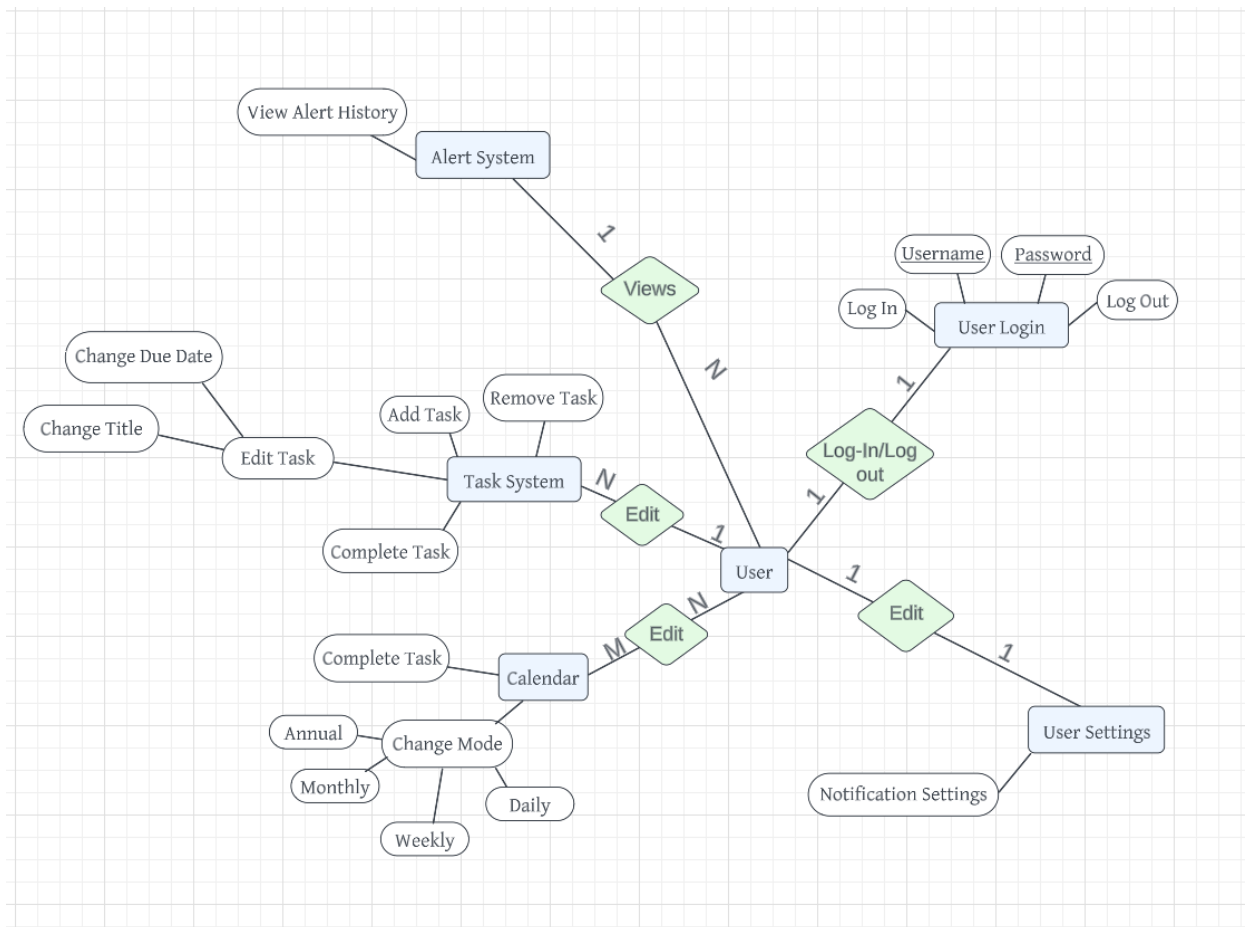
Administrators possess a distinct set of primary identifiers: name, admin ID, phone number, and professional certifications. Mirroring user capabilities, admins can modify their notification and information settings within a dedicated admin portal, despite sharing a common login interface with users. The delineation of roles becomes evident through the admin's overarching control — they have the exclusive authority to scrutinize, amend, expunge, or veto entries pertaining to users, their settings, and their login activities. This absolute participation underscores the admin's pivotal role in system oversight, contrasting with the user's limited participation, which excludes any influence over administrative functions.

**Figure 1: ER Diagram**



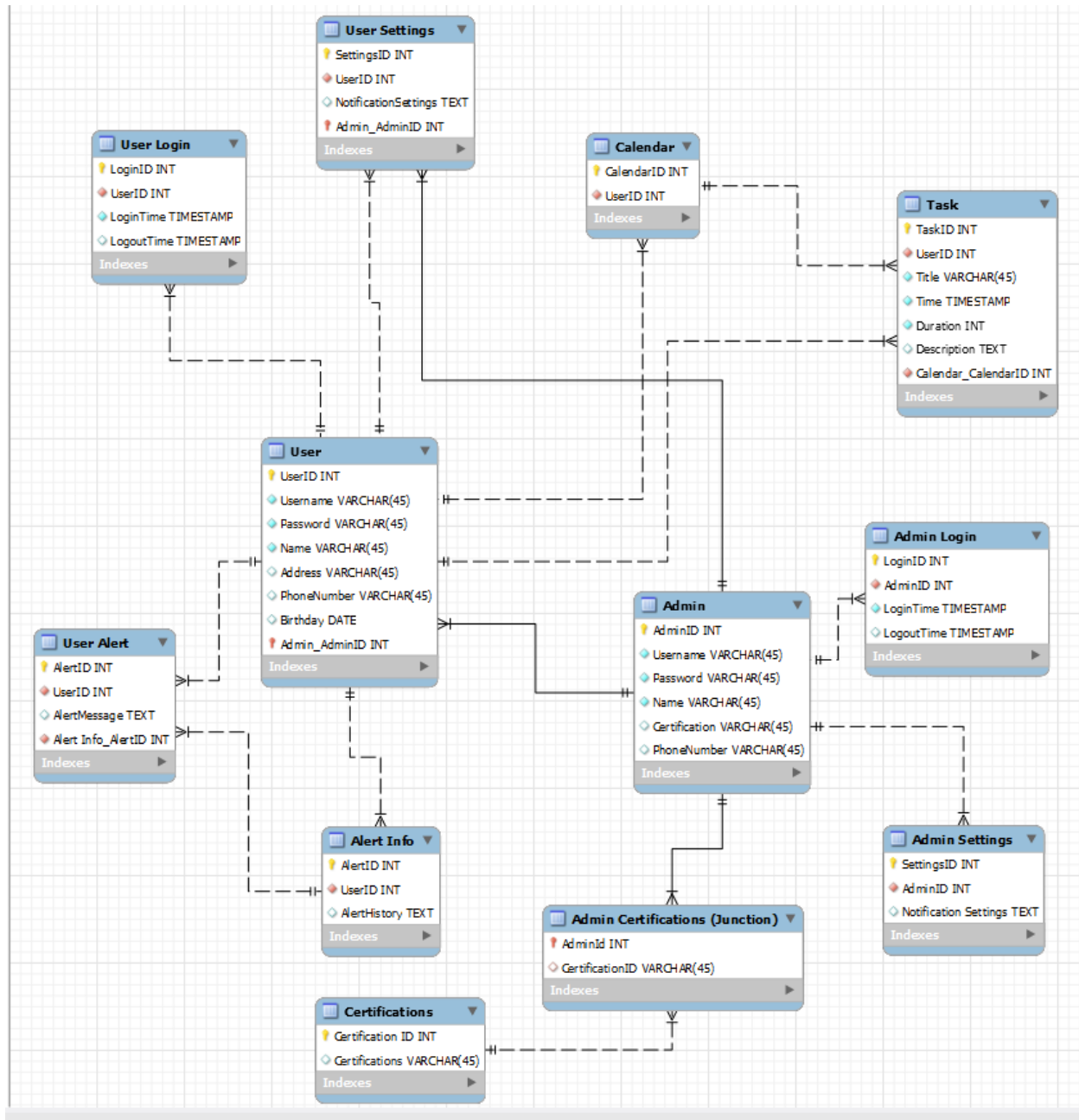
**Figure 2: External Diagram #1**



**Figure 3: External Diagram #2****EER Diagram:**

My Enhanced Entity Relationship (EER) Diagram provides a detailed view of the database structure for my Task Management System involving users and administrators. Starting with the User table, it has a one-to-many relationship with multiple tables: User Settings: Each user has corresponding settings, indicating that a user can have multiple settings but each setting is associated with only one user. Calendar: Users can have multiple calendar entries, each uniquely identified within the Calendar table. User Login: This tracks the login and logout timestamps for users. A user can have multiple login records. User Alert: Indicates alerts specific to users, where a user can have multiple alerts. Task: Users are associated with tasks, where each task is related to one user, but a user can have multiple tasks. The Admin entity has a similar structure: Admin Login: Tracks login and logout activities of an admin. Admin Settings: Stores settings for each admin, with a one-to-many relationship. Admin Certifications (Function): It appears to represent a function rather than a table, potentially indicating certifications that an

admin can issue or possess. The exact nature of the function is not clear from the diagram alone. Certifications are a multi-valued attribute and thus needed a junction table in the form of admin certifications, which is then connected to admin. Foreign keys that establish the relationships between tables, such as user IDs or alert information. The foreign keys create a relational structure that allows the database to maintain referential integrity.



## Phase 3: Presentation and SQL Files:

### Presentation:

#### Slide 2: Default Values

- A DEFAULT clause indicates a default value for a column.
- It can be a literal constant or an expression.
- Expression values are put in parentheses.
- For TIMESTAMP and DATETIME columns, CURRENT\_TIMESTAMP can be used without parentheses.
- SERIAL DEFAULT VALUE is used for an integer column, acting as an alias for NOT NULL AUTO\_INCREMENT UNIQUE.
- BLOB, TEXT, GEOMETRY, and JSON data types can only be assigned default values if they are written as expressions.

#### Slide 3: Rules for Expression Default Values

- Allowed constructs: Literals, built-in functions, and operators.
- Disallowed constructs: Subqueries, parameters, variables, stored functions, and loadable functions.
- An expression default cannot use a column that has an AUTO\_INCREMENT attribute.
- An expression default value for one column can refer to other table columns, but references must be to columns that appear earlier in the table definition.
- This rule also applies to ALTER TABLE to reorder columns.

#### Slide 4: Altering Tables

- For CREATE TABLE ... LIKE and CREATE TABLE ... SELECT, the destination table uses default values from the original table.
- If an expression default value has a nondeterministic function, it cannot be used for statement-based replication.
- This includes INSERT and UPDATE.
- If binary logging is disabled, the statement will execute.
- If binary logging is enabled and binlog\_format is set to STATEMENT, the statement will execute but a warning will be sent to the error log.
- If binlog\_format is set to MIXED or ROW, the statement will execute.

#### Slide 5: Inserting Rows

- To insert a new row, the default value of a column with an expression value can be inserted by removing the column name or setting it to DEFAULT.

- The use of DEFAULT(col\_name) to specify the default value for a named column can only be used for columns that have a literal default value.
- Not all storage engines allow expression values.
- ER\_UNSUPPORTED\_ACTION\_ON\_DEFAULT\_VAL\_GENERATED will occur.

#### Slide 6: Implicit Default Handling

- If there is no explicit DEFAULT value, MySQL will:
- Define the column with DEFAULT NULL clause if the column can take NULL as a value.
- Define the column with no explicit DEFAULT clause if null cannot be used.
- For data entry into a NOT NULL column:
- In strict SQL mode, an error will occur.
- In non-strict mode, MySQL sets the column to the implicit default value for the column data type.

#### Slide 7: Implicit Defaults

- SHOW CREATE TABLE statement reveals columns with an explicit DEFAULT clause.
- Numeric types default to 0.
- Integer or floating-point types with AUTO\_INCREMENT attribute default to the next value.
- Date and time types (except TIMESTAMP) default to the correct "zero" value.
- TIMESTAMP's default depends on explicit\_defaults\_for\_timestamp system variable.
- If enabled, TIMESTAMP defaults to the current date and time.
- Otherwise, it defaults to the current date and time if it's the first TIMESTAMP column.
- Default value for non-ENUM string types is the empty string.
- ENUM defaults to the first enumeration value.

#### Slide 8: Demo 1

```
1 • CREATE DATABASE IF NOT EXISTS ProjectPhaseDefaultValuesDemo;
2 • USE ProjectPhaseDefaultValuesDemo;
3
4 • DROP TABLE IF EXISTS DefaultValueExample;
5
6 -- Create a table with default value specifications
7 • CREATE TABLE DefaultValueExample (
8     myInt INT DEFAULT 0,
9     myVarchar VARCHAR(45) DEFAULT 'No Data',
10    myFloat FLOAT DEFAULT 1.0,
11    myDate DATE DEFAULT (CURRENT_DATE + INTERVAL 1 DAY),
12    myTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
13    myEnum ENUM('Value1', 'Value2', 'Value3') DEFAULT 'Value1',
14    mySet SET('A', 'B', 'C') DEFAULT 'A,B'
15 );
```

## Slide 9: Demo 2

```
1 • USE ProjectPhaseDefaultValuesDemo;
2
3 -- Inserts a row with all of the default values
4 • INSERT INTO DefaultValueExample () VALUES();
5
6 -- Inserts a new row with only 3 values specified, the rest will be default
7 • INSERT INTO DefaultValueExample (myInt, myVarchar, myDate) VALUES (99, 'Data', '2023-05-05');
8
9 • SELECT * FROM DefaultValueExample;
10
11 • SELECT CONCAT(
12     myInt, ', ',
13     myVarchar, ', ',
14     myFloat, ', ',
15     myDate, ', ',
16     myTimestamp, ', ',
17     myEnum, ', (' ,
18     mySet, ') '
19 ) AS allColumnConcatenation
20 FROM DefaultValueExample;
```

## Slide 10: Choosing the Right Type for a Column

- Optimal storage: Prefer the most precise data type in most cases.
- Example: Use MEDIUMINT UNSIGNED for values 1-99999 in an integer column.
- DECIMAL column calculations: Have a precision of 65 decimal digits for basic operations (+, -, \*, and /).
- Speed over accuracy: Consider using the DOUBLE type when precision isn't critical.
- High precision option: Convert to a fixed-point type stored in a BIGINT for precision.
- Perform calculations with 64-bit integers and convert results back to floating-point values as needed.

