Contact Management System

Database Management

CMPT308N - 113

Fourth Floor

Marist College
School of Computer Science and Mathematics

Submitted To: Dr. Reza Sadeghi

11 / 29 / 2023

Progress Report of Contact Management System

Fourth Floor

Contact Management System

Team Members

1. Dylan Laewe	dylan.laewe1@marist.edu (Team Head)
2. Joseph Fortunato	joseph.fortunato1@marist.edu (Team Member)

Description of Team Members

1. Dylan Laewe:

   My name is Dylan Laewe, and I am an aspiring software developer who has a deep interest in technology. Throughout this past year getting familiar with programming, my curiosity has heightened, and I am excited to work on this project. As for the selection of my team, it made sense to team with someone I got to know while living in the same dorm last year. Joe and I are extremely versatile in our skill sets and I believe that we will be able to work together in a fluid and efficient manner. We selected myself as the team head because of my organizational skills.

2. Joseph Fortunato:

   My name is Joseph Fortunato, as a programmer I am inherently goal oriented and driven. I thrive on setting goal objectives and pursuing them with unwavering determination. Whether it’s crafting efficient algorithms, designing elegant software solutions, or troubleshooting complex issues within a program. I approach each task with a focused mindset. My dedication to a continuous learning experience at Marist college fuels my passion for coding and problem solving. I view challenges as opportunities for growth and a chance to solve intricate problems head on. My commitment to deliver high quality code and achieving project milestones reflects my strong work ethic and dedication to the field of computer science. I am driven to make a significant impact in the world of technology.

Table of Contents

Related Work Page

HubSpot:

HubSpot is one of the largest and widely used contact management systems in the world. It is held in high regard, and it has great reviews.

Positive Aspects:

HubSpot offers integration with applications such as Gmail and outlook in addition to providing an option to track contact activity and emails.

Negative Aspects:

HubSpot does not support any languages other than English, which is a major flaw.

Source 1: **https://www.zendesk.com/sell/crm/contact-management-software/**

Streak:

Streak is a contact management system which is used for small to medium sized businesses.

Positive Aspects:

This application has a plethora of features such as social media integration, and the ability to work with Google Workspace applications. Streak also offers a mobile app for its users which adds convenience to its already impressive features.

Negative Aspects:

The major flaw of this CMS is that since it is primarily built for Gmail users. As such, if you use platforms like Outlook, then you are out of luck.

Source 2: **https://www.getapp.com/customer-management-software/a/pipedrive/reviews/**

Pipedrive:

Pipedrive is a popular CMS that is primarily used to help organize sales teams.

Positive Aspects:

Pipedrive offers features that could be extremely beneficial to sales teams. Some of these attributes include but are not limited to the AI-powered virtual assistant in addition to an integrated calendar system.

Negative Aspects:

Reportedly, the site can be quite confusing in addition to the customer service being somewhat lackluster.

Source3 : https://www.getapp.com/customer-management-software/a/pipedrive/reviews/

Merits of our CMS

Our contact management system offers a robust and user-friendly solution for individuals and organizations seeking efficient contact management and collaboration. With the ability to add and edit contacts seamlessly, users can ensure that their contact information remains accurate and up to date, reducing the risk of communication errors. The powerful search and sorting features enable quick and easy access to specific contacts, while the ability to log interactions and add notes ensures that users can keep a detailed history of their engagements. One standout feature of our system is its commitment to security and access control which includes password and username protection. Users can rest assured that their data is protected, and they can only access information pertaining to their roles and permissions. Our systems features make it an asset for anyone seeking a dependable contact management solution.

End User Capabilities:

1. **Adding Contacts** – User will be able to add new contacts to the system, including their names, contact information (phone numbers, email addresses, physical addresses).
2. **Editing Contact Information** – Users can update and edit existing contact details, ensuring that the information is accurate and up to date.
3. **Searching and Sorting** - Users can search for specific contacts using various criteria (name, company, category) and sort contacts alphabetically or by other attributes.
4. **Contact History and Notes** - Users can keep track of interactions with contacts by adding notes, logging calls, meetings, and other activities related to each contact.
5. **Security and Access Control** – Our contact management systems will implement access control measures to ensure that users can only access and edit contacts and information

relevant to their roles and permissions such as implementing passwords and usernames that can only access data pertaining to that associated person.

6. **Reminders and Notifications** - The system will provide features for setting reminders and receiving notifications about important events or follow-ups related to contacts. This is good for the organization and helps them stay on schedule (list all scheduled works for one day).
7. **Duplicate Protection** – The system will keep records for data that has already been imported. If a user tries to unintentionally duplicate the input, a warning will pop up.
8. **Exit/ Welcome Functions** – The users will be greeted with a user-friendly welcome page as well as an exit function followed by a Thank You! for using the software.

GitHub Repository Address:

https://github.com/StunnaboyD/CMPT308N-113_ContactManagementSystem_FourthFloor

Entity Relationship Model and Diagram

<u>Entities:</u>

Events

<u>Attributes:</u>

1. EventID (Primary Key)
2. Event Name
3. Date
4. Location
5. Description

Search History

<u>Attributes:</u>

1. SearchID (Primary Key)
2. UserID (Foreign Key)
3. SearchQuery
4. SearchDate
5. SearchResults

Event Attendance:

<u>Attributes:</u>

1. AttendenceID (Primary Key)
2. UserID
3. EventID
4. Attendence_status
5. Attendence_Time

Contacts

<u>Attributes:</u>

1. ContactID (Primary Key)
2. Fname
3. Lname
4. Email (Multivalued Attribute)

5. Phone_Num

User

Attributes:

1. UserID (Primary Key)
2. Username
3. Password
4. Fname
5. Lname
6. Email
7. Phone_Num
8. Address (Composite Attribute)
9. GroupID (Foreign Key)

Groups

Attributes:

1. GroupID (Primary Key)
2. GroupName
3. GroupDescription
4. GroupNum
5. GroupType

Notifications/ Task

Attributes:

1. TaskID (Primary Key)
2. Title
3. Due Date
4. Priority
5. Status
6. Assigned_To (Contact)

Interaction

Attributes:

1. InteractionID (Primary Key)
2. ContactID (Foreign Key)
3. EventID (Foreign Key)
4. Interaction_Date
5. Notes

Messages

Attributes:

1. MessageID (Primary Key)
2. Sender (User or Contact)
3. Recipient (User or Contact)
4. Subject
5. Content
6. Timestamp

Notes

Attributes:

1. NoteID (Primary Key)
2. Title
3. Content
4. Created_Date
5. Last_Modified_Date

Partial Participation:

Interaction: Partial participation with Contact (1-N) and Event (1-N). This means not all interactions require participation from both a **contact** and an **event**.

Total Participation:

Notifications/Task is an example of total participation with **User** (1-N). This means every **Task** is assigned to at least one **User**.

Multivalued Attribute:

Contact has a multivalued attribute named emails. This is a multivalued attribute because an email can have multiple email addresses.

Composite Attribute:

The attribute in **User** has a composite attribute named address. This is an example of a composite attribute because it has various sub attributes such as street, city, state, and postal code.

Derived Attribute:

In the entity **User** the attribute named Full_Name is a derived attribute because it's a combination of Fname and Lname. This simplifies querying and eliminates the use of concatenating Fname and Lname.

1-1 Cardinality Ratio:

There is a 1-1 Cardinality ratio between **User** and **Group**. They share a 1-1 cardinality ratio because each **user** belongs to exactly one **group**, and each **group** can have only one **user**.

1-N Cardinality Ratio:

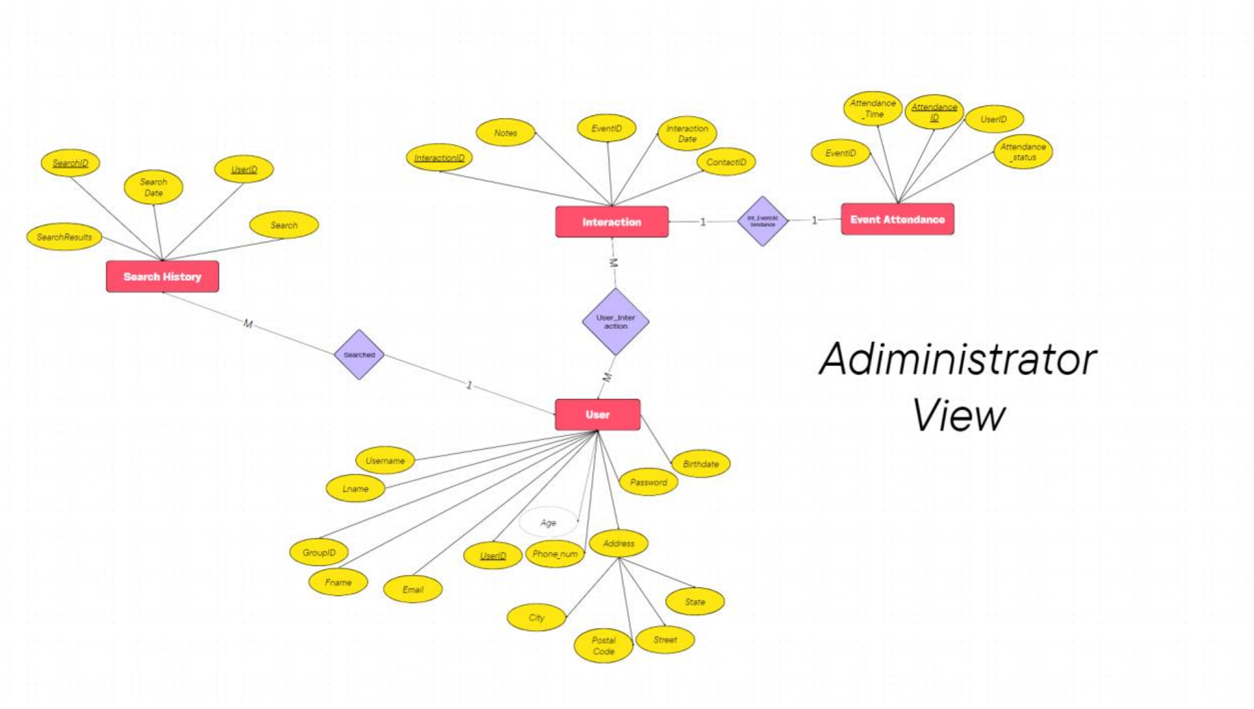
Notifications/Task because Task (1) is assigned to **User** (N), representing a 1-N cardinality ratio.

M-N Cardinality Ratio:

Each record in the **EventAttendance** entity represents a **user's** attendance status for a specific **event**.

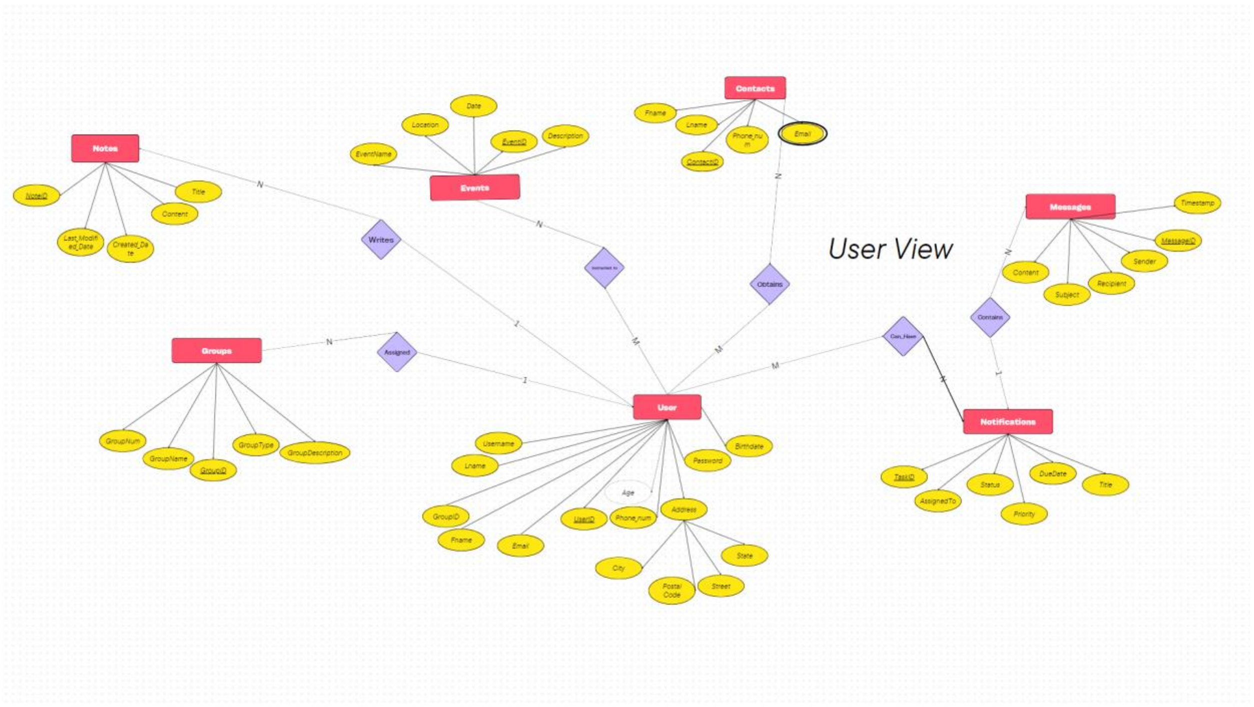
Administrator’s ER Model:

Description: The administrator’s view is basically having a view on workers search history, Event Attendance, and interactions they have. These tables are good to have from the administrator’s view because they can track what the user does and if they attend events or not. This view is more so for the record of the employee so administrator can track what each user, otherwise known as employee, does with the system.



User’s ER Model:

Description: The way that we came up with two external models is we asked ourselves what would make sense with our database. We came up with two external models, one named Managers View and the other User View. The manager’s view is basically all the stuff that would be necessary for a manager to see such as search history, Event attendance and interactions. The Users view holds all the necessary tables that it would need for an end users view on the contact management system.



Relationships

__Events:__

Event Attendance: (Many-to-Many relationship). An **event** can have multiple **attendees** (users), and a user can attend multiple events.

Interaction: (Many-to-Many relationship). **Events** can be associated with multiple **interactions**, and interactions can involve multiple events.

__Search History:__

User: (Many-to-One relationship). Each **search history** entry is associated with one **user** who performed the search.

__Event Attendance__:

User: (Cardinality M-N relationship). Each **attendance** record is associated with one **user**.

__User:__

Group: (1-N relationships). Each **user** belongs to one **group**.

Notifications/Task: (Cardinality 1-N relationship). **Users** can have multiple **tasks** assigned to them.

__Groups:__

User: (One-to-Many relationship). Each group can have multiple users but only one user can be assigned to a group.

__Notifications/Task:__

Contact: (Many-to-One relationship). **Tasks** can be assigned to one **contact**.

__Interaction:__

Contact: (Many-to-Many relationship). Each **interaction** is associated with one **contact**.

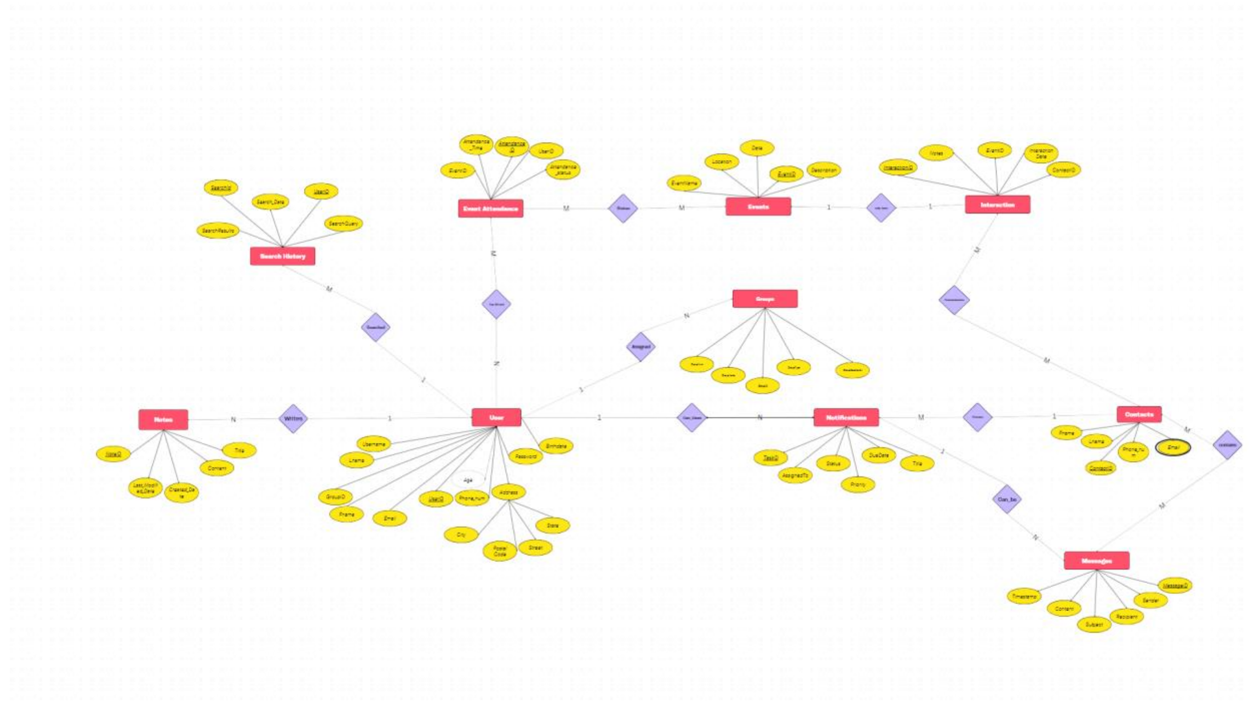
Event: (1-1 relationship). Each **interaction** is associated with one **event**.

Messages:

Contact (Sender and Recipient): (Many-to-Many relationship. **Messages** can be sent by and received by multiple **Contacts.**

Notes:

User: (1-N relationship). Every **Note** must be assigned to a **User** but not every **User** must have a **note**.



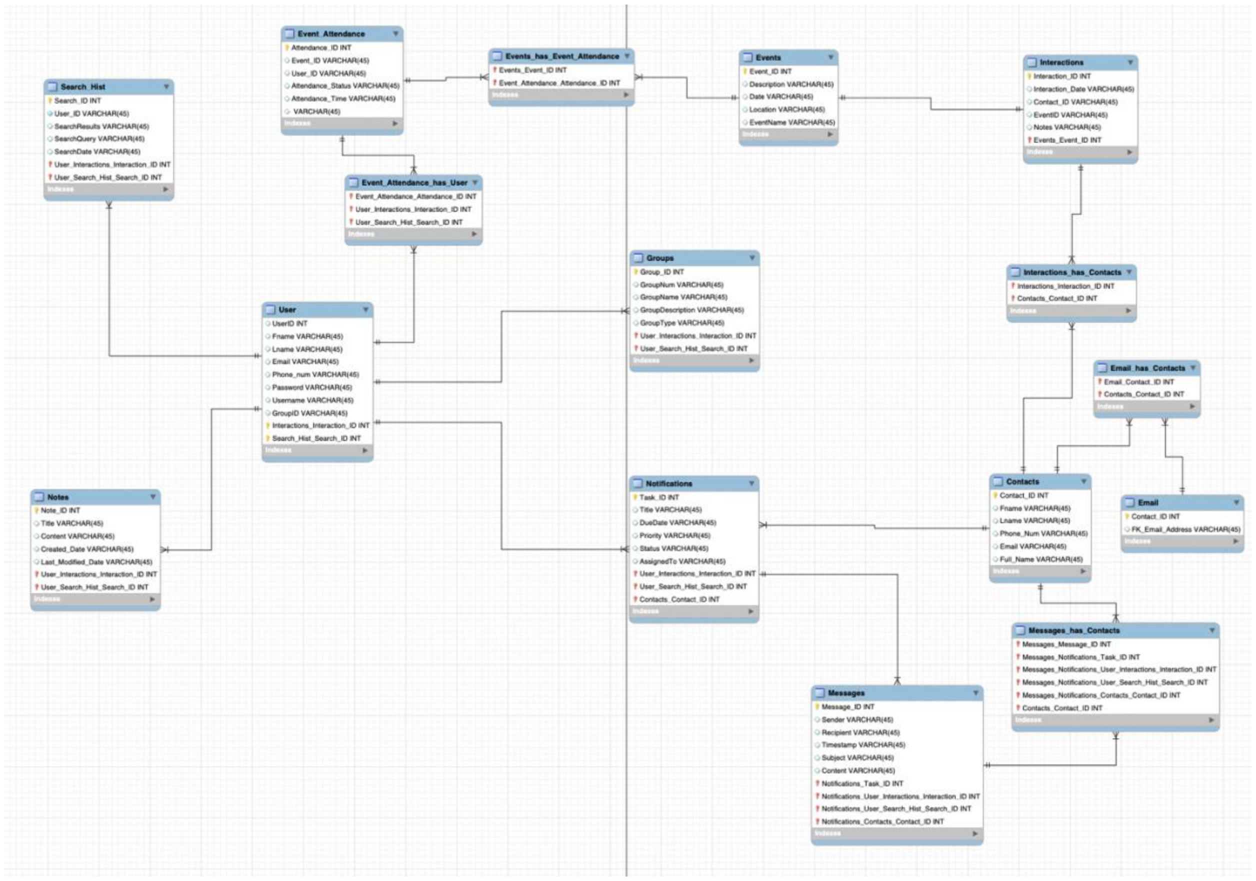
Enhanced Entity Relationship Model:

In an EER diagram, keys and relationships are used to define the structure of the database. A key is a set of attributes that uniquely identifies an entity in a table. Relationships are used to connect entities in different tables.

For the contact management system with the entities User, Contacts, Notifications, Events, EventAttendance, Interactions, Groups, SearchHistory, Messages, and Notes, we can define the following keys and relationships:

- User: The primary key for the User entity is the user ID. This key can be used to link the User entity to other entities in the database. We chose to use the INT data type to store the value for the user ID attribute.
- Contacts: The primary key for the Contacts entity is the contact ID. This key can be used to link the Contacts entity to other entities in the database. We used an INT data type to store the values for this key.
- Notifications: The primary key for the Notifications entity is the notification ID. This key can be used to link the Notifications entity to other entities in the database. The INT data type was used to store the values for this primary key.
- Events: The primary key for the Events entity is the event ID. This key can be used to link the Events entity to other entities in the database. This primary key utilized the INT data type for value storage.
- EventAttendance: The primary key for the EventAttendance entity is a combination of the user ID and the event ID. This key can be used to link the EventAttendance entity to the User and Events entities. Values for this primary key were stored using the INT data type.
- Interactions: The primary key for the Interactions entity is the interaction ID. This key can be used to link the Interactions entity to other entities in the database. The storage of values for this primary key was achieved through the use of the INT data type.
- Groups: The primary key for the Groups entity is the group ID. This key can be used to link the Groups entity to other entities in the database. Employing the INT data type, the values for this primary key were stored.
- SearchHistory: The primary key for the SearchHistory entity is the search ID. This key can be used to link the SearchHistory entity to other entities in the database. The INT data type was leveraged for storing the values of this primary key.
- Messages: The primary key for the Messages entity is the message ID. This key can be used to link the Messages entity to other entities in the database. Storage of values for this primary key was accomplished by implementing the INT data type.
- Notes: The primary key for the Notes entity is the note ID. This key can be used to link the Notes entity to other entities in the database. The use of the INT data type facilitated the storage of values for this primary key.

Enhanced Entity Relationship:



Database Development:

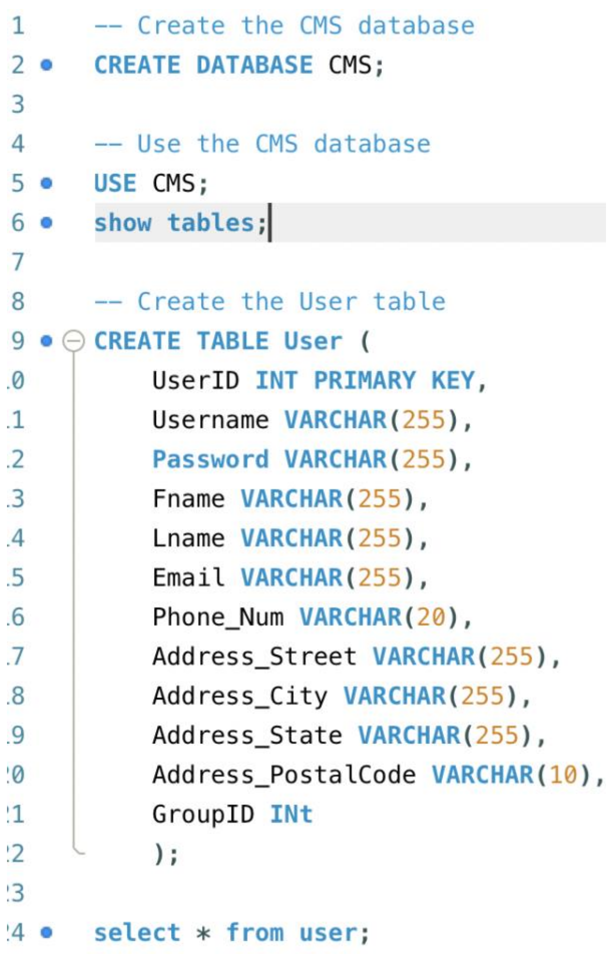
User Table:

Attributes: UserID (Primary Key), Username, Password, Fname, Lname, Email, Phone_Num, Address_Street, Address_City, Address_State, Address_PostalCode, GroupID

Data Types: INT, VARCHAR

Usage: Stores information about users.

Relationships: N/A



```sql
-- Create the CMS database
CREATE DATABASE CMS;

-- Use the CMS database
USE CMS;
show tables;

-- Create the User table
CREATE TABLE User (
    UserID INT PRIMARY KEY,
    Username VARCHAR(255),
    Password VARCHAR(255),
    Fname VARCHAR(255),
    Lname VARCHAR(255),
    Email VARCHAR(255),
    Phone_Num VARCHAR(20),
    Address_Street VARCHAR(255),
    Address_City VARCHAR(255),
    Address_State VARCHAR(255),
    Address_PostalCode VARCHAR(10),
    GroupID INt
    );

select * from user;
```

Groupss Table:

Attributes: GroupID (Primary Key), GroupName, GroupDescription, GroupNum, GroupType

Data Types: INT, VARCHAR

Usage: Stores information about different groups.

Relationships: N/A

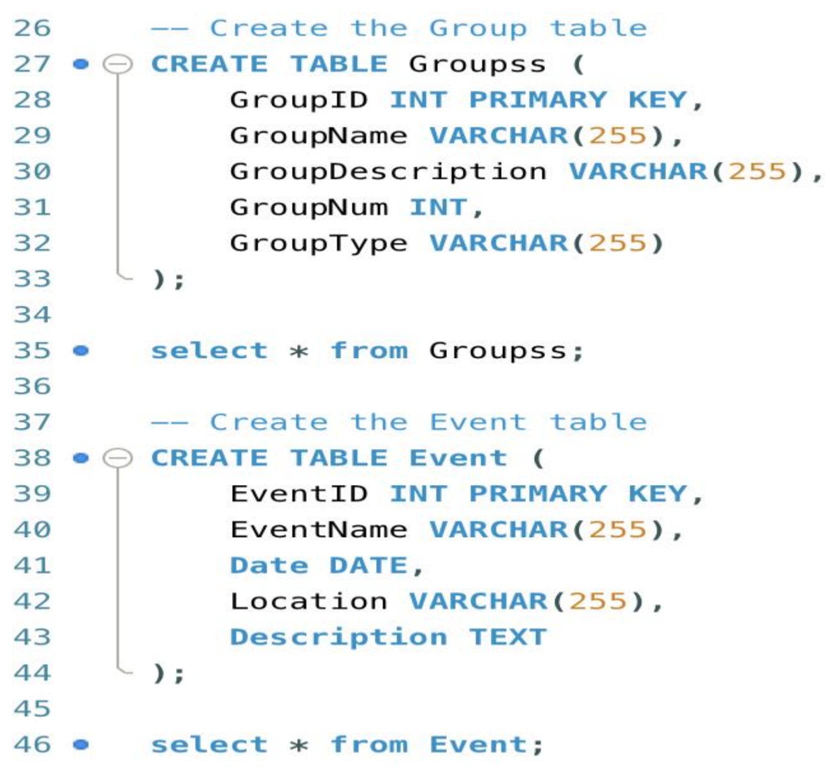
Event Table:

Attributes: EventID (Primary Key), EventName, Date, Location, Description

Data Types: INT, VARCHAR, DATE, TEXT

Usage: Stores information about different events.

Relationships: N/A



```sql
-- Create the Group table
CREATE TABLE Groupss (
    GroupID INT PRIMARY KEY,
    GroupName VARCHAR(255),
    GroupDescription VARCHAR(255),
    GroupNum INT,
    GroupType VARCHAR(255)
);

select * from Groupss;

-- Create the Event table
CREATE TABLE Event (
    EventID INT PRIMARY KEY,
    EventName VARCHAR(255),
    Date DATE,
    Location VARCHAR(255),
    Description TEXT
);

select * from Event;
```

Event Attendance Table:

Attributes: AttendanceID (Primary Key), UserID, EventID, AttendanceStatus, AttendanceTime

Data Types: INT, VARCHAR, DATETIME

Usage: Tracks the attendance of users at different events.

Relationships: Contains foreign key references to User and Event tables.

Contact Table:

Attributes: ContactID (Primary Key), Fname, Lname, Phone_Num

Data Types: INT, VARCHAR

Usage: Stores information about different contacts.

Relationships: N/A

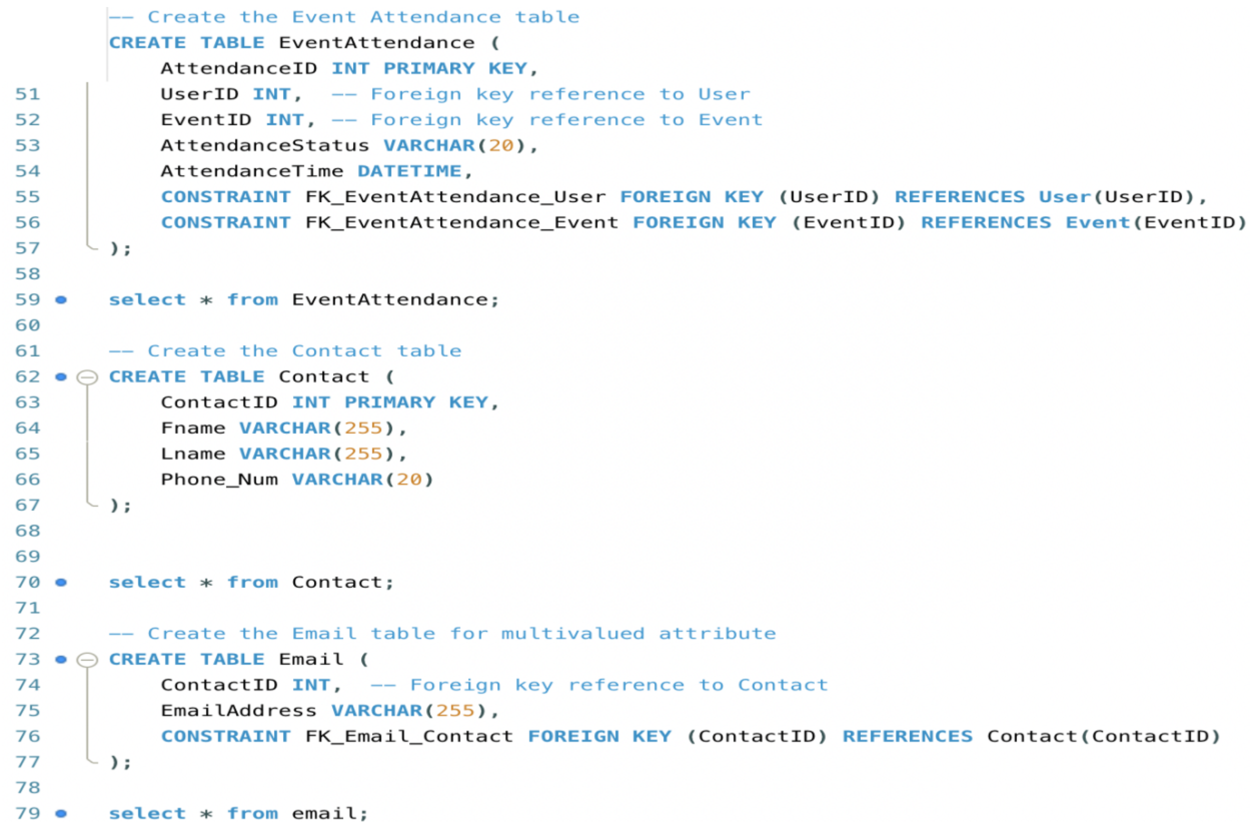
Email Table:

Attributes: ContactID, EmailAddress

Data Types: INT, VARCHAR

Usage: Stores email addresses related to specific contacts.