## Slide 11: Demo 3

```

1 • create database if not exists TheRightType;
2 • use TheRightType;
3
4 • DROP TABLE IF EXISTS TheRightType;
5
6 • CREATE TABLE Example (
7     myInt int,
8     myTinyInt tinyint,
9     mySmallInt smallint,
10    myMediumInt mediumint,
11    myDecimal decimal,
12    myFloat float
13 );
14 • insert into Example (myInt, myTinyInt, mySmallInt, myMediumInt, myDecimal, myFloat)
15 values (3359000, 0, 50, 43000, 33.2, 5.983),
16        (190, -2, 120, 45, -3.9, 10.254),
17        (495, 1, 7, 568, 276.4, -96.892);

```

## Slide 12: Demo 4

```

1    -- Use the previously created database and table
2    USE TheRightType;
3
4    -- Query to demonstrate that MEDIUMINT is sufficient for storing values between 1 and 99999
5 •  SELECT * FROM Example WHERE myMediumInt BETWEEN 1 AND 99999;
6
7    -- Query to show calculations with FLOAT type columns
8 •  SELECT myFloat / 2 AS 'HalfFloat' FROM Example;
9
10   -- Query to demonstrate high precision with DECIMAL
11 •  SELECT myDecimal + 0.0000000001 AS 'HighPrecisionDecimal' FROM Example;
12
13   -- Query to demonstrate speed with FLOAT but loss of precision
14 •  SELECT myFloat + 0.0000000001 AS 'LowPrecisionFloat' FROM Example;
15
16   -- Query to demonstrate range of SMALLINT
17 •  SELECT * FROM Example WHERE mySmallInt BETWEEN -32768 AND 32767;
18
19   -- Query to demonstrate range of TINYINT
20 •  SELECT * FROM Example WHERE myTinyInt BETWEEN -128 AND 127;

```



## Phase 4: Database Development

The 'CREATE TABLE IF NOT EXISTS' statement checks if the 'User' table already exists within the database. If it does not, the following table will be created. This code defines the structure for the 'User' table with various attributes related to user information. 'UserID' is an integer column that represents a unique identifier for each user. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. 'Username' is a variable character column that has a maximum length of 45 characters. It stores the display name of the user. 'Password' is a variable character column that has a maximum length of 45 characters. It stores the user's password. 'Name' is a variable character column that has a maximum length of 45 characters. It stores the full name of the user. 'Address' is a variable character column that has a maximum length of 45 characters. It stores the user's address. 'PhoneNumber' is a variable character column that has a maximum length of 45 characters. It stores the user's phone number. 'Birthday' is a date type column. It stores the user's birthday. The columns 'Username', 'Password', and 'Name' have the 'NOT NULL' constraint, meaning these columns must always have a value in them.

```
-- Create the User table
CREATE TABLE IF NOT EXISTS User (
  UserID INT PRIMARY KEY,
  Username VARCHAR(45) NOT NULL,
  Password VARCHAR(45) NOT NULL,
  Name VARCHAR(45) NOT NULL,
  Address VARCHAR(45),
  PhoneNumber VARCHAR(45),
  Birthday DATE
);
```

Figure 5: User Table Code

The 'CREATE TABLE IF NOT EXISTS' statement checks if the 'UserSettings' table already exists within the database. If it does not, the following table will be created. This code defines the structure for the 'UserSettings' table that holds settings related to individual users. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. 'SettingsID' is an integer column that represents a unique identifier for each setting. 'UserID' is an integer column that is used to link settings to a user. This column is a foreign key, which means that each record in 'UserSettings' has a valid corresponding 'UserID' in the 'User' table. This establishes a many to one relationship between the 'UserSettings' table and the 'User' table. 'Mode' is a variable character column that has a maximum length of 45 characters. It stores the mode of the user settings (e.g., dark mode or light mode). 'NotificationSettings' is a text type column, which stores notification settings set by the user.

```
-- Create the User Settings table
CREATE TABLE IF NOT EXISTS UserSettings (
    SettingsID INT PRIMARY KEY,
    UserID INT,
    Mode VARCHAR(45),
    NotificationSettings TEXT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 6: User Settings Table Code

The 'CREATE TABLE IF NOT EXISTS' statement checks if the 'Calendar' table already exists within the database. If it does not, the following table will be created. This code defines the structure for the 'Calendar' table that holds calendar related data for individual users. 'CalendarID' is an integer column that represents a unique identifier for each calendar entry. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. 'UserID' is an integer column that is used to link a calendar to a user. This column is a foreign key, which means that each record in 'Calendar' has a valid corresponding 'UserID' in the 'User' table. This establishes a many to one relationship between the 'Calendar' table and the 'User' table. 'Mode' is an integer column that represents the numerical code that indicates a specific mode of the calendar.

```
-- Create the Calendar table
CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT PRIMARY KEY,
    UserID INT,
    Mode INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 7: Calendar Table Code

The 'CREATE TABLE IF NOT EXISTS Task' table is created with several columns to store task-related information such as the title, description, and duration as well as having the calendar linked to the table. With the CalendarID entity linked to this table, the user is allowed to give their tasks a time table with a due date and time. The UserID is also linked to this table as it allows each user to have their own, unique task table. The Title entity allows you to name a task that is up to 45 characters in length. This value can never be blank, therefore, each task created must have a title. The rest of the entities have data types that can be filled in by the user when they create the task.

```
-- Create the Task table
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT PRIMARY KEY,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

Figure 8: Task Table Code

The 'CREATE TABLE IF NOT EXISTS UserAlert' allows the system to create user alerts for their FoxTask to-do's. The AlertID is the primary key as it is the distinct value for this table and it is needed for the rest of the values. The UserID is an integer and is different for every user. The AlertMessage is a TEXT data type because when the user gets the alert, it has to be in text form so they can properly read it. Each alert will be different depending on the task which is why each alert has an AlertID, this is an integer and will change each alert the user gets.

```
-- Create the User Alert table
CREATE TABLE IF NOT EXISTS UserAlert (
    AlertID INT PRIMARY KEY,
    UserID INT,
    AlertMessage TEXT,
    Alert_Info_AlertID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 9: User Alert Table Code

The 'CREATE TABLE IF NOT EXISTS UserLogin' is a table with columns for a unique login identifier(LoginID), a user identifier(UserID) to link a user in another table with timestamps that allow you to record login and logout times. LoginID is the primary key as it uniquely identifies each row in the table. The UserID is an integer that will represent the user's identification throughout the whole process, this is linked to the other UserID's in these tables. The LoginTime and LogoutTimes represent time stamps with dates and times when the user logs in or out of the app.

```
-- Create the User Login table
CREATE TABLE IF NOT EXISTS UserLogin (
    LoginID INT PRIMARY KEY,
    UserID INT,
    LoginTime TIMESTAMP,
    LogoutTime TIMESTAMP,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 10: User Login Table Code

The 'CREATE TABLE IF NOT EXISTS AlertInfo' table has three columns, AlertID, UserID, and AlertHistory. These values just establish relationships with the other entities in the remaining diagrams. The foreign key ensures that the values in the "UserID" column of the "AlertInfo" table must exist in the "UserID" column of the User table which creates a link between the two tables. Once again, each user has their own unique ID and alert ID to allow the data to be separated from others.

```
-- Create the Alert Info table
CREATE TABLE IF NOT EXISTS AlertInfo (
    AlertID INT PRIMARY KEY,
    UserID INT,
    AlertHistory TEXT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 11: Alert Info Table Code

The 'CREATE TABLE IF NOT EXISTS Admin' allows the admins to log in and create a username, password, name, certification, and phone number for their profile. We say "IF NOT EXISTS" in this situation as it reduces redundancy and errors if the table was already created somewhere else. Each of these entities must be in the range of 0-45 characters or else it is not valid. It is optional for the admin to add their certification and phone number but the rest of the values must be filled as it is declared as NOT NULL. The foreign key establishes a relationship between UserID and AlertInfo. This ensures that the values of the UserID column of the AlertInfo table must exist in the UserID column of the user table, which also creates a link between these two tables.

```
-- Create the Admin table
CREATE TABLE IF NOT EXISTS Admin (
  AdminID INT PRIMARY KEY,
  Username VARCHAR(45) NOT NULL,
  Password VARCHAR(45) NOT NULL,
  Name VARCHAR(45) NOT NULL,
  Certification VARCHAR(45),
  PhoneNumber VARCHAR(45)
);
```

Figure 12: Admin Table Code

The 'CREATE TABLE IF NOT EXISTS AdminLogin' allows admin to login through their adminID and gets a login ID with an integer and primary key. It also logs the specific time that the person logged into their account and then when they log out it keeps a stamp of when they logged out. Using the AdminID Key.

```
-- Create the Admin Login table
CREATE TABLE IF NOT EXISTS AdminLogin (
  LoginID INT PRIMARY KEY,
  AdminID INT,
  LoginTime TIMESTAMP,
  LogoutTime TIMESTAMP,
  FOREIGN KEY (AdminID) REFERENCES Admin(AdminID)
);
```

Figure 13: Admin Login Table Code

"CREATE TABLE IF NOT EXISTS AdminSettings" table is designed to store various settings for administrators. The AdminID column makes a relationship with the "Admin" table, making sure that each setting is associated with an administrator. The table's primary key is the SettingsID column, which provides a unique identifier for each row in the table.

```
-- Create the Admin Settings table
CREATE TABLE IF NOT EXISTS AdminSettings (
  SettingsID INT PRIMARY KEY,
  AdminID INT,
  Mode VARCHAR(45),
  NotificationSettings TEXT,
  FOREIGN KEY (AdminID) REFERENCES Admin(AdminID)
);
```

Figure 14: Admin Settings Table Code



## Phase 5.1: Loading data and performance enhancements:

### Handling Foreign Key Constraints:

Part B: In my SQL code, I have included an example of an insertion error due to foreign key constraints. (Error below)

```
Cannot add or update a child row: a foreign key constraint fails
(`redfoxtask`.`usersettings`, CONSTRAINT `usersettings_ibfk_1` FOREIGN KEY
(`UserID`) REFERENCES `user` (`UserID`))      0.000 sec
```

This error occurs when trying to insert data into a child table (UserSettings) with a foreign key (UserID) that does not exist in the parent table (User). (The line that caused the error is displayed below)

```
-- Line that would cause the insertion error due to foreign key constraints
INSERT INTO UserSettings (SettingsID, UserID, Mode, NotificationSettings) VALUES
(11, 999, 'Dark', 'Email');--Primary Key Constraint Test
```

Part C: The method that we used to handle the issue of foreign key inconsistency was to set foreign key checks to 0 before the insertions and then set it back to 1 after the insertions. We used this method because it is useful for bulk data insertion where the order of data insertion might temporarily violate foreign key constraints, and our data insertion is done in bulk. This can be seen at the start of our data insertion code here:

```
1 • use RedFoxTask;
2
3 • SET FOREIGN_KEY_CHECKS=0;
```

And the end of our data insertion code here:

```
137 (8, 8, 'Light', 'Email, SMS'),
138 (9, 9, 'Dark', 'Email'),
139 (10, 10, 'Light', 'SMS');
140
141 • SET FOREIGN_KEY_CHECKS=1;
---
```

## Importing Data:

Our importing process involved inserting data into tables such as User, UserSettings, Calendar, Task, UserAlert, and several others. Each table received a variety of data, showcasing our database's capability to handle different data types, including VARCHAR, INT, DATE, TIMESTAMP, and TEXT.

**Figure 1:** Importing the data for the User and UserSettings tables

```
-- Optimized bulk insertions take a total time of 0.000 seconds (I assume this means a number
-- smaller than 0.000 but too small to be displayed with 3 significant figures)
• INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
(1, 'KevinR', 'KevinR1', 'Kevin Reiff', '123 Main St', '123-456-7890', '2002-10-23'),
(2, 'janedoe', 'pass456', 'Jane Doe', '124 Main St', NULL, '1992-02-02'),
(3, 'mikeb', 'mikepass', 'Mike Brown', '125 Main St', '123-456-7891', '1988-03-03'),
(4, 'emmagreen', 'emma1234', 'Emma Green', NULL, '123-456-7892', '1995-04-04'),
(5, 'samw', 'samspass', 'Sam Wilson', '126 Main St', '123-456-7893', '1991-05-05'),
(6, 'lucyp', 'lucyword', 'Lucy Parker', '127 Main St', '123-456-7894', '1989-06-06'),
(7, 'davidt', 'davidpwd', 'David Taylor', NULL, '123-456-7895', '1993-07-07'),
(8, 'sarahc', 'sarahpass', 'Sarah Connor', '128 Main St', '123-456-7896', '1994-08-08'),
(9, 'rickg', 'rick123', 'Rick Grimes', '129 Main St', NULL, '1996-09-09'),
(10, 'annak', 'anna456', 'Anna Klein', '130 Main St', '123-456-7897', '1990-10-10');

• INSERT INTO UserSettings (SettingsID, UserID, NotificationSettings) VALUES
(1, 1, 'SMS'),
(2, 2, NULL),
(3, 3, 'SMS'),
(4, 4, NULL),
(5, 5, 'SMS'),
(6, 6, NULL),
(7, 7, NULL),
(8, 8, 'SMS'),
(9, 9, 'SMS'),
(10, 10, 'SMS');
```

The User table stores information about individual users, including their username, password, personal details, and contact information. As for the imported data, 10 distinct user records were inserted. The data demonstrates a range of different usernames, addresses, and birthdays.

Notably, some users do not have addresses or phone numbers, showcasing optional fields. The UserSettings table holds settings preferences for each user, like display mode and notification settings. The data shows variations and notification settings, including null values.



**Figure 2:** Importing the data for the Calendar, Task, and UserAlert tables

```

45 • INSERT INTO Calendar (CalendarID, UserID, Mode) VALUES
46     (1, 1, 1),
47     (2, 2, 0),
48     (3, 3, 1),
49     (4, 4, 0),
50     (5, 5, 1),
51     (6, 6, 0),
52     (7, 7, 1),
53     (8, 8, 0),
54     (9, 9, 1),
55     (10, 10, 0);
56
57 • INSERT INTO Task (TaskID, UserID, Title, Time, Duration, Description, Calendar_CalendarID) VALUES
58     (1, 1, 'Task 1', '2023-11-10 09:00:00', 60, 'Description for Task 1', 1),
59     (2, 1, 'Task 2', '2023-11-10 10:00:00', 30, 'Description for Task 2', 1),
60     (3, 2, 'Task 3', '2023-11-11 09:00:00', 200, 'Description for Task 3', 2),
61     (4, 2, 'Task 4', '2023-11-11 10:15:00', 1000, 'Description for Task 4', 2),
62     (5, 3, 'Task 5', '2023-11-12 09:00:00', 300, 'Description for Task 5', 3),
63     (6, 3, 'Task 6', '2023-11-12 10:00:00', 45, 'Description for Task 6', 3),
64     (7, 4, 'Task 7', '2023-11-13 09:00:00', 120, 'Description for Task 7', 4),
65     (8, 4, 'Task 8', '2023-11-13 10:00:00', 180, 'Description for Task 8', 4),
66     (9, 5, 'Task 9', '2023-11-14 09:00:00', 90, 'Description for Task 9', 5),
67     (10, 5, 'Task 10', '2023-11-14 10:15:00', 365, 'Description for Task 10', 10);
68
69 • INSERT INTO UserAlert (AlertID, UserID, AlertMessage, Alert_Info_AlertID) VALUES
70     (1, 1, 'Alert Message 1', 1),
71     (2, 2, 'Alert Message 2', 2),
72     (3, 3, 'Alert Message 3', 3),
73     (4, 4, 'Alert Message 4', 4),
74     (5, 5, 'Alert Message 5', 5),
75     (6, 6, 'Alert Message 6', 6),
76     (7, 7, 'Alert Message 7', 7),
77     (8, 8, 'Alert Message 8', 8),
78     (9, 9, 'Alert Message 9', 9),
79     (10, 10, 'Alert Message 10', 10);

```

The Calendar table links users to their calendar entries, represented by a mode. The mode field demonstrates binary states, 0 and 1, for different calendar types. The Task table contains tasks associated with users, including title, duration, and description. The imported data showcases varying lengths and types of tasks, with different durations and descriptions. The UserAlert table contains alert messages for the users. The imported data shows that each record is a unique combination of an alert ID, a corresponding user ID, an alert message, and a

reference to an alert information ID. These records demonstrate the ability of the table to store personalized alerts for each user.

**Figure 3:** Importing the data for the Userlogin, AlertInfo, and Admin tables

```

81 • INSERT INTO UserLogin (LoginID, UserID, LoginTime, LogoutTime) VALUES
82     (1, 1, '2023-11-10 08:00:00', '2023-11-10 17:00:00'),
83     (2, 2, '2023-11-10 09:00:00', '2023-11-10 18:00:00'),
84     (3, 3, '2023-11-11 08:30:00', '2023-11-11 17:30:00'),
85     (4, 4, '2023-11-11 09:30:00', '2023-11-11 18:30:00'),
86     (5, 5, '2023-11-12 08:00:00', '2023-11-12 17:00:00'),
87     (6, 6, '2023-11-12 09:00:00', '2023-11-12 18:00:00'),
88     (7, 7, '2023-11-13 08:30:00', '2023-11-13 17:30:00'),
89     (8, 8, '2023-11-13 09:30:00', '2023-11-13 18:30:00'),
90     (9, 9, '2023-11-14 08:00:00', '2023-11-14 17:00:00'),
91     (10, 10, '2023-11-14 09:00:00', '2023-11-14 18:00:00');
92
93 • INSERT INTO AlertInfo (AlertID, UserID, AlertHistory) VALUES
94     (1, 1, 'History 1'),
95     (2, 2, 'History 2'),
96     (3, 3, 'History 3'),
97     (4, 4, 'History 4'),
98     (5, 5, 'History 5'),
99     (6, 6, 'History 6'),
100    (7, 7, 'History 7'),
101    (8, 8, 'History 8'),
102    (9, 9, 'History 9'),
103    (10, 10, 'History 10');
104
105 • INSERT INTO Admin (AdminID, Username, Password, Name, Certification, PhoneNumber) VALUES
106    (1, 'admin1', 'adminpass1', 'Admin One', 'Certified', '111-222-3333'),
107    (2, 'admin2', 'adminpass2', 'Admin Two', 'Certified', '111-222-3334'),
108    (3, 'admin3', 'adminpass3', 'Admin Three', 'Certified', '111-222-3335'),
109    (4, 'admin4', 'adminpass4', 'Admin Four', NULL, '111-222-3336'),
110    (5, 'admin5', 'adminpass5', 'Admin Five', 'Certified', '111-222-3337'),
111    (6, 'admin6', 'adminpass6', 'Admin Six', NULL, '111-222-3338'),
112    (7, 'admin7', 'adminpass7', 'Admin Seven', 'Certified', '111-222-3339'),
113    (8, 'admin8', 'adminpass8', 'Admin Eight', 'Certified', '111-222-3340'),
114    (9, 'admin9', 'adminpass9', 'Admin Nine', NULL, '111-222-3341'),
115    (10, 'admin10', 'adminpass10', 'Admin Ten', 'Certified', '111-222-3342');

```

The UserLogin table tracks each user's login and logout activities. It includes LoginID, UserID, LoginTime, and LogoutTime. This query adds 10 entries, each representing a distinct user session. It captures the time each user logs in and out, providing a record of user activity within the system. The AlertInfo table is intended to store detailed historical information about alerts for each user. This section demonstrates the ability of the AlertInfo table to maintain a

history log for each alert. By inserting 10 distinct records, the table shows a relationship between users and their respective alert histories. The Admin table is designed to manage administrator accounts. This query populates the Admin table with 10 entries. Each entry represents an administrator, containing a unique combination of ID, username, password, name, certification status, and contact number. This showcases the table's capability to handle multiple admin profiles with distinct credentials and personal details.

**Figure 4:** Importing the data for the AdminLogin and AdminSettings tables:

```
INSERT INTO AdminLogin (LoginID, AdminID, LoginTime, LogoutTime) VALUES
(1, 1, '2023-11-10 08:00:00', '2023-11-10 17:00:00'),
(2, 2, '2023-11-10 09:00:00', '2023-11-10 18:00:00'),
(3, 3, '2023-11-11 08:30:00', '2023-11-11 17:30:00'),
(4, 4, '2023-11-11 09:30:00', '2023-11-11 18:30:00'),
(5, 5, '2023-11-12 08:00:00', '2023-11-12 17:00:00'),
(6, 6, '2023-11-12 09:00:00', '2023-11-12 18:00:00'),
(7, 7, '2023-11-13 08:30:00', '2023-11-13 17:30:00'),
(8, 8, '2023-11-13 09:30:00', '2023-11-13 18:30:00'),
(9, 9, '2023-11-14 08:00:00', '2023-11-14 17:00:00'),
(10, 10, '2023-11-14 09:00:00', '2023-11-14 18:00:00');

INSERT INTO AdminSettings (SettingsID, AdminID, NotificationSettings) VALUES
(1, 1, 'Email'),
(2, 2, 'SMS'),
(3, 3, 'Email, SMS'),
(4, 4, NULL),
(5, 5, 'Email'),
(6, 6, 'SMS'),
(7, 7, NULL),
(8, 8, 'Email, SMS'),
(9, 9, 'Email'),
(10, 10, 'SMS');
```

The AdminLogin table is designed to track the login and logout activities of administrators in the system. The insertion command above adds 10 distinct entries to the AdminLogin table. Each entry details a specific admin session, highlighting the precise times of logging in and out. This data is critical for understanding admin usage patterns and enhancing system security. The AdminSettings table manages the individual settings preferences for each administrator. This insertion operation populates the AdminSettings table with 10 records, each corresponding to an administrator. These records demonstrate the flexibility of the system in accommodating notification settings SMS or No SMS. This capability is essential for ensuring

that administrators can customize their working environment according to their preferences, thus improving efficiency and user experience.

**Figure 5:** Testing Data Constraints

```

14  -- Primary Key Constraint Test: Try to insert a record into the User table with an existing UserID.
15  • INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
16    (1, 'newuser', 'newpass', 'New User', '131 Main St', '123-456-7898', '1991-11-11');
17
18  -- Foreign Key Constraint Test: Insert a record in UserSettings with a UserID that doesn't exist in the User table.
19  • INSERT INTO UserSettings (SettingsID, UserID, Mode, NotificationSettings) VALUES
20    (11, 999, 'Dark', 'Email, SMS');
21
22  -- Data Type Constraint Test: Attempt to insert a non-date value into a date column in the User table
23  • INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
24    (11, 'testuser', 'testpass', 'Test User', '132 Main St', '123-456-7899', 'not a date');
25
26  -- Unique Constraint Test: Assuming you have a unique constraint on a column (like Username in User table),
27  -- try inserting a duplicate username.
28  • INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
29    (11, 'johndoe', 'uniquepass', 'Unique User', '133 Main St', '123-456-7800', '1992-12-12');

```

- The Primary Key Constraint Test: Attempts to insert a record into the User table with an existing UserID. As UserID is a primary key, it must be unique for each record. The test confirms that the database prevents duplication in primary key fields, thereby maintaining the uniqueness and integrity of each record. This test will provide this error:

Error Code: 1062. Duplicate entry '1' for key 'user.PRIMARY'

- Foreign Key Constraint Test: Here, the insertion into UserSettings includes a UserID that does not exist in the User table. This test verifies that the foreign key constraint is active and prevents the insertion of records with non-existent foreign key values, ensuring referential integrity between tables. This test provides this error:

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails ('redfoxtask`.`usersettings`, CONSTRAINT `usersettings\_ibfk\_1` FOREIGN KEY ('UserID') REFERENCES `user` ('UserID'))

- (It's a little hard to see the screenshot so here's a printout: **Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails ('redfoxtask`.`usersettings`, CONSTRAINT `usersettings\_ibfk\_1` FOREIGN KEY ('UserID') REFERENCES `user` ('UserID')) 0.000 sec** )
- Data Type Constraint Test: This command attempts to insert a non-date value into the date column (Birthday) in the User table. The test checks the enforcement of data type constraints, ensuring that each field in the database only accepts data of the correct type. This test provides this error:

Error Code: 1292. Incorrect date value: 'not a date' for column 'Birthday' at row 1

- Unique Constraint Test: This test tries to insert a duplicate Username into the User table, assuming a unique constraint on the Username column. It validates that the unique constraint is working correctly by preventing the duplication of values in specific fields, ensuring data uniqueness. This test provides this error:

Error Code: 1062. Duplicate entry 'johndoe' for key 'user.Username'

**Figure 6:** Table Verification Queries:

```

32  -- Lines that verify Tables Structures
33
34  •  SELECT * FROM User;
35  •  SELECT * FROM User WHERE Birthday > '1990-01-01' ORDER BY Birthday;
36  •  SELECT * FROM UserSettings;
37  •  SELECT User.Username, UserSettings.Mode FROM User INNER JOIN UserSettings ON User.UserID = UserSettings.UserID;
38  •  SELECT * FROM Calendar WHERE Mode = 1;
39  •  SELECT User.Name, Calendar.CalendarID FROM User RIGHT JOIN Calendar ON User.UserID = Calendar.UserID;
40  •  SELECT * FROM Task;
41  •  SELECT User.Name, Task.Title FROM User LEFT JOIN Task ON User.UserID = Task.UserID
42  UNION
43  SELECT User.Name, Task.Title FROM User RIGHT JOIN Task ON User.UserID = Task.UserID;
44  •  SELECT * FROM UserAlert ORDER BY AlertID DESC;
45  •  SELECT UserAlert.AlertMessage, User.Name FROM UserAlert INNER JOIN User ON UserAlert.UserID = User.UserID;

```

These queries all demonstrate that the tables have proper structure given that each of these queries is able to be performed successfully. This shows that data can be taken from each table and displayed/manipulated in a variety of ways, thus demonstrating the correct structure of these tables.