Relationships: Contains a foreign key reference to the Contact table.



```sql
-- Create the Event Attendance table
CREATE TABLE EventAttendance (
    AttendanceID INT PRIMARY KEY,
    UserID INT,  -- Foreign key reference to User
    EventID INT, -- Foreign key reference to Event
    AttendanceStatus VARCHAR(20),
    AttendanceTime DATETIME,
    CONSTRAINT FK_EventAttendance_User FOREIGN KEY (UserID) REFERENCES User(UserID),
    CONSTRAINT FK_EventAttendance_Event FOREIGN KEY (EventID) REFERENCES Event(EventID)
);

select * from EventAttendance;

-- Create the Contact table
CREATE TABLE Contact (
    ContactID INT PRIMARY KEY,
    Fname VARCHAR(255),
    Lname VARCHAR(255),
    Phone_Num VARCHAR(20)
);


select * from Contact;

-- Create the Email table for multivalued attribute
CREATE TABLE Email (
    ContactID INT,  -- Foreign key reference to Contact
    EmailAddress VARCHAR(255),
    CONSTRAINT FK_Email_Contact FOREIGN KEY (ContactID) REFERENCES Contact(ContactID)
);

select * from email;
```

Notifications Task Table:

Attributes: TaskID (Primary Key), Title, DueDate, Priority, Status, AssignedTo

Data Types: INT, VARCHAR, DATE

Usage: Stores information about notifications and tasks assigned to users.

Relationships: Contains a foreign key reference to the User table.

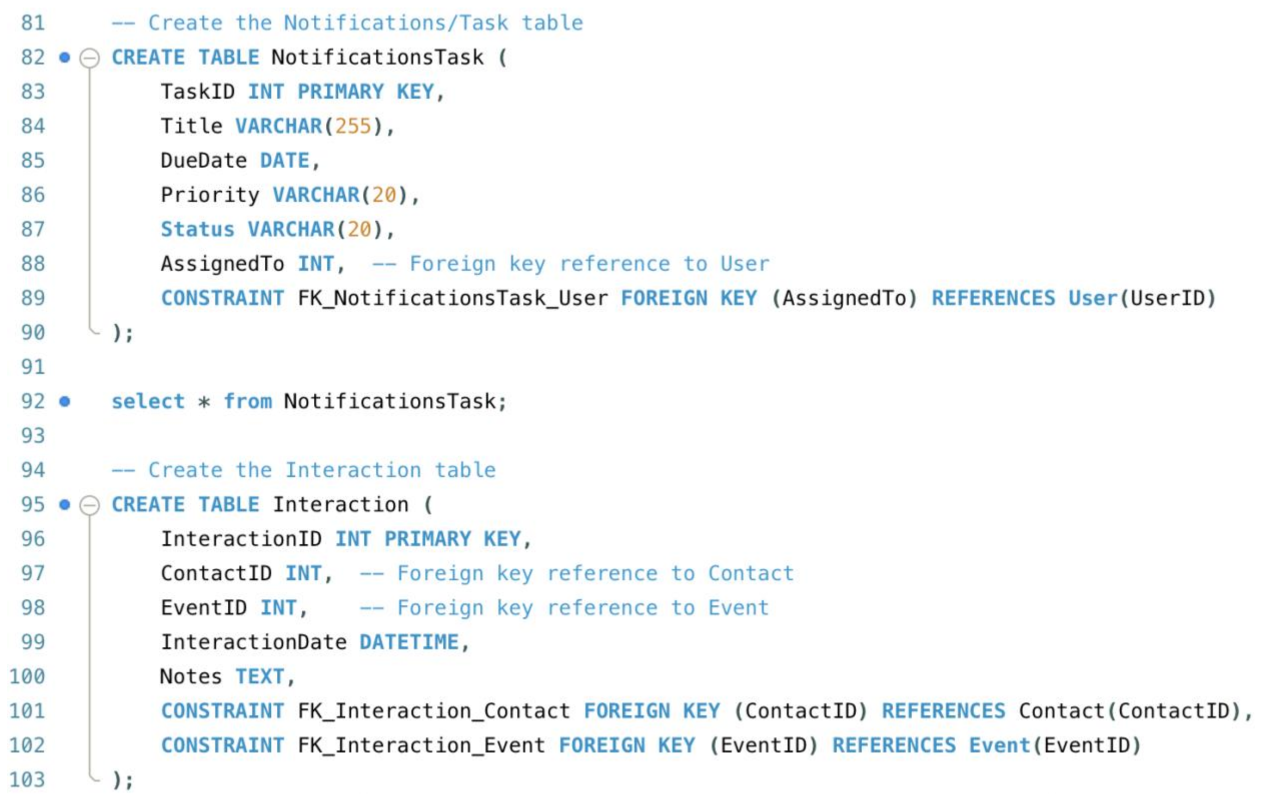
Interaction Table:

Attributes: InteractionID (Primary Key), ContactID, EventID, InteractionDate, Notes

Data Types: INT, DATETIME, TEXT

Usage: Tracks interactions between contacts and events.

Relationships: Contains foreign key references to the Contact and Event tables.



```sql
-- Create the Notifications/Task table
CREATE TABLE NotificationsTask (
    TaskID INT PRIMARY KEY,
    Title VARCHAR(255),
    DueDate DATE,
    Priority VARCHAR(20),
    Status VARCHAR(20),
    AssignedTo INT,  -- Foreign key reference to User
    CONSTRAINT FK_NotificationsTask_User FOREIGN KEY (AssignedTo) REFERENCES User(UserID)
);

select * from NotificationsTask;

-- Create the Interaction table
CREATE TABLE Interaction (
    InteractionID INT PRIMARY KEY,
    ContactID INT,  -- Foreign key reference to Contact
    EventID INT,    -- Foreign key reference to Event
    InteractionDate DATETIME,
    Notes TEXT,
    CONSTRAINT FK_Interaction_Contact FOREIGN KEY (ContactID) REFERENCES Contact(ContactID),
    CONSTRAINT FK_Interaction_Event FOREIGN KEY (EventID) REFERENCES Event(EventID)
);
```

Messages Table:

Attributes: MessageID (Primary Key), SenderID, RecipientID, Subject, Content, Timestamp

Data Types: INT, VARCHAR, DATETIME, TEXT

Usage: Stores information about messages sent between users.

Relationships: Contains foreign key references to the User table.

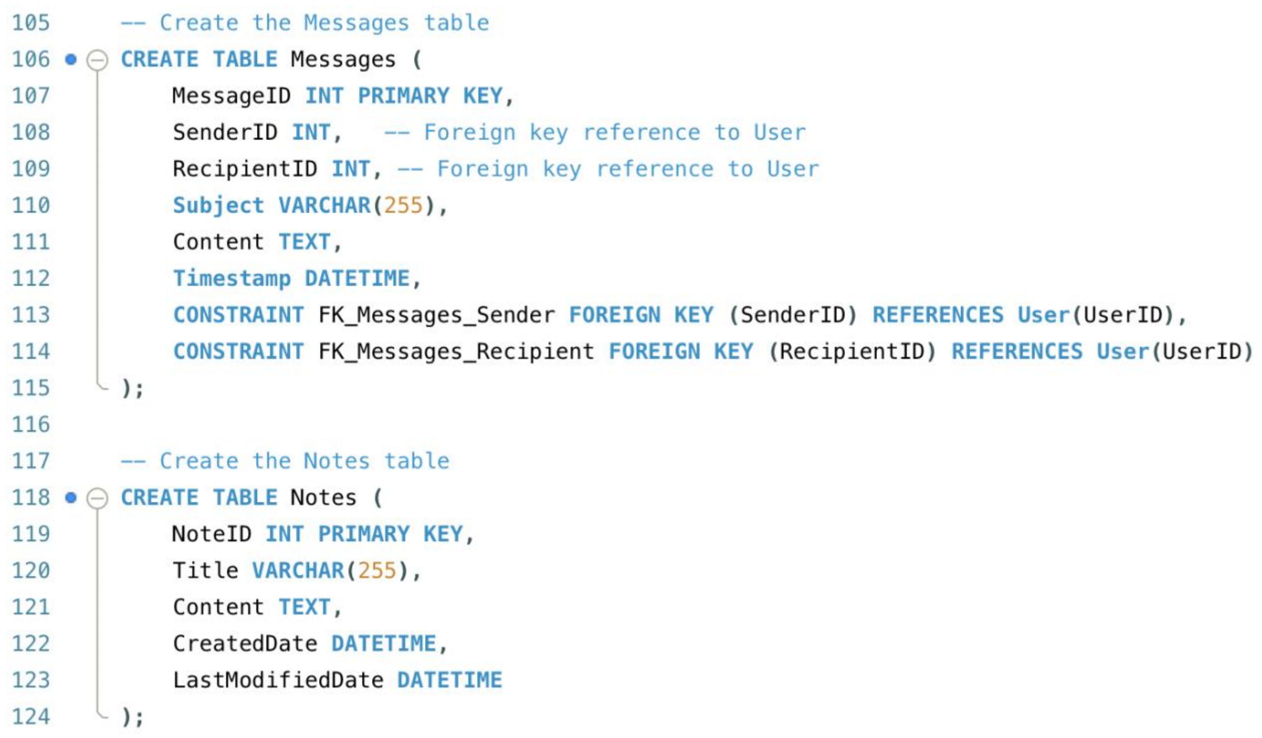
Notes Table:

Attributes: NoteID (Primary Key), Title, Content, CreatedDate, LastModifiedDate

Data Types: INT, VARCHAR, DATETIME, TEXT

Usage: Stores notes along with relevant information.

Relationships: N/A

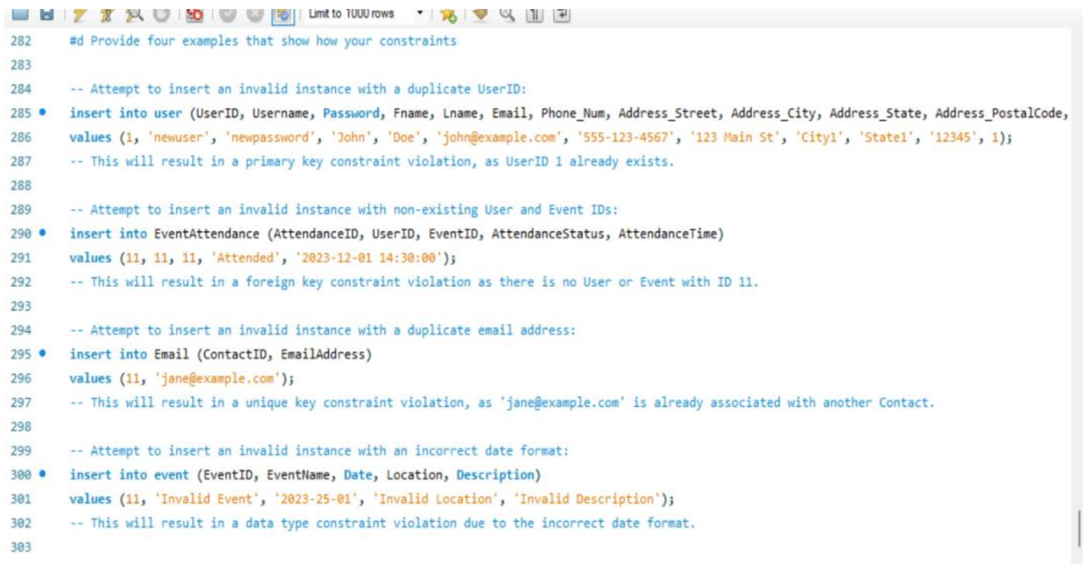


```sql
-- Create the Messages table
CREATE TABLE Messages (
    MessageID INT PRIMARY KEY,
    SenderID INT,   -- Foreign key reference to User
    RecipientID INT, -- Foreign key reference to User
    Subject VARCHAR(255),
    Content TEXT,
    Timestamp DATETIME,
    CONSTRAINT FK_Messages_Sender FOREIGN KEY (SenderID) REFERENCES User(UserID),
    CONSTRAINT FK_Messages_Recipient FOREIGN KEY (RecipientID) REFERENCES User(UserID)
);

-- Create the Notes table
CREATE TABLE Notes (
    NoteID INT PRIMARY KEY,
    Title VARCHAR(255),
    Content TEXT,
    CreatedDate DATETIME,
    LastModifiedDate DATETIME
);
```

Handling foreign Key Constraints

Handling foreign key constraints when filling one table with data and then the other requires careful planning and execution. To ensure data integrity, start by populating the parent table, which contains the primary key referenced by the foreign key in the child table. This sequence is crucial, as the foreign key constraint in the child table relies on the existence of corresponding records in the parent table. Once the parent table is populated, you can proceed to fill the child table, ensuring that the foreign key values match those of the parent table. It's essential to maintain the referential integrity of your database by double-checking the data you insert to avoid violating foreign key constraints. By following this procedure, you can establish a strong relationship between the tables, maintain data consistency, and prevent potential issues in your database.

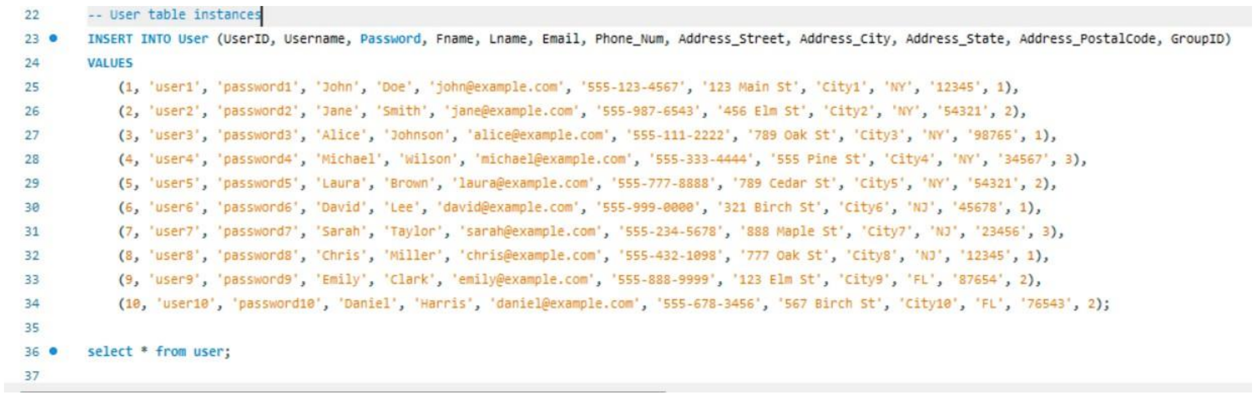
- For example, to avoid foreign key errors I filled the table of Event first and then filled the table of Event Attendance. By following through with this process, I avoided the foreign key errors and was able to import the data.



```sql
#d Provide four examples that show how your constraints

-- Attempt to insert an invalid instance with a duplicate UserID:
insert into user (UserID, Username, Password, Fname, Lname, Email, Phone_Num, Address_Street, Address_City, Address_State, Address_PostalCode,
values (1, 'newuser', 'newpassword', 'John', 'Doe', 'john@example.com', '555-123-4567', '123 Main St', 'City1', 'State1', '12345', 1);
-- This will result in a primary key constraint violation, as UserID 1 already exists.

-- Attempt to insert an invalid instance with non-existing User and Event IDs:
insert into EventAttendance (AttendanceID, UserID, EventID, AttendanceStatus, AttendanceTime)
values (11, 11, 11, 'Attended', '2023-12-01 14:30:00');
-- This will result in a foreign key constraint violation as there is no User or Event with ID 11.

-- Attempt to insert an invalid instance with a duplicate email address:
insert into Email (ContactID, EmailAddress)
values (11, 'jane@example.com');
-- This will result in a unique key constraint violation, as 'jane@example.com' is already associated with another Contact.

-- Attempt to insert an invalid instance with an incorrect date format:
insert into event (EventID, EventName, Date, Location, Description)
values (11, 'Invalid Event', '2023-25-01', 'Invalid Location', 'Invalid Description');
-- This will result in a data type constraint violation due to the incorrect date format.