## Insertion Optimization:

In the development of the RedFoxTask database, we applied optimization techniques to improve the efficiency of data insertion. This optimization is particularly important in a real-world scenario where databases often handle large volumes of data, necessitating efficient data management practices. We experimented with two different methods of data insertion: single-line insertions and bulk insertions.

**Figure 7:** Single Line Insertions: (This figure ended up being really hard to see, so here's a link to it)

[https://drive.google.com/file/d/1p4T9jIAkhUD6MU4LNT1Ut2-vhL\\_Yd44i/view?usp=drive\\_link](https://drive.google.com/file/d/1p4T9jIAkhUD6MU4LNT1Ut2-vhL_Yd44i/view?usp=drive_link)

Initially, we inserted data into our tables one row at a time. This method, while straightforward, was inefficient, especially when dealing with large datasets. In our case, the single-line insertions took a total of approximately 0.016 seconds per insertion, as demonstrated in the sample code provided.

**Figure 8:** Bulk Insertions:

```

18  -- Optimized bulk insertions take a total time of 0.000 seconds (I assume this means a number
19  -- smaller than 0.000 but too small to be displayed with 3 significant figures
20  • Insert INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
21  (1, 'johndoe', 'password123', 'John Doe', '123 Main St', '123-456-7890', '1990-01-01'),
22  (2, 'janedoe', 'pass456', 'Jane Doe', '124 Main St', NULL, '1992-02-02'),
23  (3, 'mikeb', 'mikepass', 'Mike Brown', '125 Main St', '123-456-7891', '1988-03-03'),
24  (4, 'emmagreen', 'emma1234', 'Emma Green', NULL, '123-456-7892', '1995-04-04'),
25  (5, 'samw', 'samspass', 'Sam Wilson', '126 Main St', '123-456-7893', '1991-05-05'),
26  (6, 'lucyp', 'lucyword', 'Lucy Parker', '127 Main St', '123-456-7894', '1989-06-06'),
27  (7, 'davidt', 'davidpwd', 'David Taylor', NULL, '123-456-7895', '1993-07-07'),
28  (8, 'sarahc', 'sarahpass', 'Sarah Connor', '128 Main St', '123-456-7896', '1994-08-08'),
29  (9, 'rickg', 'rick123', 'Rick Grimes', '129 Main St', NULL, '1996-09-09'),
30  (10, 'annak', 'anna456', 'Anna Klein', '130 Main St', '123-456-7897', '1990-10-10');

```

To enhance performance, we then implemented bulk insertions. This method allows multiple rows to be inserted in a single command, significantly reducing the time spent on database communication overhead. The optimized bulk insertions demonstrated a remarkable improvement in execution time, taking a total of approximately 0.000 seconds. We're assuming that this means a number that is smaller than 0.000 but cannot be displayed within three significant figures.

## Normalization Check:

**Figure 9:** Calendar Table:

```

CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT PRIMARY KEY,
    UserID INT,
    Mode INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);

```

The current design of the Calendar table includes a UserID which is not part of the primary key. This could potentially violate 2NF since CalendarID doesn't uniquely determine

UserID. To normalize, I have made CalendarID and UserID a composite primary key since multiple calendars can exist per user, thus eliminating the 2nd normal form issues. (Updated code below)

```
CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT,
    UserID INT,
    Mode INT,
    PRIMARY KEY (CalendarID, UserID),
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 10: Task Table

```
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT PRIMARY KEY,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

Similar to the Calendar table, the Task table has a UserID which is not part of the primary key. Since the task is specific to a user and a calendar, I have made TaskID, UserID, and Calendar\_CalendarID a composite primary key. This ensures that each task is uniquely identified by its ID, the user it belongs to, and the calendar it is associated with, thus eliminating the 2nd normal form issues. (Updated Code below)

```
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    PRIMARY KEY (TaskID, UserID, Calendar_CalendarID),
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

## Phase 5.2: Loading Data and Performance Enhancements

### Flowchart 1:

The log in chart begins with "Start" leading to "Login". A decision point then asks if "User information correct?". If "No", the flow returns to "Login" with the note "Invalid login, try again." If "Yes", it proceeds to "Display tasks and calendar page", and then to "Finish". This flowchart describes the path that the user will take upon reaching the landing 'log in' page and the actions that will be taken to log in.

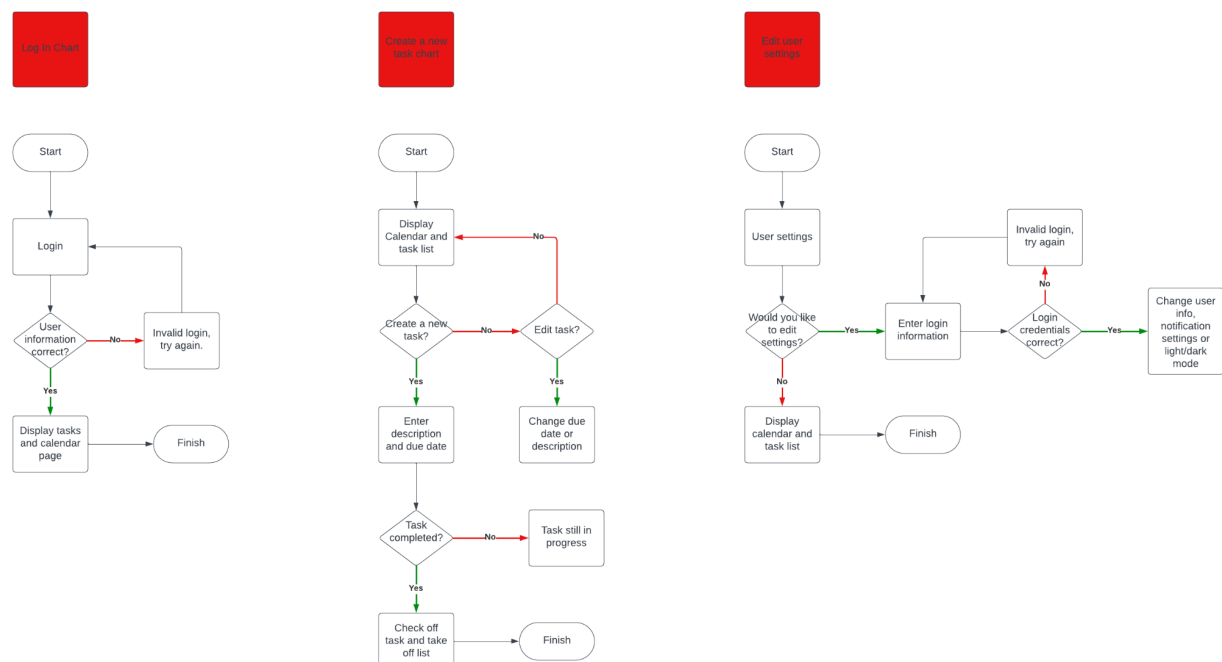
### Flowchart 2:

The create a new task chart begins with "Start" leading to "Display Calendar and task list". A decision point then asks if the user wants to "Create a new task?". If "No", another decision point asks "Edit task?". If "Yes" to editing, it moves to "Change due date or description", then back to the "Task still in progress" point. If "Yes" to create a task, it proceeds to "Enter description and due date". Then, another decision point asks if the "Task completed?". If "No", the flow leads to "Task still in progress". If "Yes", it leads to "Check off task and take off list" and then to "Finish".

### Flowchart 3:

The edit user settings chart begins with "Start", leading to "User settings". A decision point asks "Would you like to edit settings?". If "No", the flow moves to "Display calendar and task list", and then to "Finish". If "Yes", it moves to "Enter login information". A decision point follows asking if "Login credentials correct?". If "No", the flow returns to "Invalid login, try again." If "Yes", it proceeds to "Change user info, notification settings or light/dark mode", and then to "Finish".

### Flow charts:





## Views Implementation:

### View 1: Login page

The login page will have a simple form asking for the username and password. Once the user inputs their credentials and submits the form, the system will check against the LoginView to validate the credentials. The LoginView will pull the necessary credentials data from the User table. The application logic will use this view to match the input from the user against the usernames and hashed passwords stored in the database.

```
CREATE VIEW `LoginView` AS
SELECT UserID, Username, Password
FROM User;
```

### View 2: Task Management Page

The task management page will show a list of tasks from the user's calendar. It will allow the user to create new tasks, edit existing ones, and mark tasks as completed. The TaskManagementView will be used to fetch and display this information in a user-friendly format. The TaskManagementView joins the User, Calendar, and Task tables to provide a comprehensive list of tasks associated with the user. It enables Create, Read, Update, and Delete operations on tasks while ensuring that the actions are reflected in the correct calendar.

```
CREATE VIEW `TaskManagementView` AS
SELECT u.UserID, u.Username, c.CalendarID, t.TaskID, t.Title, t.Time, t.Duration, t.Description
FROM User u
JOIN Calendar c ON u.UserID = c.UserID
JOIN Task t ON u.UserID = t.UserID AND c.CalendarID = t.Calendar_CalendarID;
```

### View 3: User Settings Page

The user settings page will provide a form where users can edit their settings such as notification preferences. After making changes, the user can save the settings which will be updated in the UserSettingsView. The UserSettingsView is a simple representation of the

UserSettings table. It will be used to fetch the current settings for display in the form and update them as per the user's changes.

```
CREATE VIEW `UserSettingsView` AS  
SELECT UserID, Mode, NotificationSettings  
FROM UserSettings;
```

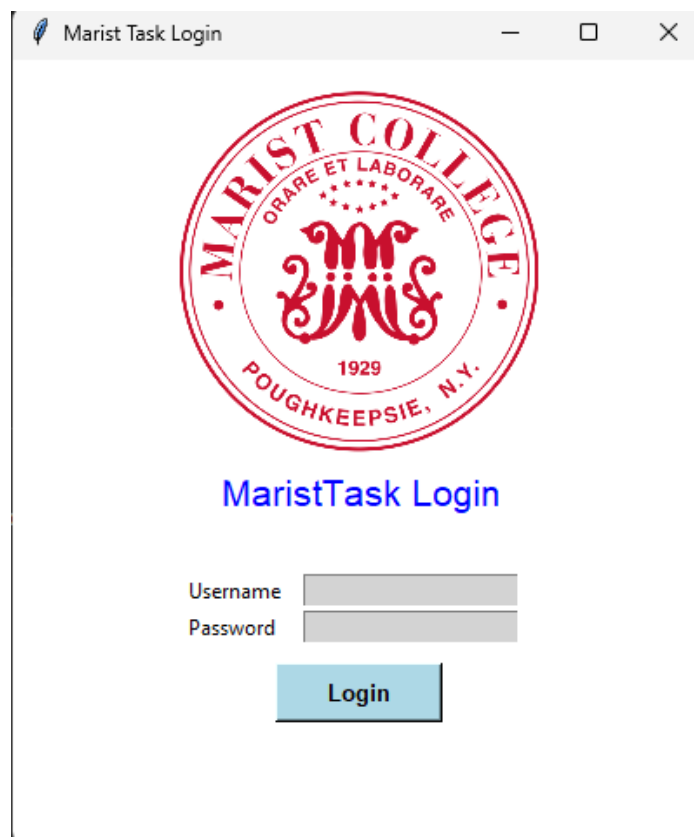
---

## Phase 6.1 Application Development:

### Login Page:

The code provided below is for a login page interface designed using Python's Tkinter library, integrated with MySQL for user authentication. The program first establishes a connection to a MySQL database named "MaristTask", using predefined credentials. It then defines a function `check_login` to authenticate users. This function checks whether the provided username and password (encrypted by the `Encrypt.encrypt_password` function) match either a regular user or an admin in the database, returning the user's role and details upon successful authentication. Upon a successful login attempt, the `open_welcome_page` function is triggered, closing the current window and opening a welcome page with user details. The login attempt (whether successful or not) is recorded using the `History.record_login` function.

The GUI is created using Tkinter widgets. It features a window titled "Marist Task Login" with a white background, displaying a logo and a welcome text. The login form includes fields for the username and password, and a customized login button styled with specific text and background colors, font settings, border thickness, and size. The login function is bound to the login button and the 'Enter' key for user convenience. The application enters a GUI loop using `mainloop` to keep the window responsive. Overall, the code integrates backend database operations with a user-friendly front-end interface, providing secure and efficient user authentication and navigation.



```
Phase06 > LoginPage.py > ...
1  import mysql.connector
2  from tkinter import *
3  from tkinter import messagebox
4  import os
5  from Functions import History, Encrypt, WelcomePage
6
7  # Connect to the database
8  db = mysql.connector.connect(
9      host="localhost",
10     user="root",
11     passwd="root",
12     database="MaristTask"
13 )
14
15 # Function to check login credentials and determine role
16 def check_login(username, input_password):
17     cursor = db.cursor()
18     encrypted_input_password = Encrypt.encrypt_password(input_password)
19
20     # Check if user is a regular user
21     user_query = "SELECT UserID, Name, Password FROM User WHERE Username = %s"
22     cursor.execute(user_query, (username,))
23     user_result = cursor.fetchone()
```

```

24
25     if user_result and encrypted_input_password == user_result[2]:
26         return True, 'user', user_result[0], user_result[1]
27
28     # Check if user is an admin
29     admin_query = "SELECT AdminID, Name, Password FROM Admin WHERE Username = %s"
30     cursor.execute(admin_query, (username,))
31     admin_result = cursor.fetchone()
32
33     if admin_result and encrypted_input_password == admin_result[2]:
34         return True, 'admin', admin_result[0], admin_result[1]
35
36     return False, None, None, None
37
38 # Function to open the welcome page
39 def open_welcome_page(role, user_id, name, db):
40     window.destroy() # Close the login window
41     WelcomePage.show_welcome_page(role, user_id, name, db) # Open the welcome page with additional details
42
43 # Function to handle the login button click
44 # Function to handle the login button click
45 def on_login_click(event=None):
46     username = entry_username.get()
47     password = entry_password.get()
48     login_success, role, user_id, name = check_login(username, password)
49
50     if login_success:
51         messagebox.showinfo("Login Success", f"You have successfully logged in as {role}!")
52         is_admin = role == 'admin'
53         History.record_login(db, user_id, is_admin) # Record the login time
54         open_welcome_page(role, user_id, name, db) # Pass the 'db' argument here
55     else:
56         messagebox.showwarning("Login Failed", "Incorrect username or password")
57
58
59
60
61 # Create the main window
62 window = Tk()
63 window.title("Marist Task Login")
64 window.geometry("400x450") # Adjust the size as needed
65 window.configure(bg='white') # Set the background color to white
66
67 # Define the relative path to the images directory
68 # Note that 'Images' is capitalized as per your folder structure
69 image_path = os.path.join(os.path.dirname(__file__), 'Images')
70
71 # Load the logo image using a relative path
72 logo_image_path = os.path.join(image_path, 'logo.png') # 'logo.png' is the image file in the Images directory
73 logo_image = PhotoImage(file=logo_image_path)
74 logo_image = logo_image.subsample(2, 2) # Adjust the subsample factors as needed
75
76 # Use the scaled image in the Label widget
77 logo_label = Label(window, image=logo_image, bg='white') # Set the background color to white
78 logo_label.pack(side=TOP, pady=(10, 0))
79

```

```

78 logo_label.pack(side=TOP, pady=(10, 0))
79
80 # Welcome label
81 welcome_text = Label(window, text="MaristTask Login", fg="blue", bg='white', font=("Helvetica", 16))
82 welcome_text.pack(side=TOP, pady=(0, 20)) # Adjust padding as needed
83
84 # Create and place the login form
85 form_frame = Frame(window, bg='white') # Set the background color to white
86 form_frame.pack(side=TOP, pady=10)
87
88 label_username = Label(form_frame, text="Username", bg='white')
89 label_username.grid(row=0, column=0, sticky=W)
90
91 entry_username = Entry(form_frame, bg='#D3D3D3')
92 entry_username.grid(row=0, column=1, padx=10)
93
94 label_password = Label(form_frame, text="Password", bg='white')
95 label_password.grid(row=1, column=0, sticky=W)
96
97 entry_password = Entry(form_frame, show="*", bg='#D3D3D3')
98 entry_password.grid(row=1, column=1, padx=10)
99
100 # Modified login button with new properties
101 button_login = Button(form_frame, text="Login", command=on_login_click,
102                      fg="black", # Text color
103                      bg="#ADD8E6", # Background color
104                      font=("Helvetica", 10, "bold"), # Font settings (size, weight)
105                      highlightthickness=4, # Border thickness
106                      width=10, height=1) # Button size
107 button_login.grid(row=2, column=0, columnspan=2, pady=10)
108
109
110 # Bind the enter key to the login function
111 window.bind('<Return>', on_login_click)
112
113 # Start the GUI loop
114 window.mainloop()
115

```

## Main Menu Page:

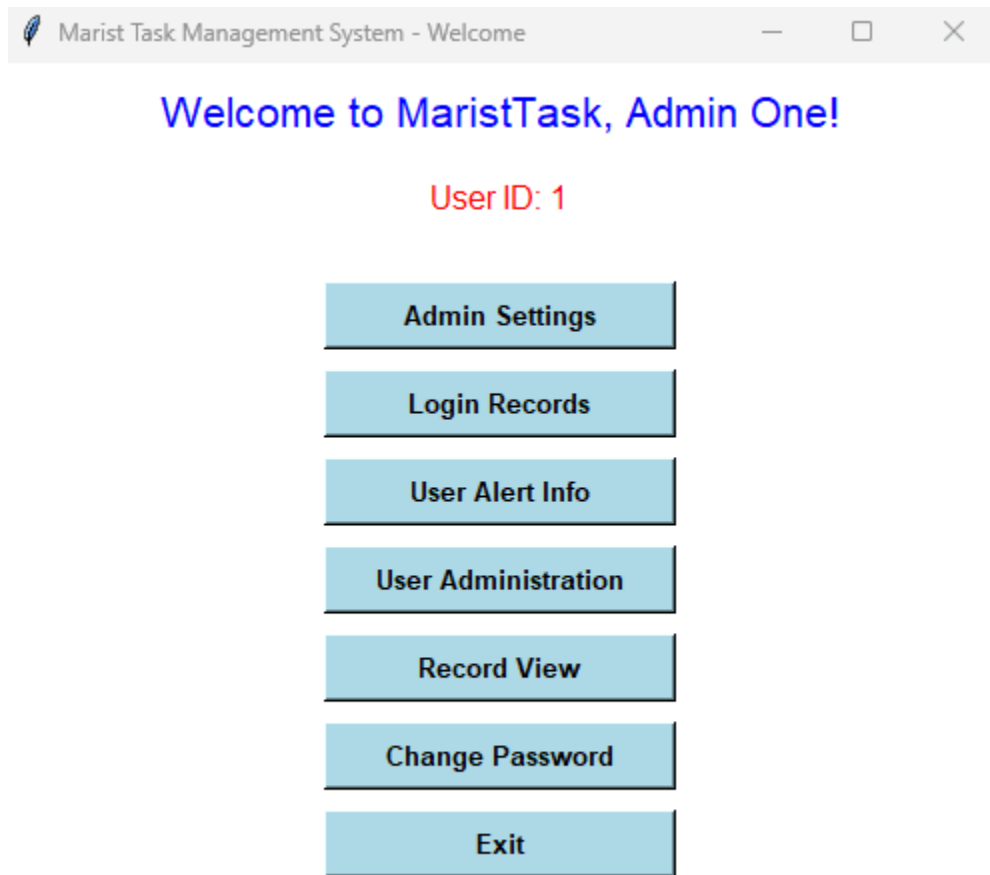
This code defines a Main Menu page for the "Marist Task Management System," using the Tkinter library in Python. The function `show_welcome_page` is designed to create a main window with a user interface tailored to the role of the logged-in user (admin or regular user). The window's title is set, and its size varies depending on the user's role—smaller for admins and larger for regular users. The background color is set to white, and the user is greeted with a welcome label displaying their name.

The UI features a series of buttons organized in a frame, with a common style defined for all buttons. These buttons are linked to various functionalities like task management (add, edit, remove, search tasks), calendar, alerts, settings, and history. For admins, additional buttons like 'Admin Settings', 'Login Records', 'User Alert Info', 'User Administration', and 'Record View' are

provided. A 'Change Password' button is available for all users. An 'Exit' button is included to close the application, and upon closing, a message is displayed thanking the user for using the software, and the logout time is recorded in the history.

The interface adapts based on the user's role, displaying different sets of buttons and functionalities. For regular users, there's also a check for alerts when the window is opened. The GUI loop starts at the end of the function, initiating the user interface. This design demonstrates a dynamic, role-based UI approach in a desktop application.







```

1  from tkinter import Tk, Label, Button, Frame, messagebox
2  from Functions import ChangePassword, UserAdministration, AddTask, EditTask, RemoveTask, SearchTask, Calendar, Alerts, Settings, History, Records
3
4  def show_welcome_page(role, user_id, name, db):
5      # Create the main window
6      welcome_window = Tk()
7      welcome_window.title("Marist Task Management System - Welcome")
8
9      # Adjust the size based on the role
10     if role == 'admin':
11         welcome_window.geometry("500x450") # Smaller size for admin
12     else:
13         welcome_window.geometry("600x750") # Larger size for regular user
14
15     welcome_window.configure(bg='white') # Set the background color to white
16
17     # Welcome label with user's name
18     welcome_label = Label(welcome_window, text=f"Welcome to MaristTask, {name}!",
19                           fg="blue", bg='white', font=("Helvetica", 16))
20     welcome_label.pack(pady=10)
21
22     # Display user ID (for debugging or reference)
23     user_id_label = Label(welcome_window, text=f"User ID: {user_id}",
24                           fg="red", bg='white', font=("Helvetica", 12))
25     user_id_label.pack(pady=5)
26
27     # Button frame
28     button_frame = Frame(welcome_window, bg='white')
29     button_frame.pack(pady=20)
30
31     # Define a common button style
32     button_style = {'fg': "black", 'bg': "#ADD8E6", 'font': ("Helvetica", 10, "bold"),
33                     'highlightthickness': 4, 'width': 20, 'height': 1}
34
35     def on_close():
36         """ Function to handle window close event """
37         is_admin = role == 'admin'
38         History.record_logout(db, user_id, is_admin) # Record the logout time
39         messagebox.showinfo("Thank You", "Thank you for using the MaristTask software!")
40         welcome_window.destroy() # Close the window
41
42     welcome_window.protocol("WM_DELETE_WINDOW", on_close) # Bind the close event
43
44     # Role-specific features and buttons
45     if role == 'admin':
46         # Admin-specific buttons
47         btn_admin_settings = Button(button_frame, text="Admin Settings", **button_style,
48                                     command=lambda: Settings.settings_window(db, user_id, role))
49         btn_admin_settings.grid(row=0, column=0, padx=10, pady=5)

```

```

50
51     btn_admin_login_history = Button(button_frame, text="Login Records", **button_style,
52                                     command=lambda: History.show_admin_login_records(db))
53     btn_admin_login_history.grid(row=1, column=0, padx=10, pady=5)
54
55     btn_user_alert_info = Button(button_frame, text="User Alert Info", **button_style,
56                                 command=lambda: Alerts.show_user_alert_info(db))
57     btn_user_alert_info.grid(row=2, column=0, padx=10, pady=5)
58
59     btn_user_admin = Button(button_frame, text="User Administration", **button_style,
60                             command=lambda: UserAdministration.user_administration(db))
61     btn_user_admin.grid(row=3, column=0, padx=10, pady=5)
62
63     btn_record_view = Button(button_frame, text="Record View", **button_style,
64                              command=lambda: Records.record_view(db))
65     btn_record_view.grid(row=4, column=0, padx=10, pady=5)
66
67     else:
68         # User-specific buttons
69         btn_add_task = Button(button_frame, text="Add Task", **button_style,
70                               command=lambda: AddTask.add_task(db, user_id))
71         btn_add_task.grid(row=0, column=0, padx=10, pady=5)
72
73         btn_edit_task = Button(button_frame, text="Edit Task", **button_style,
74                                command=lambda: EditTask.edit_task(db, user_id))
75         btn_edit_task.grid(row=1, column=0, padx=10, pady=5)
76
77         btn_remove_task = Button(button_frame, text="Remove Task", **button_style,
78                                  command=lambda: RemoveTask.remove_task(db, user_id))
79         btn_remove_task.grid(row=2, column=0, padx=10, pady=5)
80
81         btn_search_task = Button(button_frame, text="Search Task", **button_style,
82                                  command=lambda: SearchTask.search_task(db, user_id))
83         btn_search_task.grid(row=3, column=0, padx=10, pady=5)
84
85         default_calendar_id = 1
86         btn_calendar = Button(button_frame, text="Calendar", **button_style,
87                                command=lambda: Calendar.show_calendar(db, user_id, default_calendar_id))
88         btn_calendar.grid(row=4, column=0, padx=10, pady=5)
89
90         btn_alerts = Button(button_frame, text="Alerts", **button_style,
91                              command=lambda: Alerts.show_user_alerts(db, user_id))
92         btn_alerts.grid(row=5, column=0, padx=10, pady=5)
93
94         btn_user_settings = Button(button_frame, text="User Settings", **button_style,
95                                    command=lambda: Settings.settings_window(db, user_id, role))
96         btn_user_settings.grid(row=6, column=0, padx=10, pady=5)

```