```

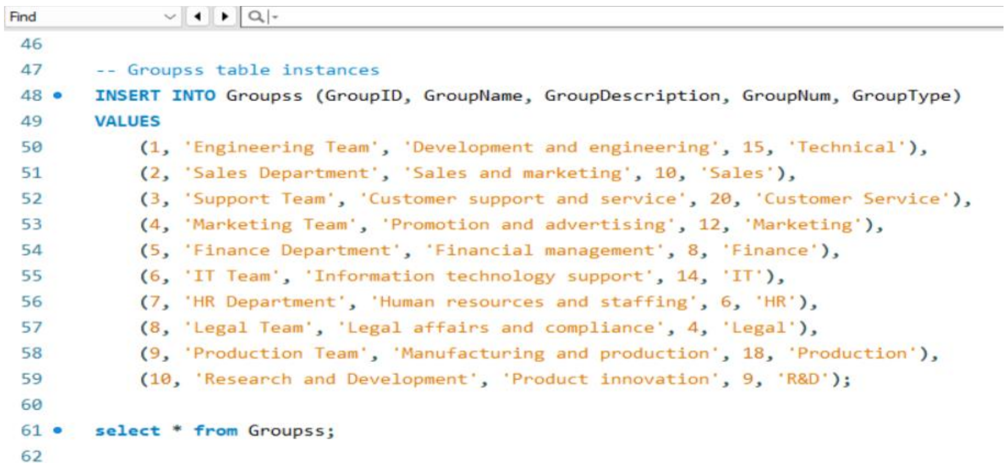
Importing Data

Description: For the Table of user, I imported the data for the following columns and imported the correct data value for the columns corresponding data type. The username is increased by 1 to create a unique username. The same technique was used for passwords so that each user has a different password than the other.



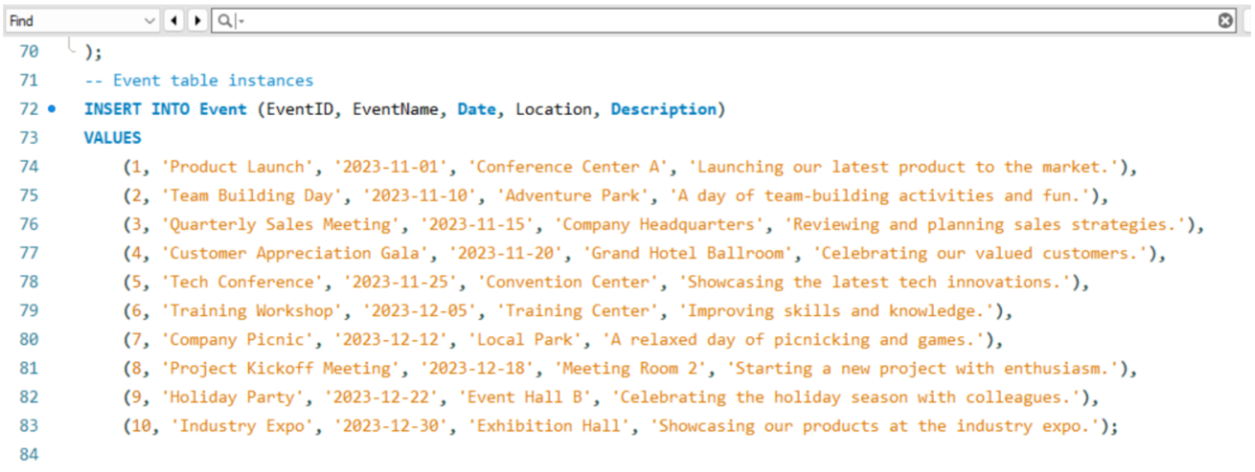
```sql
-- User table instances
INSERT INTO User (UserID, Username, Password, Fname, Lname, Email, Phone_Num, Address_Street, Address_City, Address_State, Address_PostalCode, GroupID)
VALUES
    (1, 'user1', 'password1', 'John', 'Doe', 'john@example.com', '555-123-4567', '123 Main St', 'City1', 'NY', '12345', 1),
    (2, 'user2', 'password2', 'Jane', 'Smith', 'jane@example.com', '555-987-6543', '456 Elm St', 'City2', 'NY', '54321', 2),
    (3, 'user3', 'password3', 'Alice', 'Johnson', 'alice@example.com', '555-111-2222', '789 Oak St', 'City3', 'NY', '98765', 1),
    (4, 'user4', 'password4', 'Michael', 'Wilson', 'michael@example.com', '555-333-4444', '555 Pine St', 'City4', 'NY', '34567', 3),
    (5, 'user5', 'password5', 'Laura', 'Brown', 'laura@example.com', '555-777-8888', '789 Cedar St', 'City5', 'NY', '54321', 2),
    (6, 'user6', 'password6', 'David', 'Lee', 'david@example.com', '555-999-0000', '321 Birch St', 'City6', 'NJ', '45678', 1),
    (7, 'user7', 'password7', 'Sarah', 'Taylor', 'sarah@example.com', '555-234-5678', '888 Maple St', 'City7', 'NJ', '23456', 3),
    (8, 'user8', 'password8', 'Chris', 'Miller', 'chris@example.com', '555-432-1098', '777 Oak St', 'City8', 'NJ', '12345', 1),
    (9, 'user9', 'password9', 'Emily', 'Clark', 'emily@example.com', '555-888-9999', '123 Elm St', 'City9', 'FL', '87654', 2),
    (10, 'user10', 'password10', 'Daniel', 'Harris', 'daniel@example.com', '555-678-3456', '567 Birch St', 'City10', 'FL', '76543', 2);

select * from user;
```

Description: For the table of Groupss I imported the instances for the corresponding data type. This time there are several teams with different GroupNums but the Group ID is a primary key in this table so it can help determine which users are apart of which team.



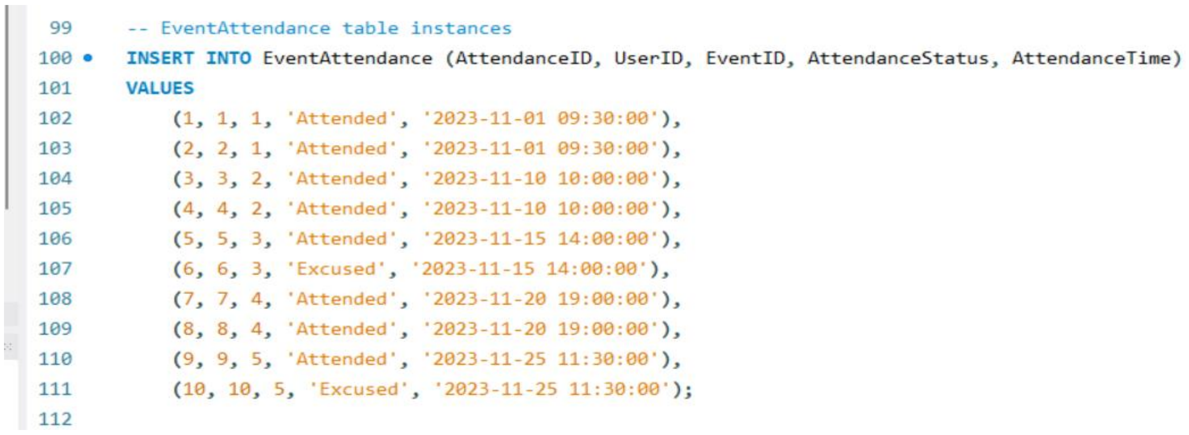
```sql
-- Groupss table instances
INSERT INTO Groupss (GroupID, GroupName, GroupDescription, GroupNum, GroupType)
VALUES
    (1, 'Engineering Team', 'Development and engineering', 15, 'Technical'),
    (2, 'Sales Department', 'Sales and marketing', 10, 'Sales'),
    (3, 'Support Team', 'Customer support and service', 20, 'Customer Service'),
    (4, 'Marketing Team', 'Promotion and advertising', 12, 'Marketing'),
    (5, 'Finance Department', 'Financial management', 8, 'Finance'),
    (6, 'IT Team', 'Information technology support', 14, 'IT'),
    (7, 'HR Department', 'Human resources and staffing', 6, 'HR'),
    (8, 'Legal Team', 'Legal affairs and compliance', 4, 'Legal'),
    (9, 'Production Team', 'Manufacturing and production', 18, 'Production'),
    (10, 'Research and Development', 'Product innovation', 9, 'R&D');

select * from Groupss;
```

Description: For the table of Event, I used the EventID for the primary key so it can distinctly identify which event is what and it gives it a unique identifier. Each event comes with a description of a general view of what is occurring in the event.



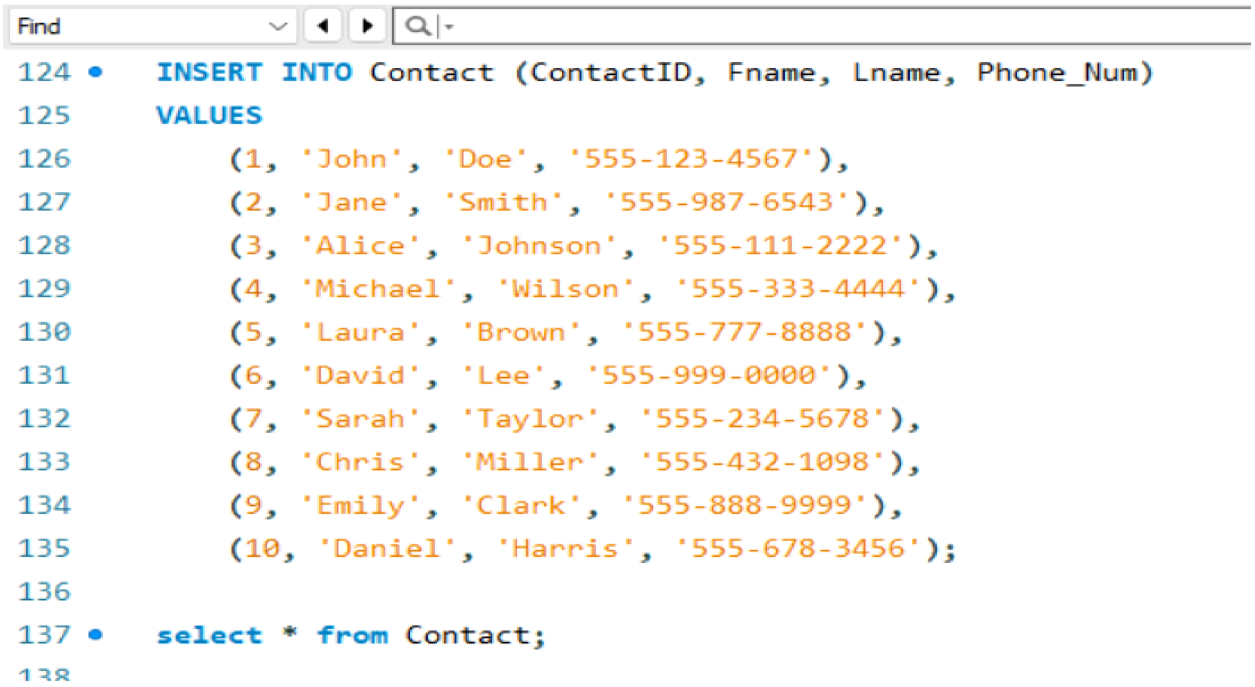
```sql
);
-- Event table instances
INSERT INTO Event (EventID, EventName, Date, Location, Description)
VALUES
    (1, 'Product Launch', '2023-11-01', 'Conference Center A', 'Launching our latest product to the market.'),
    (2, 'Team Building Day', '2023-11-10', 'Adventure Park', 'A day of team-building activities and fun.'),
    (3, 'Quarterly Sales Meeting', '2023-11-15', 'Company Headquarters', 'Reviewing and planning sales strategies.'),
    (4, 'Customer Appreciation Gala', '2023-11-20', 'Grand Hotel Ballroom', 'Celebrating our valued customers.'),
    (5, 'Tech Conference', '2023-11-25', 'Convention Center', 'Showcasing the latest tech innovations.'),
    (6, 'Training Workshop', '2023-12-05', 'Training Center', 'Improving skills and knowledge.'),
    (7, 'Company Picnic', '2023-12-12', 'Local Park', 'A relaxed day of picnicking and games.'),
    (8, 'Project Kickoff Meeting', '2023-12-18', 'Meeting Room 2', 'Starting a new project with enthusiasm.'),
    (9, 'Holiday Party', '2023-12-22', 'Event Hall B', 'Celebrating the holiday season with colleagues.'),
    (10, 'Industry Expo', '2023-12-30', 'Exhibition Hall', 'Showcasing our products at the industry expo.');
```

Description: For the table of EventAttendence this was made to track the status of which employee went to which event. This table has a foreign key of UserID to determine which employee is who and another foreign key of EventID from Event. This helps see what event they went to and attended or not.



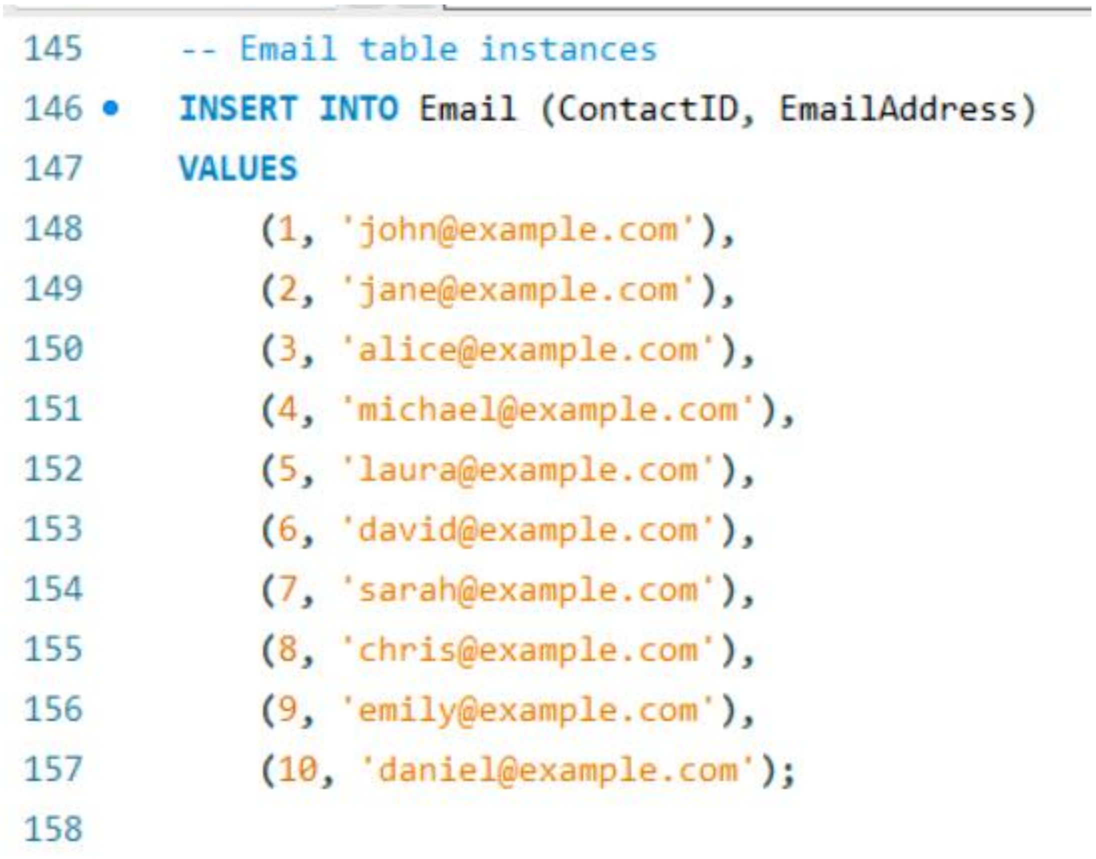
```sql
-- EventAttendance table instances
INSERT INTO EventAttendance (AttendanceID, UserID, EventID, AttendanceStatus, AttendanceTime)
VALUES
    (1, 1, 1, 'Attended', '2023-11-01 09:30:00'),
    (2, 2, 1, 'Attended', '2023-11-01 09:30:00'),
    (3, 3, 2, 'Attended', '2023-11-10 10:00:00'),
    (4, 4, 2, 'Attended', '2023-11-10 10:00:00'),
    (5, 5, 3, 'Attended', '2023-11-15 14:00:00'),
    (6, 6, 3, 'Excused', '2023-11-15 14:00:00'),
    (7, 7, 4, 'Attended', '2023-11-20 19:00:00'),
    (8, 8, 4, 'Attended', '2023-11-20 19:00:00'),
    (9, 9, 5, 'Attended', '2023-11-25 11:30:00'),
    (10, 10, 5, 'Excused', '2023-11-25 11:30:00');
```

Description: For the table of Contact I inserted all the provided data in the picture to the corresponding columns it needed to be entered. I followed the data types correctly, so all the information was validly put in. I made the primary key ContactID so that a user can uniquely be identified by the corresponding ID and name.