```

96         btn_user_settings.grid(row=6, column=0, padx=10, pady=5)
97
98         btn_user_login_history = Button(button_frame, text="User Login History", **button_style,
99                                         command=lambda: History.show_login_history(db, user_id, False))
100         btn_user_login_history.grid(row=7, column=0, padx=10, pady=5)
101
102         # Change Password Button (for all users)
103         btn_change_password = Button(button_frame, text="Change Password", **button_style,
104                                     command=lambda: ChangePassword.change_password(db, user_id, role))
105         btn_change_password.grid(row=8 if role == 'admin' else 9, column=0, padx=10, pady=5)
106
107         # Exit button
108         btn_exit = Button(button_frame, text="Exit", command=on_close, **button_style)
109         btn_exit.grid(row=9 if role == 'admin' else 10, column=0, padx=10, pady=5)
110
111         # Check for alerts (if user)
112         if role != 'admin':
113             Alerts.welcome_alerts(db, user_id)
114
115         # Start the GUI loop
116         welcome_window.mainloop()
117


```

## Action Pages:

### Figure 1: Add Task

The code below defines a Python function `add_task` that creates a graphical user interface (GUI) for adding tasks to a database, using Tkinter for the GUI components and MySQL for database operations. The function is designed to be part of a larger application where each user can add tasks to their calendar. The GUI window, titled "Add a Task", contains several input fields for task details: title, start date, start time (with AM/PM selection), duration (in days), description, and a calendar ID. These fields are implemented using labels and entry widgets, with a special frame for the start time and AM/PM radio buttons. The start date and time inputs are validated and formatted to fit the required database format. An error message is displayed if the date or time format is invalid.

The user can submit the task by clicking the "Add Task" button or pressing the Enter key. The submission process involves checking that all fields are filled, executing an SQL query to insert the task into the database, and handling any potential errors. On successful addition, a success message is shown, and the add task window is closed. There's also a "Return to Main Page" button for closing the window without adding a task.

 Add a Task—□×

## Add a Task!

Title

Start Date (YYYY-MM-DD)

Start Time (HH:MM)

☐ AM ☒ PM

Duration (days)

Description

Calendar ID

Add Task

Return to Main Page

```

1 import mysql.connector
2 from tkinter import Radiobutton, Tk, Label, Entry, Button, messagebox, StringVar, Frame
3 import datetime
4
5 def add_task(db, user_id):
6     # Function to handle the task submission process.
7     def submit_task(event=None):
8         # Retrieve values from the form fields.
9         title = entry_title.get()
10        start_date = entry_start_date.get()
11        start_time = entry_start_time.get() + " " + am_pm.get()
12        duration = entry_duration.get()
13        description = entry_description.get()
14        calendar_id = entry_calendar_id.get()
15
16        # Attempt to format and validate the start date and time.
17        try:
18            formatted_start_time = datetime.datetime.strptime(f"{start_date} {start_time}", '%Y-%m-%d %I:%M %p')
19            formatted_start_time_str = formatted_start_time.strftime('%Y-%m-%d %H:%M:%S')
20        except ValueError:
21            messagebox.showerror("Error", "Invalid date or time format")
22            return
23
24        # Check if all form fields are filled.
25        if not title or not start_date or not start_time or not duration or not description or not calendar_id:
26            messagebox.showerror("Error", "All fields are required")
27            return
28
29        # Create a database cursor for executing SQL queries.
30        cursor = db.cursor()
31        # SQL query to insert a new task into the database.
32        insert_query = """
33        INSERT INTO Task (UserID, Title, Time, Duration, Description, Calendar_CalendarID)
34        VALUES (%s, %s, %s, %s, %s, %s)
35        """
36        try:
37            # Execute the insert query with user-provided values.
38            cursor.execute(insert_query, (user_id, title, formatted_start_time_str, int(duration) * 1440, description, calendar_id))
39            db.commit()
40            messagebox.showinfo("Success", "Task added successfully")
41            add_window.destroy() # Close the window after successful task addition.
42        except mysql.connector.Error as err:
43            messagebox.showerror("Error", f"An error occurred: {err}")
44        finally:
45            cursor.close()
46
47        # Set up the GUI window for adding a task.
48        add_window = Tk()
49        add_window.title("Add a Task")
50        add_window.geometry("400x500")
51        add_window.configure(bg='white')
52
53        label_main_title = Label(add_window, text="Add a Task!", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
54        label_main_title.pack(pady=(10, 20))
55
56        # Widgets for task title input.
57        label_title = Label(add_window, text="Title", bg='white')
58        label_title.pack(pady=(5, 5))
59        entry_title = Entry(add_window, bg='#D3D3D3')
60        entry_title.pack()
61
62        # Widgets for task start date input.
63        label_start_date = Label(add_window, text="Start Date (YYYY-MM-DD)", bg='white')
64        label_start_date.pack(pady=5)
65        entry_start_date = Entry(add_window, bg='#D3D3D3')
66        entry_start_date.pack()

```

```

64     label_start_date.pack(pady=5)
65     entry_start_date = Entry(add_window, bg='#D3D3D3')
66     entry_start_date.pack()
67
68     # Frame for time input and AM/PM selection.
69     time_frame = Frame(add_window, bg='white')
70     time_frame.pack(pady=5)
71
72     label_start_time = Label(time_frame, text="Start Time (HH:MM)", bg='white')
73     label_start_time.pack(side="left")
74     entry_start_time = Entry(time_frame, bg='#D3D3D3', width=10)
75     entry_start_time.pack(side="left")
76
77     # Radio buttons for AM/PM selection.
78     am_pm = StringVar(value="AM")
79     rb_am = Radiobutton(time_frame, text="AM", variable=am_pm, value="AM", bg='white')
80     rb_am.pack(side="left")
81     rb_pm = Radiobutton(time_frame, text="PM", variable=am_pm, value="PM", bg='white')
82     rb_pm.pack(side="left")
83
84     # Widgets for task duration input.
85     label_duration = Label(add_window, text="Duration (days)", bg='white')
86     label_duration.pack(pady=5)
87     entry_duration = Entry(add_window, bg='#D3D3D3')
88     entry_duration.pack()
89
90     # Widgets for task description input.
91     label_description = Label(add_window, text="Description", bg='white')
92     label_description.pack(pady=5)
93     entry_description = Entry(add_window, bg='#D3D3D3')
94     entry_description.pack()
95
96     # Widgets for calendar ID input.
97     label_calendar_id = Label(add_window, text="Calendar ID", bg='white')
98     label_calendar_id.pack(pady=5)
99     entry_calendar_id = Entry(add_window, bg='#D3D3D3')
100    entry_calendar_id.pack()
101
102    # Button to submit the task information.
103    button_submit = Button(add_window, text="Add Task", fg="black", bg="#ADD8E6",
104                          font=("Helvetica", 10, "bold"), highlightthickness=4, command=submit_task)
105    button_submit.pack(pady=20)
106
107    # Button to close the add task window.
108    button_return = Button(add_window, text="Return to Main Page", fg="black", bg="#ADD8E6",
109                          font=("Helvetica", 10, "bold"), highlightthickness=4, command=add_window.destroy)
110    button_return.place(x=10, y=450)
111
112    # Enable submission of the task by pressing the Enter key.
113    add_window.bind('<Return>', submit_task)
114
115    add_window.mainloop()

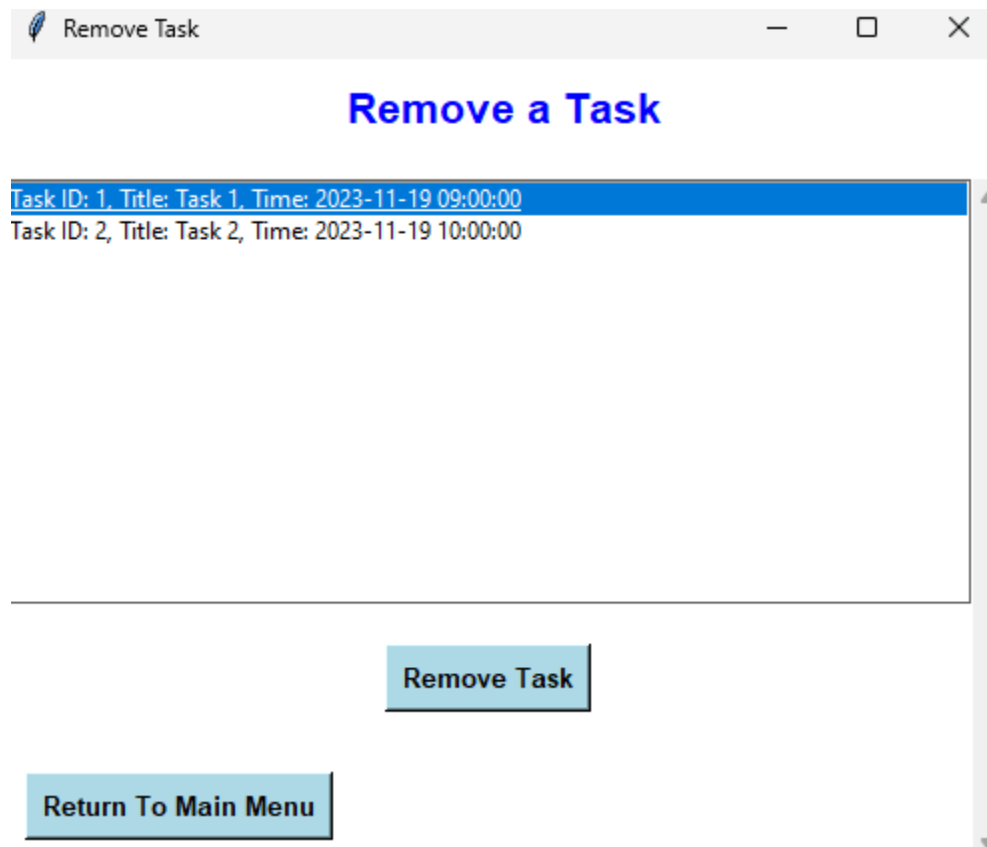
```

**Figure 2:** Remove Task

This code describes a Python application for managing tasks, with a focus on a feature that allows users to remove tasks. It uses mysql.connector to interact with a MySQL database and tkinter for the graphical user interface (GUI). The central functionality is encapsulated in the `remove_task` function, which creates a window where tasks associated with a given user ID can be viewed and selectively removed. This function has two inner functions: `refresh_task_list` and `select_task_to_remove`. `refresh_task_list` queries the database to retrieve tasks for the specified user, displaying them in a Listbox widget with an associated scrollbar for navigation. Each task's

ID, title, and time are displayed, and the IDs are stored in a list for further operations. The `select_task_to_remove` function allows the user to select a task from the list and delete it from the database after confirmation, with error handling for any issues during the deletion process.

The GUI is composed of a main window titled "Remove Task", a title label, a task display Listbox with a scrollbar, and buttons for removing a selected task and returning to the main menu. The layout is designed for ease of use, with clear labels and a simple, intuitive interface. The application emphasizes user interaction and confirmation for task removal, ensuring a user-friendly experience with necessary precautions against accidental deletions.



```

1  import mysql.connector
2  from tkinter import Tk, Label, Button, Frame, Listbox, Scrollbar, messagebox
3
4  def remove_task(db, user_id):
5      # Function to update the task list in the GUI
6      def refresh_task_list():
7          # Clear existing items in the listbox
8          listbox_tasks.delete(0, 'end')
9          # Clear the list that tracks task IDs
10         task_ids.clear()
11         # Create a database cursor for querying
12         cursor = db.cursor()
13         # Retrieve tasks specific to the given user ID
14         cursor.execute("SELECT TaskID, Title, Time FROM Task WHERE UserID = %s", (user_id,))
15         # Populate the listbox with tasks and store their IDs
16         for task_id, title, time in cursor:
17             listbox_tasks.insert("end", f"Task ID: {task_id}, Title: {title}, Time: {time}")
18             task_ids.append(task_id)
19         # Close the cursor to free database resources
20         cursor.close()
21
22     # Function to handle task removal
23     def select_task_to_remove():
24         # Get the selected item's index from the listbox
25         selected_index = listbox_tasks.curselection()
26         # Check if an item is selected
27         if not selected_index:
28             messagebox.showwarning("Warning", "Please select a task to remove")
29             return
30         # Retrieve the task ID corresponding to the selected item
31         task_id = task_ids[selected_index[0]]
32
33         # Ask for confirmation before deleting the task
34         if messagebox.askyesno("Confirm Deletion", "Are you sure you want to delete this task?"):
35             cursor = db.cursor()
36             delete_query = "DELETE FROM Task WHERE TaskID = %s"
37             try:
38                 # Execute the deletion query
39                 cursor.execute(delete_query, (task_id,))
40                 db.commit()
41                 messagebox.showinfo("Success", "Task removed successfully")
42                 # Refresh the list to reflect the deletion
43                 refresh_task_list()
44             except mysql.connector.Error as err:
45                 messagebox.showerror("Error", f"An error occurred: {err}")
46             finally:
47                 # Always close the cursor after the operation
48                 cursor.close()
49
50     # Create the main window using Tkinter
51     main_window = Tk()
52     main_window.title("Remove Task")
53     main_window.geometry("500x400")
54     main_window.configure(bg='white')
55
56     # Setup the title label
57     label_main_title = Label(main_window, text="Remove a Task", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
58     label_main_title.pack(pady=(10, 20))
59
60     # Initialize a scrollbar for the listbox

```