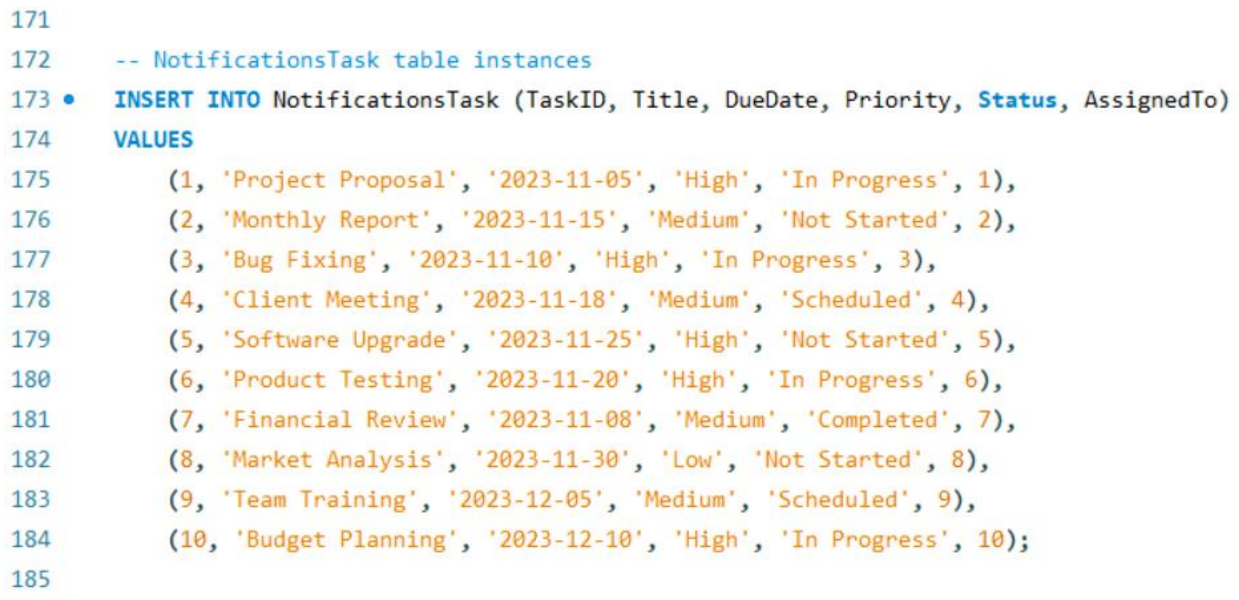
```
INSERT INTO Contact (ContactID, Fname, Lname, Phone_Num)
VALUES
    (1, 'John', 'Doe', '555-123-4567'),
    (2, 'Jane', 'Smith', '555-987-6543'),
    (3, 'Alice', 'Johnson', '555-111-2222'),
    (4, 'Michael', 'Wilson', '555-333-4444'),
    (5, 'Laura', 'Brown', '555-777-8888'),
    (6, 'David', 'Lee', '555-999-0000'),
    (7, 'Sarah', 'Taylor', '555-234-5678'),
    (8, 'Chris', 'Miller', '555-432-1098'),
    (9, 'Emily', 'Clark', '555-888-9999'),
    (10, 'Daniel', 'Harris', '555-678-3456');

select * from Contact;
```

Description: For the table of Email, I inserted the ContactID and the EmailAddress. This table had to be created from the Contact table because it's a multivalued attribute which can have multiple inputs. The Contact ID is the foreign key that was taken from the Contact table. That key helps to identify which person you want to contact.



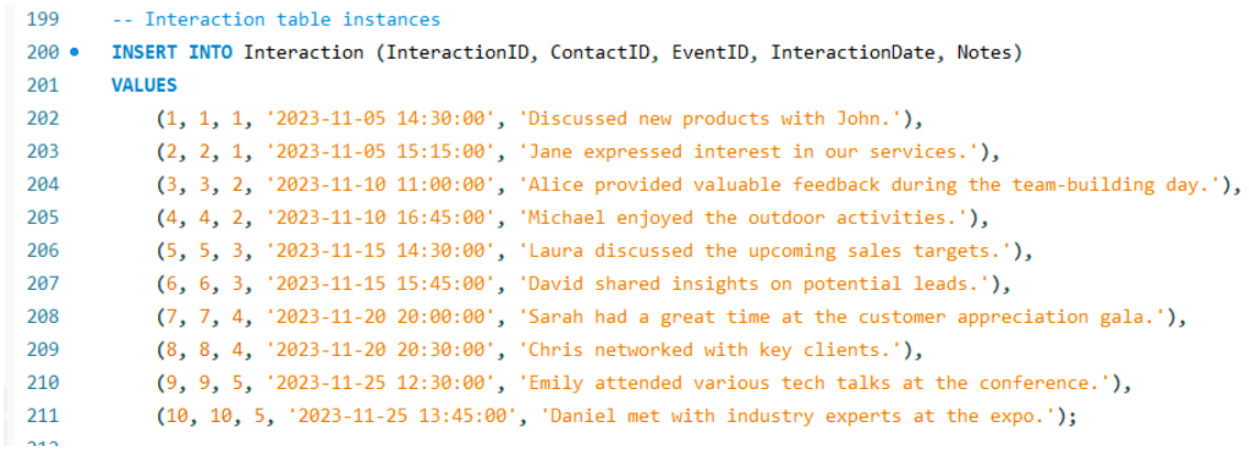
```sql
-- Email table instances
INSERT INTO Email (ContactID, EmailAddress)
VALUES
    (1, 'john@example.com'),
    (2, 'jane@example.com'),
    (3, 'alice@example.com'),
    (4, 'michael@example.com'),
    (5, 'laura@example.com'),
    (6, 'david@example.com'),
    (7, 'sarah@example.com'),
    (8, 'chris@example.com'),
    (9, 'emily@example.com'),
    (10, 'daniel@example.com');
```

Description: For the table of NotificationTask I inserted the corresponding data values that were needed and within the guidelines of the data types assigned. The primary key in this table is the taskID which is unique and is associated with different tasks that are assigned to different people. The foreign key in this table is the AssignedTo column. This key is a reference to UserID and uniquely identifies the person assigned to a task.



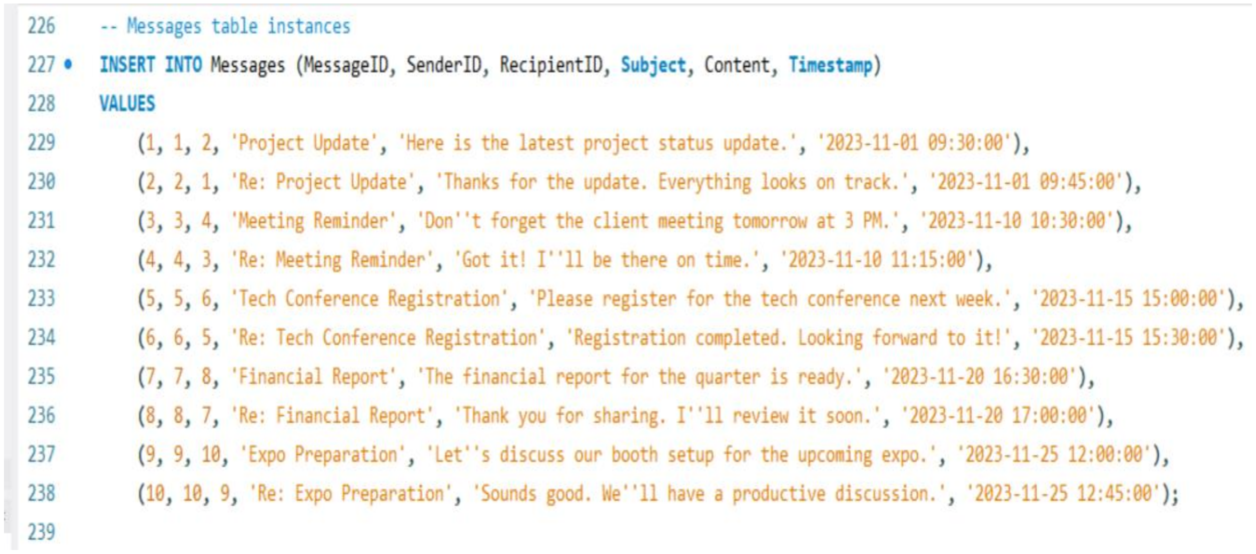
```sql

-- NotificationsTask table instances
INSERT INTO NotificationsTask (TaskID, Title, DueDate, Priority, Status, AssignedTo)
VALUES
    (1, 'Project Proposal', '2023-11-05', 'High', 'In Progress', 1),
    (2, 'Monthly Report', '2023-11-15', 'Medium', 'Not Started', 2),
    (3, 'Bug Fixing', '2023-11-10', 'High', 'In Progress', 3),
    (4, 'Client Meeting', '2023-11-18', 'Medium', 'Scheduled', 4),
    (5, 'Software Upgrade', '2023-11-25', 'High', 'Not Started', 5),
    (6, 'Product Testing', '2023-11-20', 'High', 'In Progress', 6),
    (7, 'Financial Review', '2023-11-08', 'Medium', 'Completed', 7),
    (8, 'Market Analysis', '2023-11-30', 'Low', 'Not Started', 8),
    (9, 'Team Training', '2023-12-05', 'Medium', 'Scheduled', 9),
    (10, 'Budget Planning', '2023-12-10', 'High', 'In Progress', 10);

```

Description: For the table of Interaction, I inserted the following instances to the corresponding columns. The data that was inserted was in guidelines of the data types so that it is a valid instance. In this table InteractionID is the primary key so that it can be uniquely identified. The Foreign keys in this table are ContactID and EventID. Since we want to track the interactions between contacts at certain events, we needed to input the corresponding keys to do so.



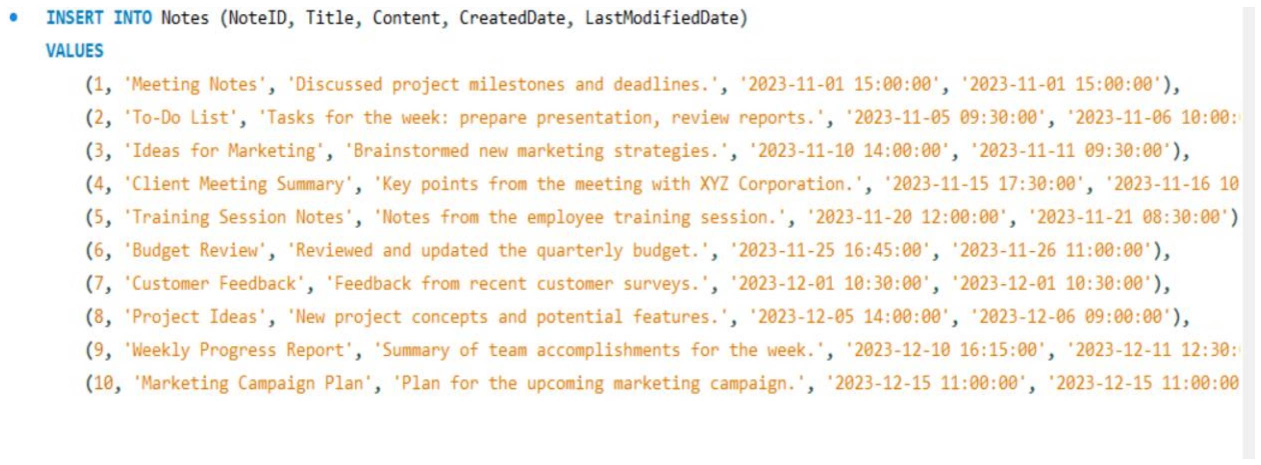
```sql
-- Interaction table instances
INSERT INTO Interaction (InteractionID, ContactID, EventID, InteractionDate, Notes)
VALUES
    (1, 1, 1, '2023-11-05 14:30:00', 'Discussed new products with John.'),
    (2, 2, 1, '2023-11-05 15:15:00', 'Jane expressed interest in our services.'),
    (3, 3, 2, '2023-11-10 11:00:00', 'Alice provided valuable feedback during the team-building day.'),
    (4, 4, 2, '2023-11-10 16:45:00', 'Michael enjoyed the outdoor activities.'),
    (5, 5, 3, '2023-11-15 14:30:00', 'Laura discussed the upcoming sales targets.'),
    (6, 6, 3, '2023-11-15 15:45:00', 'David shared insights on potential leads.'),
    (7, 7, 4, '2023-11-20 20:00:00', 'Sarah had a great time at the customer appreciation gala.'),
    (8, 8, 4, '2023-11-20 20:30:00', 'Chris networked with key clients.'),
    (9, 9, 5, '2023-11-25 12:30:00', 'Emily attended various tech talks at the conference.'),
    (10, 10, 5, '2023-11-25 13:45:00', 'Daniel met with industry experts at the expo.');
```

Description: For the table of messages, I inserted the data according to the corresponding data types that I assigned. This process helps keep the data valid when trying to query. In this table we have a primary key of MessageID which helps each message sent be unique. There are two foreign keys in this table which are SenderID and RecipientID. SenderID is a reference to UserID and ReciepientID is also a reference to UserID. This helps track who sends messages to who.



```sql
-- Messages table instances
INSERT INTO Messages (MessageID, SenderID, RecipientID, Subject, Content, Timestamp)
VALUES
    (1, 1, 2, 'Project Update', 'Here is the latest project status update.', '2023-11-01 09:30:00'),
    (2, 2, 1, 'Re: Project Update', 'Thanks for the update. Everything looks on track.', '2023-11-01 09:45:00'),
    (3, 3, 4, 'Meeting Reminder', 'Don''t forget the client meeting tomorrow at 3 PM.', '2023-11-10 10:30:00'),
    (4, 4, 3, 'Re: Meeting Reminder', 'Got it! I''ll be there on time.', '2023-11-10 11:15:00'),
    (5, 5, 6, 'Tech Conference Registration', 'Please register for the tech conference next week.', '2023-11-15 15:00:00'),
    (6, 6, 5, 'Re: Tech Conference Registration', 'Registration completed. Looking forward to it!', '2023-11-15 15:30:00'),
    (7, 7, 8, 'Financial Report', 'The financial report for the quarter is ready.', '2023-11-20 16:30:00'),
    (8, 8, 7, 'Re: Financial Report', 'Thank you for sharing. I''ll review it soon.', '2023-11-20 17:00:00'),
    (9, 9, 10, 'Expo Preparation', 'Let''s discuss our booth setup for the upcoming expo.', '2023-11-25 12:00:00'),
    (10, 10, 9, 'Re: Expo Preparation', 'Sounds good. We''ll have a productive discussion.', '2023-11-25 12:45:00');
```

Description: For the table Notes I inserted the following values to corresponding data types so that the information was kept valid. I created the NoteID column so that each note is distinctly unique and can be identified. The table shows when the note was created and the last time it was modified. The table also has the Content column so that the actual notes being taken can be created.



```
INSERT INTO Notes (NoteID, Title, Content, CreatedDate, LastModifiedDate)
VALUES
    (1, 'Meeting Notes', 'Discussed project milestones and deadlines.', '2023-11-01 15:00:00', '2023-11-01 15:00:00'),
    (2, 'To-Do List', 'Tasks for the week: prepare presentation, review reports.', '2023-11-05 09:30:00', '2023-11-06 10:00:
    (3, 'Ideas for Marketing', 'Brainstormed new marketing strategies.', '2023-11-10 14:00:00', '2023-11-11 09:30:00'),
    (4, 'Client Meeting Summary', 'Key points from the meeting with XYZ Corporation.', '2023-11-15 17:30:00', '2023-11-16 10
    (5, 'Training Session Notes', 'Notes from the employee training session.', '2023-11-20 12:00:00', '2023-11-21 08:30:00')
    (6, 'Budget Review', 'Reviewed and updated the quarterly budget.', '2023-11-25 16:45:00', '2023-11-26 11:00:00'),
    (7, 'Customer Feedback', 'Feedback from recent customer surveys.', '2023-12-01 10:30:00', '2023-12-01 10:30:00'),
    (8, 'Project Ideas', 'New project concepts and potential features.', '2023-12-05 14:00:00', '2023-12-06 09:00:00'),
    (9, 'Weekly Progress Report', 'Summary of team accomplishments for the week.', '2023-12-10 16:15:00', '2023-12-11 12:30:
    (10, 'Marketing Campaign Plan', 'Plan for the upcoming marketing campaign.', '2023-12-15 11:00:00', '2023-12-15 11:00:00
```

Optimizing Database

Inserting Instances One by One:

Pros:

You can specify individual values for each column and adjust as needed.

Ideal for manual or interactive data entry.

Easier to debug errors for individual instances.

Cons:

Slower for inserting large sets of data.

More verbose and may require repetitive SQL statements.