```


60     # Initialize a scrollbar for the listbox
61     scrollbar = Scrollbar(main_window)
62     scrollbar.pack(side="right", fill="y")
63
64     # Create a listbox to display tasks and attach the scrollbar to it
65     listbox_tasks = Listbox(main_window, yscrollcommand=scrollbar.set)
66     task_ids = [] # List to store task IDs
67     refresh_task_list() # Populate the listbox with tasks
68
69     listbox_tasks.pack(fill="both", expand=True)
70     scrollbar.config(command=listbox_tasks.yview)
71
72     # Button to trigger task removal
73     button_remove = Button(main_window, text="Remove Task", fg="black", bg="#ADD8E6",
74                             font=("Helvetica", 10, "bold"), highlightthickness=4, command=select_task_to_remove)
75     button_remove.pack(pady=20)
76
77     # Button to return to the main menu
78     button_return = Button(main_window, text="Return To Main Menu", fg="black", bg="#ADD8E6",
79                             font=("Helvetica", 10, "bold"), highlightthickness=4, command=main_window.destroy)
80     button_return.pack(side='left', padx=10, pady=10)
81
82     # Start the Tkinter event loop
83     main_window.mainloop()
84

```

**Figure 3:** Edit Task

The below code defines a Python application for editing tasks in a database, using Tkinter for the graphical user interface and mysql.connector for database operations. The program is designed for a user to select and edit tasks from a database associated with their user ID. The main function, `edit_task`, initializes the user interface and provides functionalities for listing, selecting, and editing tasks. It consists of several nested functions: `refresh_task_list`: This function populates a list box with tasks retrieved from the database. Each task includes its ID, title, and time. This list aids users in selecting a task to edit. `select_task_to_edit`: Triggered when a user selects a task from the list. It opens a new window for editing the chosen task. If no task is selected, it shows a warning message. `edit_selected_task`: This function creates an interface for editing the selected task. It displays current task details in various entry fields, including title, start date and time, duration, description, and calendar ID. The start time is formatted for ease of editing, and the duration is converted from minutes to days. `finalize_edits`: This function saves the edited task back to the database. It validates and formats the new start time and converts the duration back to minutes. If successful, it updates the task in the database and refreshes the task list; otherwise, it displays an error message.

The main window (`main_window`) contains a listbox for displaying tasks, a scrollbar for navigation, and buttons for selecting a task to edit and returning to the main menu. The edit window (`edit_window`) offers fields for editing the task's details and a button to finalize the edits.

 Choose a Task to Edit


Choose a Task to Edit

Task ID: 1, Title: Task 1, Time: 2023-11-19 09:00:00

Task ID: 2, Title: Task 2, Time: 2023-11-19 10:00:00

Select Task

Return To Main Menu

 Edit Task

Edit a Task!

Title

Task 1

Start Date (YYYY-MM-DD)

2023-11-19

Start Time (HH:MM)

09:00

☐ AM ☒ PM

Duration (days)

1

Description

Description for Task 1

Calendar ID

1

Finalize Edits

```

1 import mysql.connector
2 from tkinter import Radiobutton, Tk, Label, Entry, Button, messagebox, StringVar, Frame, Listbox, Scrollbar, Toplevel
3 import datetime
4
5 def edit_task(db, user_id):
6     # Function to refresh the task list displayed in the main window.
7     def refresh_task_list():
8         listbox_tasks.delete(0, 'end') # Clear existing items in the listbox.
9         task_ids.clear() # Clear the task ID list.
10        cursor = db.cursor() # Create a new cursor for database operations.
11        cursor.execute("SELECT TaskID, Title, Time FROM Task WHERE UserID = %s", (user_id,))
12        # Populate the listbox with tasks and store their IDs for later use.
13        for task_id, title, time in cursor:
14            listbox_tasks.insert("end", f"Task ID: {task_id}, Title: {title}, Time: {time}")
15            task_ids.append(task_id)
16        cursor.close()
17
18    # Function to handle the selection of a task to edit.
19    def select_task_to_edit():
20        selected_index = listbox_tasks.curselection() # Get the index of the selected task.
21        if not selected_index: # Check if a task is selected.
22            messagebox.showwarning("Warning", "Please select a task to edit")
23            return
24        task_id = task_ids[selected_index[0]] # Retrieve the ID of the selected task.
25        edit_window = Toplevel(main_window) # Create a new top-level window for editing.
26        edit_selected_task(edit_window, task_id) # Call function to handle editing.
27
28    # Function to edit the selected task.
29    def edit_selected_task(edit_window, task_id):
30        cursor = db.cursor()
31        # Fetch existing details of the selected task from the database.
32        cursor.execute("SELECT Title, Time, Duration, Description, Calendar_CalendarID FROM Task WHERE TaskID = %s", (task_id,))
33        task = cursor.fetchone()
34        cursor.close()
35
36        # Unpack the task details.
37        title, time, duration, description, calendar_id = task
38        # Format the start date and time for display.
39        start_date = time.strftime('%Y-%m-%d')
40        start_time = time.strftime('%I:%M %p')
41        duration_days = duration // 1440 # Convert duration from minutes to days.
42
43    # Function to finalize and save the edits to the task.
44    def finalize_edits():
45        # Fetch new task details from the entry fields.
46        new_title = entry_title.get()
47        new_start_date = entry_start_date.get()
48        new_start_time = entry_start_time.get() + " " + am_pm.get()
49        new_duration_days = entry_duration.get()
50        new_description = entry_description.get()
51        new_calendar_id = entry_calendar_id.get()
52
53        # Attempt to format and validate the new start time.
54        try:
55            formatted_start_time = datetime.datetime.strptime(f"{new_start_date} {new_start_time}", '%Y-%m-%d %I:%M %p')
56            formatted_duration = int(new_duration_days) * 1440 # Convert days back to minutes.
57        except ValueError:
58            messagebox.showerror("Error", "Invalid date or time format")
59            return
60
61        # Prepare the SQL query for updating the task.

```

```

61     # Prepare the SQL query for updating the task.
62     update_query = """
63         UPDATE Task SET Title = %s, Time = %s, Duration = %s, Description = %s, Calendar_CalendarID = %s
64         WHERE TaskID = %s
65     """
66     try:
67         cursor = db.cursor()
68         # Execute the update query with the new task details.
69         cursor.execute(update_query, (new_title, formatted_start_time, formatted_duration, new_description, int(new_calendar_id), task_id))
70         db.commit() # Commit the changes to the database.
71         messagebox.showinfo("Success", "Task updated successfully")
72         edit_window.destroy() # Close the edit window.
73         refresh_task_list() # Refresh the task list to show updated details.
74     except mysql.connector.Error as err:
75         messagebox.showerror("Error", f"An error occurred: {err}")
76     finally:
77         cursor.close()
78
79     # Setting up the edit window layout with entry fields for task details.
80     edit_window.title("Edit Task")
81     edit_window.geometry("500x500")
82     edit_window.configure(bg='white')
83
84     # Main title caption
85     label_main_title = Label(edit_window, text="Edit a Task!", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
86     label_main_title.pack(pady=(10, 20))
87
88     # Task Title
89     label_title = Label(edit_window, text="Title", bg='white')
90     label_title.pack(pady=(5, 5))
91     entry_title = Entry(edit_window, bg='#D3D3D3')
92     entry_title.insert(0, title)
93     entry_title.pack()
94
95     # Task Start Date
96     label_start_date = Label(edit_window, text="Start Date (YYYY-MM-DD)", bg='white')
97     label_start_date.pack(pady=5)
98     entry_start_date = Entry(edit_window, bg='#D3D3D3')
99     entry_start_date.insert(0, start_date)
100     entry_start_date.pack()
101
102     # Time Frame
103     time_frame = Frame(edit_window, bg='white')
104     time_frame.pack(pady=5)
105
106     # Task Start Time
107     label_start_time = Label(time_frame, text="Start Time (HH:MM)", bg='white')
108     label_start_time.pack(side="left")
109     entry_start_time = Entry(time_frame, bg='#D3D3D3', width=10)
110     entry_start_time.insert(0, start_time.split()[0])
111     entry_start_time.pack(side="left")
112
113     # AM/PM Selection
114     am_pm = StringVar(value=start_time.split()[1])
115     Radiobutton(time_frame, text="AM", variable=am_pm, value="AM", bg='white', command=lambda: am_pm.set("AM")).pack(side="left")
116     Radiobutton(time_frame, text="PM", variable=am_pm, value="PM", bg='white', command=lambda: am_pm.set("PM")).pack(side="left")
117
118     # Task Duration in Days
119     label_duration = Label(edit_window, text="Duration (days)", bg='white')
120     label_duration.pack(pady=5)

```

```

121     entry_duration = Entry(edit_window, bg='#D3D3D3')
122     entry_duration.insert(0, duration_days)
123     entry_duration.pack()
124
125     # Task Description
126     label_description = Label(edit_window, text="Description", bg='white')
127     label_description.pack(pady=5)
128     entry_description = Entry(edit_window, bg='#D3D3D3')
129     entry_description.insert(0, description)
130     entry_description.pack()
131
132     # Calendar ID
133     label_calendar_id = Label(edit_window, text="Calendar ID", bg='white')
134     label_calendar_id.pack(pady=5)
135     entry_calendar_id = Entry(edit_window, bg='#D3D3D3')
136     entry_calendar_id.insert(0, calendar_id)
137     entry_calendar_id.pack()
138
139     # Finalize Edits button
140     button_finalize = Button(edit_window, text="Finalize Edits", fg="black", bg="#ADD8E6",
141                             font=("Helvetica", 10, "bold"), highlightthickness=4, command=finalize_edits)
142     button_finalize.pack(pady=20)
143
144     # Main window for task selection
145     main_window = Tk()
146     main_window.title("Choose a Task to Edit")
147     main_window.geometry("500x400")
148     main_window.configure(bg='white')
149
150     # Title caption
151     label_main_title = Label(main_window, text="Choose a Task to Edit", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
152     label_main_title.pack(pady=(10, 20))
153
154     # Listbox with tasks
155     scrollbar = Scrollbar(main_window)
156     scrollbar.pack(side="right", fill="y")
157
158     listbox_tasks = Listbox(main_window, yscrollcommand=scrollbar.set)
159     task_ids = []
160     refresh_task_list()
161
162     listbox_tasks.pack(fill="both", expand=True)
163     scrollbar.config(command=listbox_tasks.yview)
164
165     # Select Task button
166     button_select = Button(main_window, text="Select Task", fg="black", bg="#ADD8E6",
167                             font=("Helvetica", 10, "bold"), highlightthickness=4, command=select_task_to_edit)
168     button_select.pack(pady=20)
169
170     def return_to_main_menu():
171         main_window.destroy()
172
173     button_return = Button(main_window, text="Return To Main Menu", fg="black", bg="#ADD8E6",
174                             font=("Helvetica", 10, "bold"), highlightthickness=4, command=return_to_main_menu)
175     button_return.pack(side='bottom', anchor='w', padx=10, pady=10) # Placed at bottom left
176
177     main_window.mainloop()

```

**Figure 4: Search Task**

This code defines a `search_task` function for a task management application, utilizing a MySQL database and a graphical user interface built with Tkinter in Python. The primary purpose of the function is to allow users to search for tasks based on various criteria and display detailed information about these tasks. The `search_task` function, designed for a specific user (`user_id`), incorporates several key features:

- **Search Functionality:** Users can input search criteria in a text entry field. The search encompasses multiple fields in the Task table, such as title, description, calendar ID, task ID, time, and duration. The results are fetched from the database and displayed in a Treeview widget, allowing users to view summarized task details.
- **Task Detail Display:** Clicking on a task in the search results opens a new window displaying detailed information about the task. This includes the task's title, start and end times, duration (converted from minutes to days), description, and associated calendar ID.
- **Data Handling and Display:** The application connects to a MySQL database to retrieve task data based on the search query. It uses functions to calculate the end time of tasks based on their duration and start time and employs a Treeview with a scrollbar for result display.
- **User Interface Components:** The Tkinter library is used to create a user-friendly interface, including labels, entry fields, buttons, and frames. There's a main window titled "Search Tasks," where users can input search terms, view results, and access detailed task information.
- **Navigation and Layout:** The layout is organized with clear labels and buttons for searching tasks, viewing task details, and returning to the main menu. The application is designed to be interactive and easy to navigate.

Search Tasks

Search Tasks

Search

TaskID	Title	Start Time	End Time	Duration (Days)	Description	Calendar ID
1	Task 1	2023-11-19 09:00:00	2023-11-20 09:00:00	1	Description for Task 1	1
2	Task 2	2023-11-19 10:00:00	2023-11-21 10:00:00	2	Description for Task 2	1

View Task Details

Return To Main Menu

```

1 import mysql.connector
2 from tkinter import Tk, Label, Entry, Button, Frame, Toplevel, messagebox
3 from tkinter.ttk import Treeview, Scrollbar
4 import datetime
5
6 def search_task(db, user_id):
7     # Function to calculate the end time of a task based on its start time and duration
8     def calculate_end_date(start_time, duration):
9         end_time = start_time + datetime.timedelta(minutes=duration)
10        return end_time
11
12    # Function to display detailed information of a specific task in a new window
13    def display_task_details(task_id):
14        # Creating a new top-level window for task details
15        details_window = Toplevel(main_window)
16        details_window.title("Task Information")
17        details_window.geometry("400x500")
18        details_window.configure(bg='white')
19
20        # Querying the database for task details using the task ID
21        cursor = db.cursor()
22        cursor.execute("SELECT Title, Time, Duration, Description, Calendar_CalendarID FROM Task WHERE TaskID = %s", (task_id,))
23        task = cursor.fetchone()
24        cursor.close()
25
26        # Checking if task details are found and displaying them
27        if task:
28            title, time, duration, description, calendar_id = task
29            duration_days = duration // 1440 # Converting duration from minutes to days
30            end_date = calculate_end_date(time, duration)
31
32            # Creating and configuring frames and labels to display task details
33            header_frame = Frame(details_window, bg='white')
34            header_frame.pack(pady=(10, 20), padx=10, fill='x')
35            Label(header_frame, text="Task Information", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(side='left')
36
37            details_frame = Frame(details_window, bg='white')
38            details_frame.pack(pady=(5, 10), padx=10, fill='x')
39
40            # Labels and values for each task detail
41            labels = ["Title", "Start Time", "End Time", "Duration (days)", "Description", "Calendar ID", "Task ID"]
42            values = [title, time.strftime("%Y-%m-%d %H:%M"), end_date.strftime("%Y-%m-%d %H:%M"), str(duration_days), description, str(calendar_id), str(task_id)]
43
44            # Displaying each label and its corresponding value
45            for i in range(len(labels)):
46                row_frame = Frame(details_frame, bg='white')
47                row_frame.pack(fill='x', pady=2)
48                Label(row_frame, text=f"{labels[i]}:", width=15, anchor='w', bg='white', font=("Helvetica", 10, "bold")).pack(side='left')
49                Label(row_frame, text=values[i], width=25, anchor='w', bg='white', font=("Helvetica", 10)).pack(side='left')
50
51    # Function to execute search based on user input and display results
52    def search():
53        # Getting user input for the search
54        query = search_entry.get()
55
56        # Preparing and executing the search query
57        cursor = db.cursor()
58        search_query = """
59        SELECT TaskID, Title, Time, Duration, Description, Calendar_CalendarID FROM Task
60        WHERE UserID = %s AND
61        (Title LIKE %s OR
62        Description LIKE %s OR
63        Calendar_CalendarID LIKE %s OR
64        TaskID LIKE %s OR
65        Time LIKE %s OR

```