Writing a Single Command to Insert Multiple Instances:

Pros:

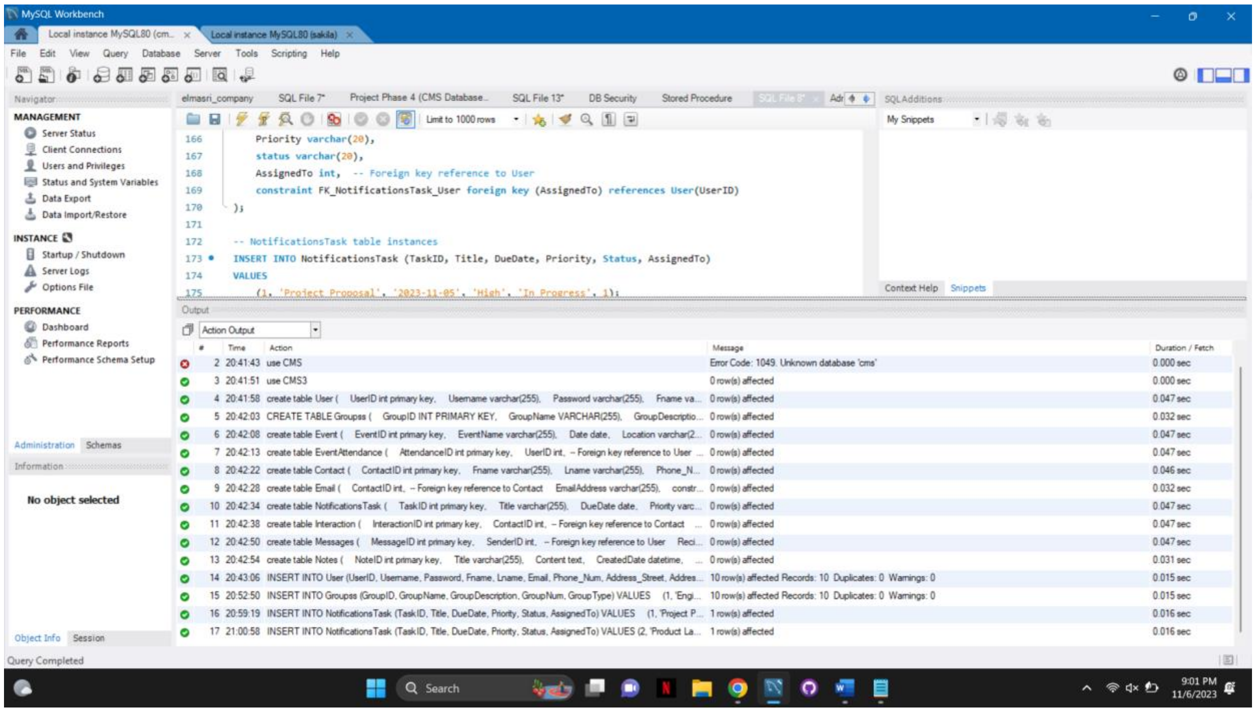
Faster for inserting many instances.

Less wordy and more efficient in terms of SQL statements.

Suitable for bulk data import or automated processes.

Cons:

Less control over individual instances, which may not be suitable for scenarios where each instance has unique values.



```sql
        Priority varchar(20),
        status varchar(20),
        AssignedTo int,  -- Foreign key reference to User
        constraint FK_NotificationsTask_User foreign key (AssignedTo) references User(UserID)
    );

    -- NotificationsTask table instances
    INSERT INTO NotificationsTask (TaskID, Title, DueDate, Priority, Status, AssignedTo)
    VALUES
        (1, 'Project Proposal', '2023-11-05', 'High', 'In Progress', 1);
```

```
#   Time      Action                                                                                                         Message                                                  Duration / Fetch
2   20:41:43  use CMS                                                                                                        Error Code: 1049. Unknown database 'cms'                 0.000 sec
3   20:41:51  use CMS3                                                                                                       0 row(s) affected                                        0.000 sec
4   20:41:58  create table User (   UserID int primary key,   Username varchar(255),   Password varchar(255),   Fname va...     0 row(s) affected                                        0.047 sec
5   20:42:03  CREATE TABLE Groupss (   GroupID INT PRIMARY KEY,   GroupName VARCHAR(255),   GroupDescriptio...                0 row(s) affected                                        0.032 sec
6   20:42:08  create table Event (   EventID int primary key,   EventName varchar(255),   Date date,   Location varchar(2...    0 row(s) affected                                        0.047 sec
7   20:42:13  create table EventAttendance (   AttendanceID int primary key,   UserID int,  -- Foreign key reference to User ... 0 row(s) affected                                     0.047 sec
8   20:42:22  create table Contact (   ContactID int primary key,   Fname varchar(255),   Lname varchar(255),   Phone_N...     0 row(s) affected                                        0.046 sec
9   20:42:28  create table Email (   ContactID int,  -- Foreign key reference to Contact   EmailAddress varchar(255),   constr... 0 row(s) affected                                     0.032 sec
10  20:42:34  create table NotificationsTask (   TaskID int primary key,   Title varchar(255),   DueDate date,   Priority varc... 0 row(s) affected                                      0.047 sec
11  20:42:38  create table Interaction (   InteractionID int primary key,   ContactID int,  -- Foreign key reference to Contact   ... 0 row(s) affected                            0.047 sec
12  20:42:50  create table Messages (   MessageID int primary key,   SenderID int,  -- Foreign key reference to User   Reci... 0 row(s) affected                                    0.047 sec
13  20:42:54  create table Notes (   NoteID int primary key,   Title varchar(255),   Content text,   CreatedDate datetime,   ... 0 row(s) affected                                      0.031 sec
14  20:43:06  INSERT INTO User (UserID, Username, Password, Fname, Lname, Email, Phone_Num, Address_Street, Addres...            10 row(s) affected Records: 10  Duplicates: 0  Warnings: 0  0.015 sec
15  20:52:50  INSERT INTO Groupss (GroupID, GroupName, GroupDescription, GroupNum, GroupType) VALUES   (1, 'Engi...           10 row(s) affected Records: 10  Duplicates: 0  Warnings: 0  0.015 sec
16  20:59:19  INSERT INTO NotificationsTask (TaskID, Title, DueDate, Priority, Status, AssignedTo) VALUES   (1, 'Project P...  1 row(s) affected                                        0.016 sec
17  21:00:58  INSERT INTO NotificationsTask (TaskID, Title, DueDate, Priority, Status, AssignedTo) VALUES (2, 'Product La...    1 row(s) affected                                        0.016 sec
```

As you can see in the picture above, the time to insert data one by one takes longer and will take longer to physically type out and insert each statement for each table. It’s a lot more efficient to type one command to insert all the data for the table.

Normalization Check

1NF:

- Our CMS database satisfies the first normalization check because it contains no repeating groups, and all attributes are atomic (indivisible).

2NF:

The second normal form requires that non-key attributes be fully functionally dependent on the entire primary key. In our database, it appears that the tables meet this requirement:

- In the User table, attributes like Fname, Lname, Email, and others are fully functionally dependent on UserID, which is the primary key.
- Similarly, in other tables, non-key attributes are dependent on their respective primary keys.
- Therefore, based on the structure and relationships we have defined, the database satisfies the second normal form. However, it's essential to ensure that the data integrity is maintained during actual data operations and that the relationships between tables are correctly enforced.

3NF:

- All non-prime (non-key) attributes must be functionally dependent on the primary key. This means that non-key attributes cannot be transitively dependent on the primary key through another non-key attribute.

- There should be no partial dependencies. A partial dependency occurs when a non-key attribute is functionally dependent on only a part of the primary key.

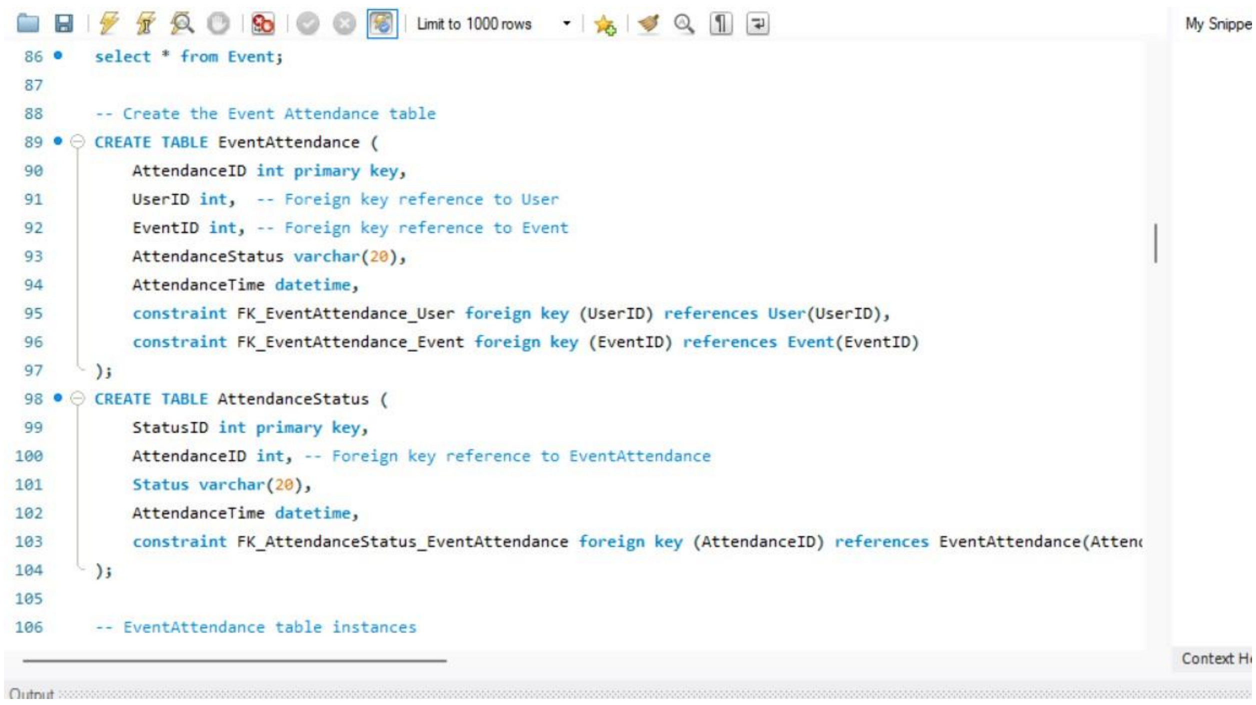
After reviewing our database, the tables EventAttendance and Interaction don’t pass the third normalization check.

Interaction Table:

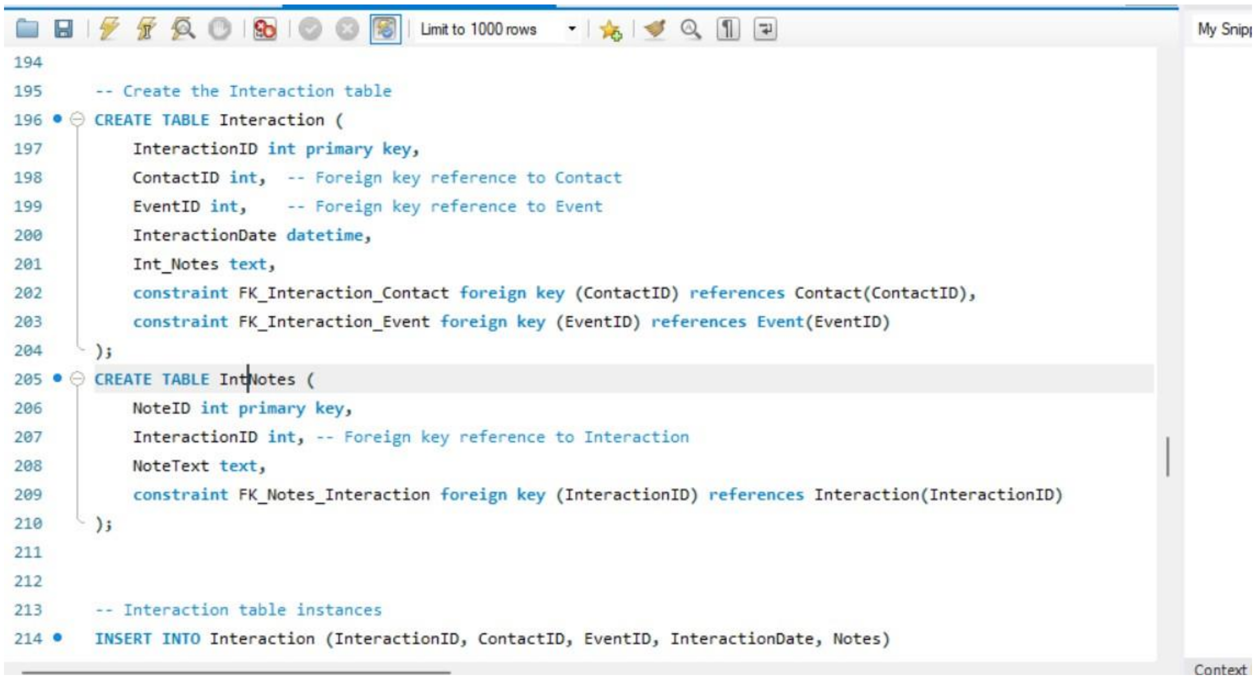
- The Interaction table is not in 3NF. The "IntNotes" attribute appears to be transitively dependent on the primary key (InteractionID) through the "ContactID" and "EventID" attributes. To achieve 3NF, we will create a separate table for "IntNotes" with references to "InteractionID."

EventAttendance Table:

- The EventAttendance table is not in 3NF. The "AttendanceStatus" attribute appears to be transitively dependent on the primary key (AttendanceID) through the "UserID" and "EventID" attributes. To achieve 3NF, we will create a separate table for "AttendanceStatus" with a reference to "AttendanceID."



```sql
select * from Event;

-- Create the Event Attendance table
CREATE TABLE EventAttendance (
    AttendanceID int primary key,
    UserID int,  -- Foreign key reference to User
    EventID int, -- Foreign key reference to Event
    AttendanceStatus varchar(20),
    AttendanceTime datetime,
    constraint FK_EventAttendance_User foreign key (UserID) references User(UserID),
    constraint FK_EventAttendance_Event foreign key (EventID) references Event(EventID)
);
CREATE TABLE AttendanceStatus (
    StatusID int primary key,
    AttendanceID int, -- Foreign key reference to EventAttendance
    Status varchar(20),
    AttendanceTime datetime,
    constraint FK_AttendanceStatus_EventAttendance foreign key (AttendanceID) references EventAttendance(Attend
);

-- EventAttendance table instances
```



```sql

-- Create the Interaction table
CREATE TABLE Interaction (
    InteractionID int primary key,
    ContactID int,  -- Foreign key reference to Contact
    EventID int,    -- Foreign key reference to Event
    InteractionDate datetime,
    Int_Notes text,
    constraint FK_Interaction_Contact foreign key (ContactID) references Contact(ContactID),
    constraint FK_Interaction_Event foreign key (EventID) references Event(EventID)
);
CREATE TABLE IntNotes (
    NoteID int primary key,
    InteractionID int, -- Foreign key reference to Interaction
    NoteText text,
    constraint FK_Notes_Interaction foreign key (InteractionID) references Interaction(InteractionID)
);


-- Interaction table instances
INSERT INTO Interaction (InteractionID, ContactID, EventID, InteractionDate, Notes)
```

UX Design

Login Page:

Functionality: Users can enter their credentials (username and password) to access the CMS.

Features:

Input fields for username and password.

"Forgot Password" option for password retrieval.

"Sign Up" option for new users.

Dashboard/Main Menu Page:

Functionality: Provides an overview of recent activities, events, tasks, and messages.

Features:

Widgets for upcoming events, tasks, and recent interactions.

Quick links to important sections such as User Management, Event Management, and Task Management.

User Management Page:

Functionality: Enables administrators to manage user accounts, permissions, and roles.

Features:

List of all users with details like username, email, and group.

Add, edit, and delete user functionalities.

Sorting and filtering options for easy user management.

Event Management Page:

Functionality: Allows users to create, edit, and manage events within the CMS.

Features:

List of all scheduled events with details such as date, location, and description.

Add, edit, and delete event functionalities.

Search and filter options for quick event retrieval.

Interaction Management Page:

Functionality: Provides an overview of interactions between users and contacts.

Features:

List of all logged interactions with details such as date, time, and notes.

Add, edit, and delete interaction functionalities.

Filtering options based on contact, event, or date.

Messaging System Page:

Functionality: Facilitates communication between users within the CMS.

Features:

Compose, reply to, and delete messages functionalities.

Threaded conversations for tracking communication history.

Inbox and Sent items sections for managing messages.

Task Management Page:

Functionality: Enables users to create, assign, and monitor tasks within the CMS.

Features:

List of all tasks with details such as title, due date, and priority.

Add, edit, and delete task functionalities.

Sorting and filtering options for efficient task management.

Contact Management Page:

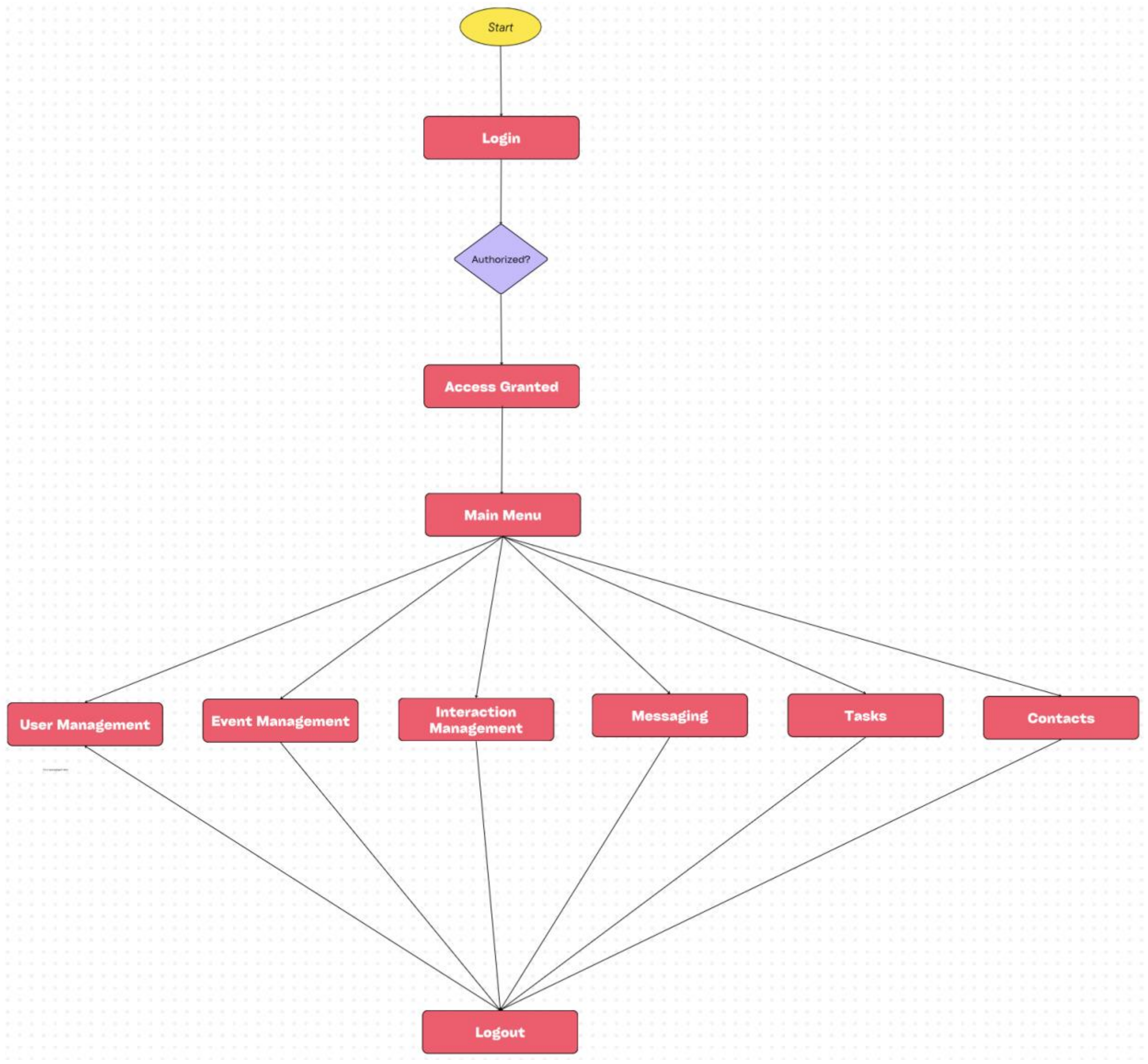
Functionality: Provides a centralized repository for managing contact information.

Features:

List of all contacts with details such as name, phone number, and email.

Add, edit, and delete contact functionalities.

Search and filter options for easy contact retrieval.


Start
Login
Authorized?
Access Granted
Main Menu
User Management
Event Management
Interaction Management
Messaging
Tasks
Contacts
Logout

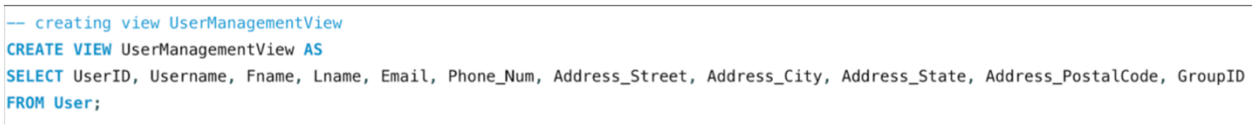
Required Views

User Management Page:

- This page allows administrators to manage user accounts, permissions, and roles.
- Users can view, add, edit, and delete user information.

User Management View Implementation Description:

The view 'UserManagementView' fetches data from the 'User' table, providing a comprehensive view of user details for efficient management.



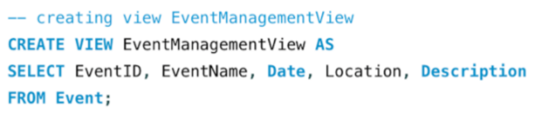
```
-- creating view UserManagementView
CREATE VIEW UserManagementView AS
SELECT UserID, Username, Fname, Lname, Email, Phone_Num, Address_Street, Address_City, Address_State, Address_PostalCode, GroupID
FROM User;
```

Event Management Page:

- This page facilitates the creation, editing, and management of various events within the CMS.
- Users can add, edit, and delete event details.

Event Management View Implementation Description:

The 'EventManagementView' retrieves data from the 'Event' table, presenting an organized view of event-related information for effective event management.



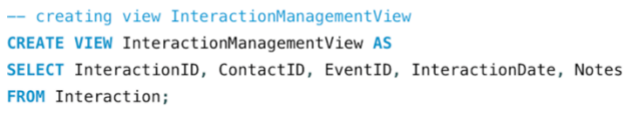
```sql
-- creating view EventManagementView
CREATE VIEW EventManagementView AS
SELECT EventID, EventName, Date, Location, Description
FROM Event;
```

Interaction Management Page:

- This page provides an overview of interactions between users and contacts.
- Users can add, edit, and delete interaction logs.

Interaction Management View Implementation Description:

The 'InteractionManagementView' retrieves data from the 'Interaction' table, presenting a comprehensive view of interaction logs for effective tracking and management.



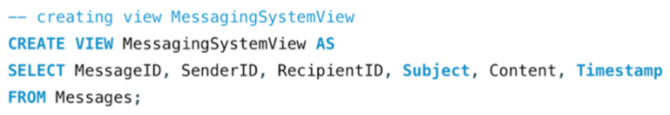
```sql
-- creating view InteractionManagementView
CREATE VIEW InteractionManagementView AS
SELECT InteractionID, ContactID, EventID, InteractionDate, Notes
FROM Interaction;
```

Messaging System Page:

- This page facilitates communication between users within the CMS.
- Users can compose, reply to, and delete messages.

Messaging System View Implementation Description:

The 'MessagingSystemView' fetches data from the 'Messages' table, enabling users to manage communication efficiently within the CMS.



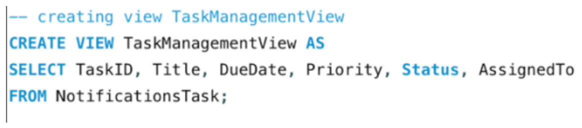
```sql
-- creating view MessagingSystemView
CREATE VIEW MessagingSystemView AS
SELECT MessageID, SenderID, RecipientID, Subject, Content, Timestamp
FROM Messages;
```

Task Management Page:

- This page allows users to create, assign, and monitor tasks within the CMS.
- Users can add, edit, and delete tasks efficiently.

Task Management View Implementation Description:

The 'TaskManagementView' retrieves data from the 'NotificationsTask' table, providing a comprehensive view of tasks for effective task management within the CMS.



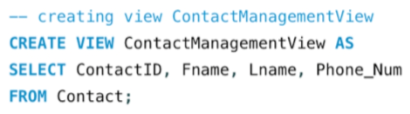
```sql
-- creating view TaskManagementView
CREATE VIEW TaskManagementView AS
SELECT TaskID, Title, DueDate, Priority, Status, AssignedTo
FROM NotificationsTask;
```

Contact Management Page:

- This page serves as a centralized repository for managing contact information.
- Users can add, edit, and delete contact details.

Contact Management View Implementation Description:

The 'ContactManagementView' fetches data from the 'Contact' table, providing a comprehensive view of contact details for efficient management within the CMS.

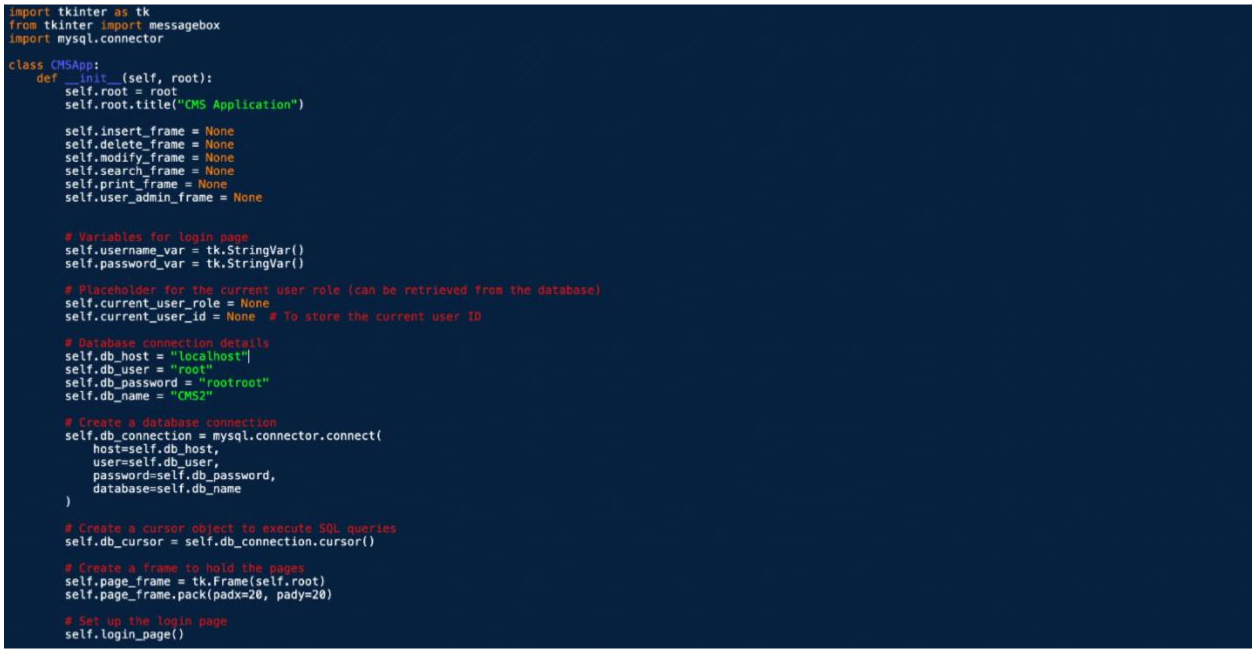


```sql
-- creating view ContactManagementView
CREATE VIEW ContactManagementView AS
SELECT ContactID, Fname, Lname, Phone_Num
FROM Contact;
```

Graphical User Interface Design:

Database Connection:

Python code that shows how we connected our SQL Database to our python GUI. This connection allowed us to connect our CMS database to. Out python GUI.

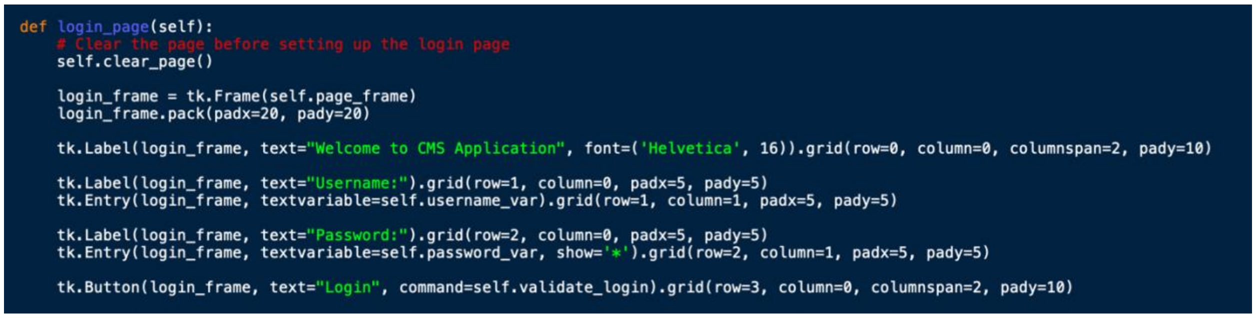


```python
import tkinter as tk
from tkinter import messagebox
import mysql.connector

class CMSApp:
    def __init__(self, root):
        self.root = root
        self.root.title("CMS Application")

        self.insert_frame = None
        self.delete_frame = None
        self.modify_frame = None
        self.search_frame = None
        self.print_frame = None
        self.user_admin_frame = None


        # Variables for login page
        self.username_var = tk.StringVar()
        self.password_var = tk.StringVar()

        # Placeholder for the current user role (can be retrieved from the database)
        self.current_user_role = None
        self.current_user_id = None  # To store the current user ID

        # Database connection details
        self.db_host = "localhost"
        self.db_user = "root"
        self.db_password = "rootroot"
        self.db_name = "CMS2"

        # Create a database connection
        self.db_connection = mysql.connector.connect(
            host=self.db_host,
            user=self.db_user,
            password=self.db_password,
            database=self.db_name
        )

        # Create a cursor object to execute SQL queries
        self.db_cursor = self.db_connection.cursor()

        # Create a frame to hold the pages
        self.page_frame = tk.Frame(self.root)
        self.page_frame.pack(padx=20, pady=20)

        # Set up the login page
        self.login_page()
```

Figure 1. Database Connection Python Script

Login Page:

The login page serves as the entry point for authorized users. It prompts users to enter their credentials (username and password) to gain access to the content management system.



```python
def login_page(self):
    # Clear the page before setting up the login page
    self.clear_page()

    login_frame = tk.Frame(self.page_frame)
    login_frame.pack(padx=20, pady=20)

    tk.Label(login_frame, text="Welcome to CMS Application", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    tk.Label(login_frame, text="Username:").grid(row=1, column=0, padx=5, pady=5)
    tk.Entry(login_frame, textvariable=self.username_var).grid(row=1, column=1, padx=5, pady=5)

    tk.Label(login_frame, text="Password:").grid(row=2, column=0, padx=5, pady=5)
    tk.Entry(login_frame, textvariable=self.password_var, show='*').grid(row=2, column=1, padx=5, pady=5)

    tk.Button(login_frame, text="Login", command=self.validate_login).grid(row=3, column=0, columnspan=2, pady=10)
```

Figure 2. Login Page Python Script

The above code defines the login page. User inputs for the username and password are associated with variables (self.username_var and self.password_var). The code employs Tkinter widgets such as Label, Entry, and Button to structure and organize the graphical elements within a frame. The validate_login method is hinted to handle the login validation logic when the user clicks the login button. Overall, this code sets up the visual components and layout for a basic login page.