```

65         Time LIKE %s OR
66         Duration LIKE %s OR
67         ADDDATE(Time, INTERVAL Duration MINUTE) LIKE %s)
68     """
69     cursor.execute(search_query, (user_id, f'{query}%', f'{query}%', f'{query}%', f'{query}%', f'{query}%', f'{query}%', f'{query}%'))
70     tasks = cursor.fetchall()
71     cursor.close()
72
73     # Clearing previous search results from the treeview
74     for i in tree.get_children():
75         tree.delete(i)
76
77     # Populating the treeview with the search results
78     for task in tasks:
79         task_id, title, start_time, duration, description, calendar_id = task
80         duration_days = duration // 1440 # Convert minutes to days
81         end_time = calculate_end_date(start_time, duration)
82         tree.insert("", "end", values=(task_id, title, start_time, end_time, duration_days, description, calendar_id))
83
84     # Setting up the main window for task search
85     main_window = Tk()
86     main_window.title("Search Tasks")
87     main_window.geometry("900x600")
88     main_window.configure(bg='white')
89
90     # Configuring and placing search bar, entry, and button in the main window
91     Label(main_window, text="Search Tasks", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))
92     search_entry = Entry(main_window, width=50)
93     search_entry.pack(pady=5)
94     Button(main_window, text="Search", command=search, bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(pady=10)
95
96     # Configuring and adding a treeview for displaying search results
97     tree = Treeview(main_window, columns=("TaskID", "Title", "Start Time", "End Time", "Duration (Days)", "Description", "Calendar ID"), show='headings')
98     tree.heading("TaskID", text="TaskID")
99     tree.heading("Title", text="Title")
100     tree.heading("Start Time", text="Start Time")
101     tree.heading("End Time", text="End Time")
102     tree.heading("Duration (Days)", text="Duration (Days)")
103     tree.heading("Description", text="Description")
104     tree.heading("Calendar ID", text="Calendar ID")
105     tree.column("TaskID", width=50, anchor='center')
106     tree.column("Title", width=75, anchor='center')
107     tree.column("Start Time", width=150, anchor='center')
108     tree.column("End Time", width=150, anchor='center')
109     tree.column("Duration (Days)", width=100, anchor='center')
110     tree.column("Description", width=200, anchor='center')
111     tree.column("Calendar ID", width=80, anchor='center')
112     tree.pack(expand=True, fill='both', padx=10, pady=10)
113
114     # Adding a scrollbar for the treeview
115     scrollbar = Scrollbar(main_window, command=tree.yview)
116     tree.configure(yscrollcommand=scrollbar.set)
117     scrollbar.pack(side='right', fill='y')
118
119     # Button to view detailed information of a selected task
120     Button(main_window, text="View Task Details", command=lambda: display_task_details(tree.item(tree.selection())['values'][0]), bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(pady=10)
121
122     # Button to return to the main menu
123     Button(main_window, text="Return To Main Menu", command=main_window.destroy, bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(side='left', pady=10)
124
125     # Initiating the main window's event loop
126     main_window.mainloop()
127

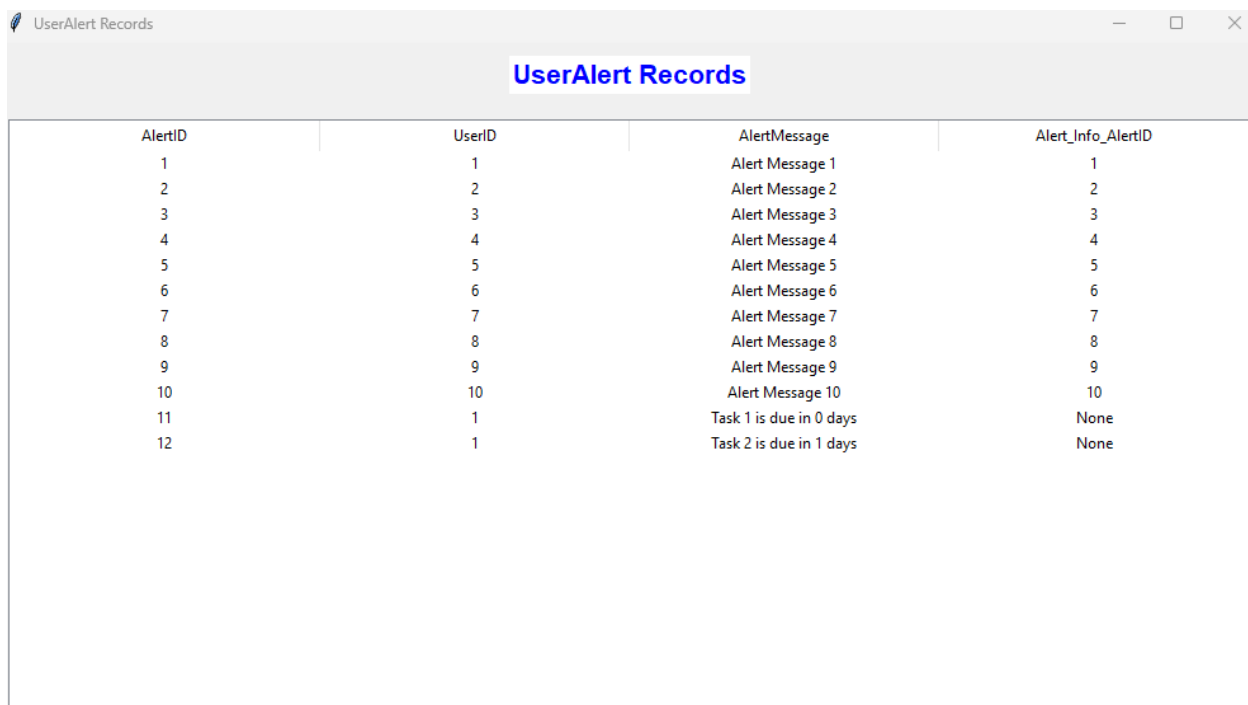
```

## Phase 6.2:

**Figure 5:** Print All Data


The below code implements a graphical user interface (GUI) using Python's Tkinter library to display data from a MySQL database in a user-friendly manner. The main function, `record_view(db)`, creates a window where users can select different database tables to view their records. Upon running, the `record_view` function opens a main window titled "Record View," offering a selection of database tables like 'User', 'UserSettings', 'Calendar', and others, each with their specific columns like 'UserID', 'Username', 'Password', etc. For each table, there's a button which, when clicked, opens a new window displaying all records from that table. This is achieved through the nested function `show_table_data(table_name, columns)`, which creates a Toplevel window with a Treeview widget for tabular data display. Each column in the Treeview is configured with headings and specific widths.

The GUI layout is user-friendly, with clear labels and button styles configured for better visual appeal. The buttons allow viewing of table records and also include a "Return To Main Menu" button for easy navigation. The code uses SQL queries to fetch data from the specified tables in the MySQL database and displays this data in an organized and interactive format, making it accessible even to those not familiar with SQL or command-line database operations.



The screenshot shows a Tkinter window titled "UserAlert Records". Inside the window, there is a table with four columns: AlertID, UserID, AlertMessage, and Alert\_Info\_AlertID. The table contains 12 rows of data. The first 10 rows show alerts for users 1 through 10, with messages like "Alert Message 1" through "Alert Message 10". The last two rows (11 and 12) show tasks for user 1, with messages "Task 1 is due in 0 days" and "Task 2 is due in 1 days".

AlertID	UserID	AlertMessage	Alert_Info_AlertID
1	1	Alert Message 1	1
2	2	Alert Message 2	2
3	3	Alert Message 3	3
4	4	Alert Message 4	4
5	5	Alert Message 5	5
6	6	Alert Message 6	6
7	7	Alert Message 7	7
8	8	Alert Message 8	8
9	9	Alert Message 9	9
10	10	Alert Message 10	10
11	1	Task 1 is due in 0 days	None
12	1	Task 2 is due in 1 days	None

 Record View

—

□

×

## Which Records would you like to view?

User

UserSettings

Calendar

Task

UserAlert

UserLogin

AlertInfo

Admin

Admin Login

Admin Settings

Return To Main Menu

```

1  import mysql.connector
2  from tkinter import Tk, Label, Button, Toplevel
3  from tkinter.ttk import Treeview
4  from tkinter import Scrollbar
5
6  def record_view(db):
7      # Function to display table data in a new window
8      def show_table_data(table_name, columns):
9          # Function to fetch and populate data into the treeview
10         def populate_data():
11             cursor = db.cursor()
12             cursor.execute(f"SELECT * FROM {table_name}") # SQL query to fetch all records from the specified table
13             records = cursor.fetchall() # Fetching all records from the table
14             for row in records:
15                 tree.insert('', 'end', values=row) # Inserting each record into the treeview
16             cursor.close() # Closing the cursor
17
18         # Creating a new window to display the data of a specific table
19         data_window = Toplevel()
20         data_window.title(f"{table_name} Records")
21         data_window.geometry("1000x600")
22
23         # Displaying a label at the top of the data window
24         Label(data_window, text=f"{table_name} Records", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))
25
26         # Setting up the treeview for displaying table data
27         tree = Treeview(data_window, columns=columns, show='headings')
28         for col in columns:
29             tree.heading(col, text=col) # Configuring column headers
30             tree.column(col, width=100, anchor='center') # Setting column width and alignment
31
32         tree.pack(expand=True, fill='both', padx=10, pady=10) # Placing the treeview in the window
33
34         # Adding a scrollbar to the treeview
35         scrollbar = Scrollbar(data_window, command=tree.yview)
36         tree.configure(yscrollcommand=scrollbar.set) # Linking the scrollbar to the treeview
37         scrollbar.pack(side='right', fill='y') # Placing the scrollbar
38
39         populate_data() # Populating the treeview with data
40
41     # Main window for selecting which table records to view
42     record_window = Tk()
43     record_window.title("Record View")
44     record_window.geometry("600x600")
45     record_window.configure(bg='white')

```

```

45     record_window.configure(bg='white')
46
47     # Main label for the record window
48     Label(record_window, text="Which Records would you like to view?", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))
49
50     # Button style configuration
51     button_style = {'fg': "black", 'bg': "#ADD8E6", 'font': ("Helvetica", 10, "bold"), 'highlightthickness': 4, 'width': 20, 'height': 1}
52
53     # Dictionary containing table names and their respective columns
54     tables = {
55         'User': ['UserID', 'Username', 'Password', 'Name', 'Address', 'PhoneNumber', 'Birthday'],
56         'UserSettings': ['SettingsID', 'UserID', 'NotificationSettings'],
57         'Calendar': ['CalendarID', 'UserID'],
58         'Task': ['TaskID', 'UserID', 'Title', 'Time', 'Duration', 'Description', 'Calendar_CalendarID'],
59         'UserAlert': ['AlertID', 'UserID', 'AlertMessage', 'Alert_Info_AlertID'],
60         'UserLogin': ['LoginID', 'UserID', 'LoginTime', 'LogoutTime'],
61         'AlertInfo': ['InfoID', 'AlertID', 'UserID', 'AlertHistory'],
62         'Admin': ['AdminID', 'Username', 'Password', 'Name', 'Certification', 'Address', 'PhoneNumber', 'Birthday'],
63         'Admin Login': ['LoginID', 'AdminID', 'LoginTime', 'LogoutTime'],
64         'Admin Settings': ['SettingsID', 'AdminID', 'NotificationSettings']
65     }
66
67     for i, (table_name, columns) in enumerate(tables.items()):
68         Button(record_window, text=table_name, **button_style, command=lambda name=table_name, cols=columns: show_table_data(name, cols)).pack(pady=5)
69
70     Button(record_window, text="Return To Main Menu", **button_style, command=record_window.destroy).pack(side='left', padx=10, pady=10)
71
72     record_window.mainloop()

```

## References:

1. “A To-Do List to Organize Your Work & Life.” *Todoist*, [todoist.com/](https://todoist.com/). Accessed 30 Sep. 2023.
2. Atlassian. “Jira: Issue & Project Tracking Software.” *Atlassian*, [www.atlassian.com/software/jira](https://www.atlassian.com/software/jira). Accessed 30 Sep. 2023.
3. “Trello Brings All Your Tasks, Teammates, and Tools Together.” *Trello*, [trello.com/home](https://trello.com/home). Accessed 30 Sep. 2023.