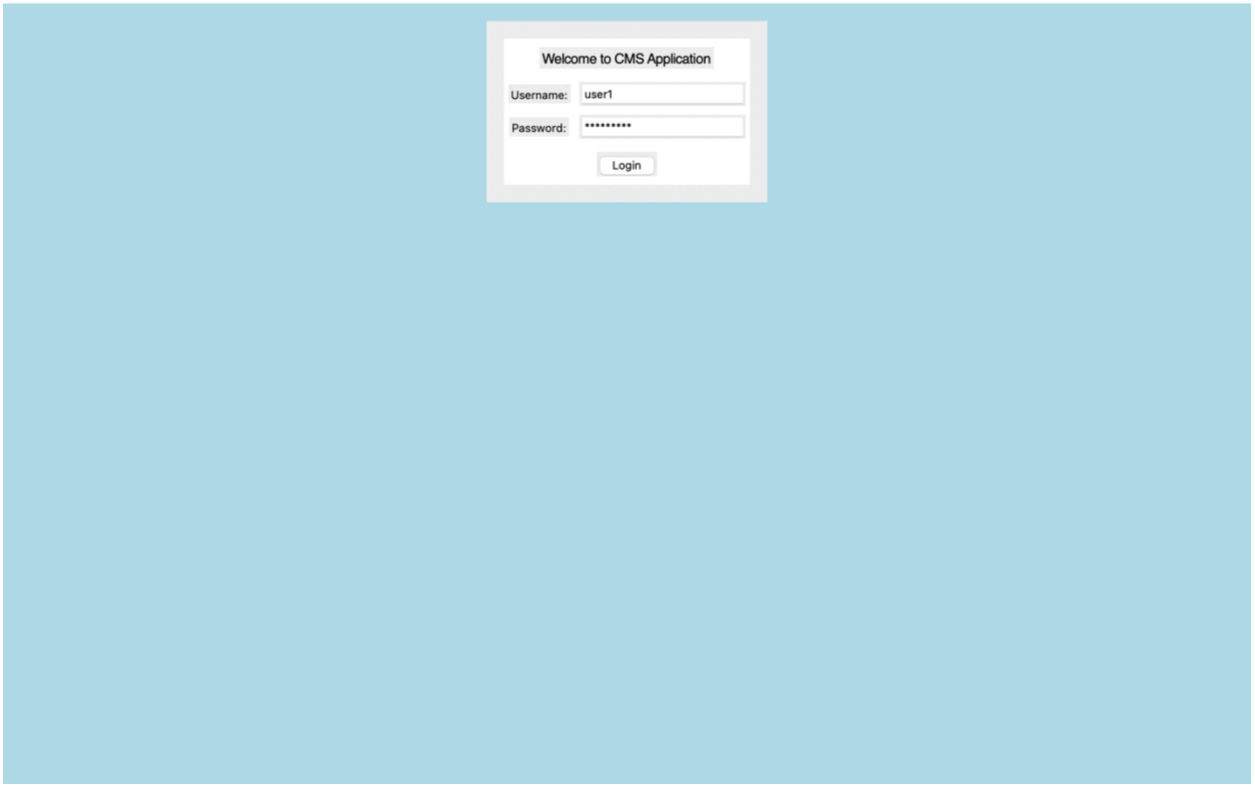
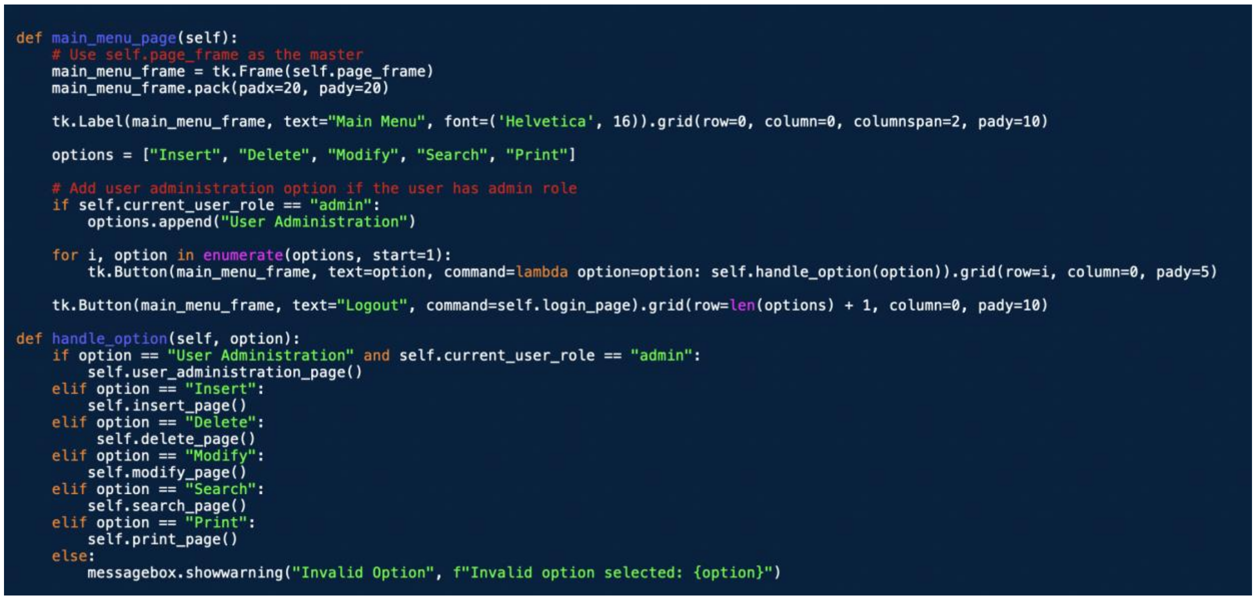


Figure 3. CMS Application Login Page

Main Menu:

The main menu page acts as the central hub for users who have successfully logged in. It provides navigation options and may display key information, offering a gateway to various functionalities within the CMS. This page includes buttons that lead to the insert page, delete page, modify page, search page, print page, change password page, and finally the logout page.



```python
def main_menu_page(self):
    # Use self.page_frame as the master
    main_menu_frame = tk.Frame(self.page_frame)
    main_menu_frame.pack(padx=20, pady=20)

    tk.Label(main_menu_frame, text="Main Menu", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    options = ["Insert", "Delete", "Modify", "Search", "Print"]

    # Add user administration option if the user has admin role
    if self.current_user_role == "admin":
        options.append("User Administration")

    for i, option in enumerate(options, start=1):
        tk.Button(main_menu_frame, text=option, command=lambda option=option: self.handle_option(option)).grid(row=i, column=0, pady=5)

    tk.Button(main_menu_frame, text="Logout", command=self.login_page).grid(row=len(options) + 1, column=0, pady=10)

def handle_option(self, option):
    if option == "User Administration" and self.current_user_role == "admin":
        self.user_administration_page()
    elif option == "Insert":
        self.insert_page()
    elif option == "Delete":
         self.delete_page()
    elif option == "Modify":
        self.modify_page()
    elif option == "Search":
        self.search_page()
    elif option == "Print":
        self.print_page()
    else:
        messagebox.showwarning("Invalid Option", f"Invalid option selected: {option}")
```

Figure 4. Main Menu Python Script

The above code defines the main menu page. This method is responsible for creating and displaying the main menu page on the GUI. The main menu consists of a heading labeled "Main Menu" with an increased font size, and a set of buttons corresponding to various options, such as "Insert," "Delete," "Modify," "Search," "Print," and "Change Password. The code dynamically creates buttons for each option using a loop, and each button is associated with a command to handle the selected option, utilizing the handle_option method. Additionally, a "Logout" button is provided at the bottom of the menu, associated with the login_page method to facilitate user logout.

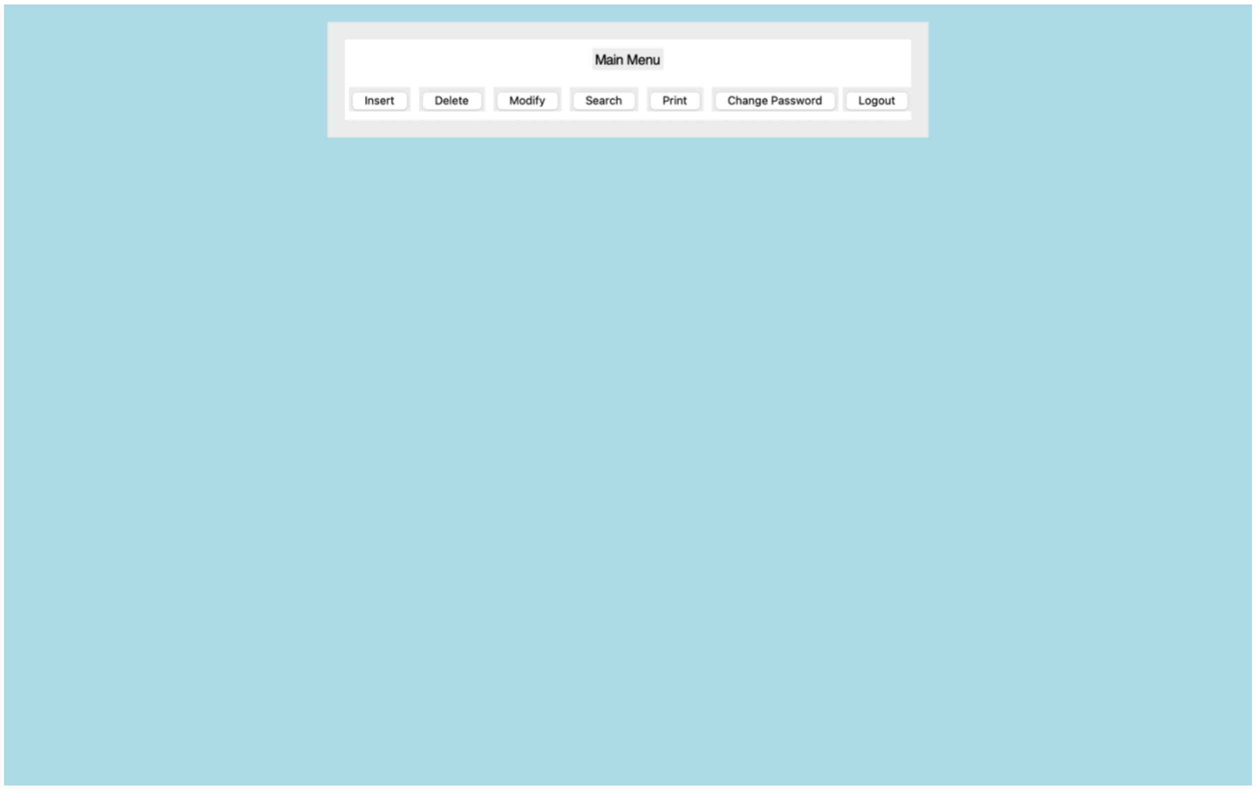
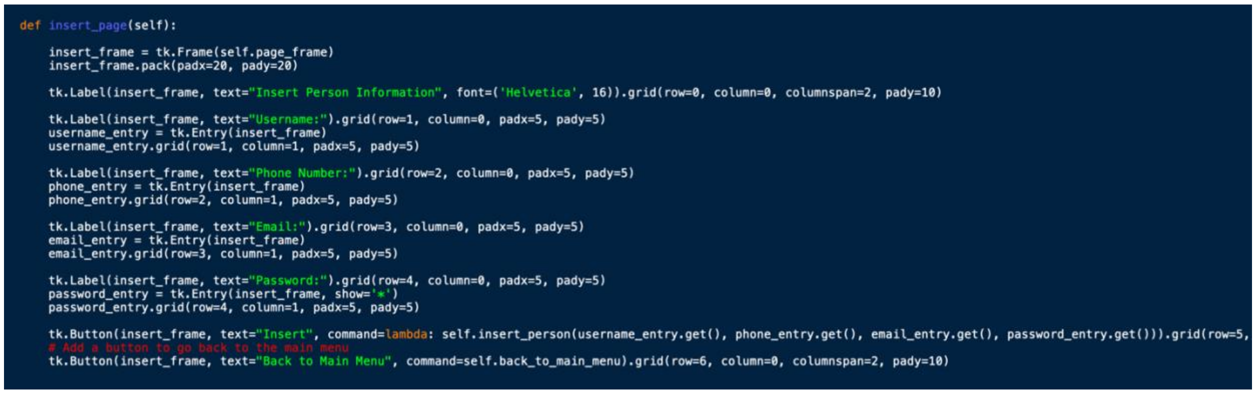


Figure 5. CMS Application: Main Menu

Insert Page:

The insert user page facilitates the addition of new users to the CMS. Administrators or authorized personnel can input relevant user details such as username, phone number, and email.



```python
def insert_page(self):

    insert_frame = tk.Frame(self.page_frame)
    insert_frame.pack(padx=20, pady=20)

    tk.Label(insert_frame, text="Insert Person Information", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    tk.Label(insert_frame, text="Username:").grid(row=1, column=0, padx=5, pady=5)
    username_entry = tk.Entry(insert_frame)
    username_entry.grid(row=1, column=1, padx=5, pady=5)

    tk.Label(insert_frame, text="Phone Number:").grid(row=2, column=0, padx=5, pady=5)
    phone_entry = tk.Entry(insert_frame)
    phone_entry.grid(row=2, column=1, padx=5, pady=5)

    tk.Label(insert_frame, text="Email:").grid(row=3, column=0, padx=5, pady=5)
    email_entry = tk.Entry(insert_frame)
    email_entry.grid(row=3, column=1, padx=5, pady=5)

    tk.Label(insert_frame, text="Password:").grid(row=4, column=0, padx=5, pady=5)
    password_entry = tk.Entry(insert_frame, show='*')
    password_entry.grid(row=4, column=1, padx=5, pady=5)

    tk.Button(insert_frame, text="Insert", command=lambda: self.insert_person(username_entry.get(), phone_entry.get(), email_entry.get(), password_entry.get())).grid(row=5,
    # Add a button to go back to the main menu
    tk.Button(insert_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=6, column=0, columnspan=2, pady=10)
```

Figure 6. Insert Page Python Script

The above code defines the insert page for our python GUI. This method is responsible for creating and displaying an insertion page on the GUI, specifically designed for inserting person information. The page includes labels and entry fields for the person's username, phone number, email, and password. Additionally, the page features "Insert" and "Back to Main Menu" buttons. The "Insert" button is associated with a lambda function that retrieves the user-inputted values from the entry fields and calls the insert_person method with these values, presumably to insert the person's information into the database.

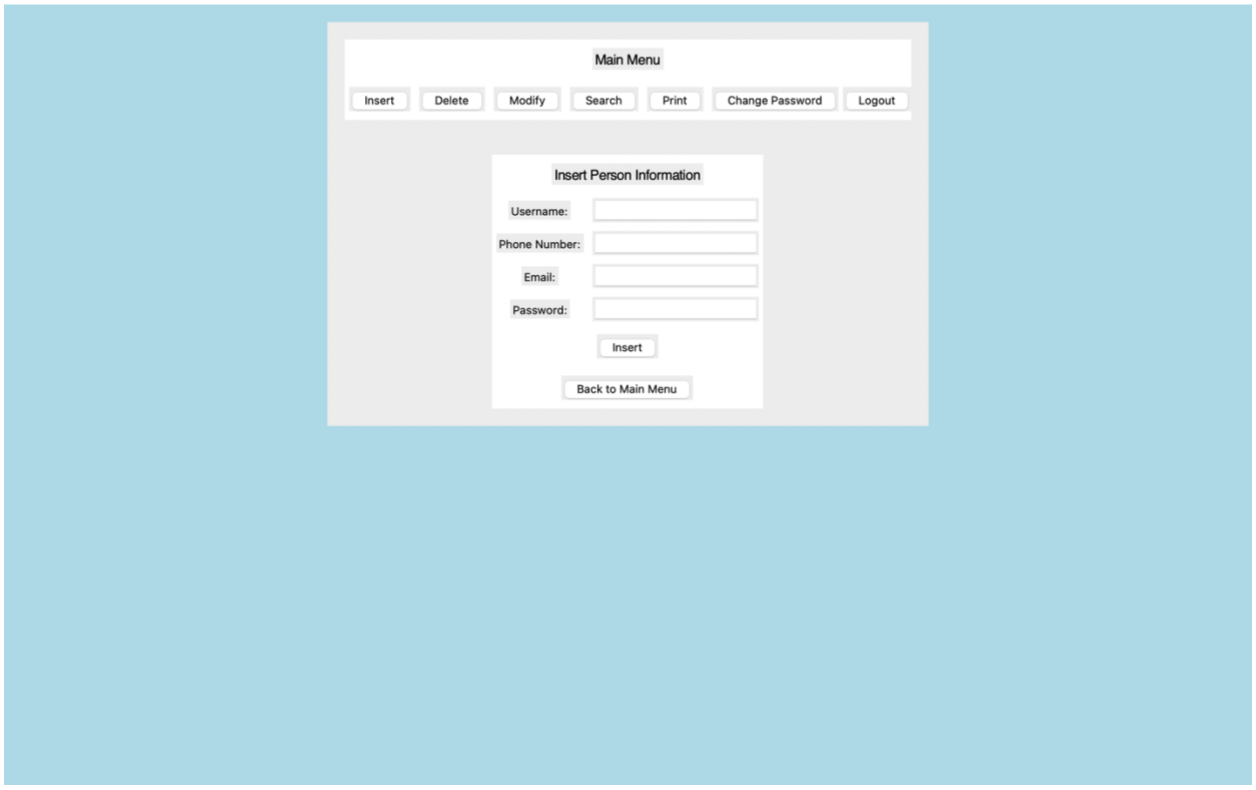
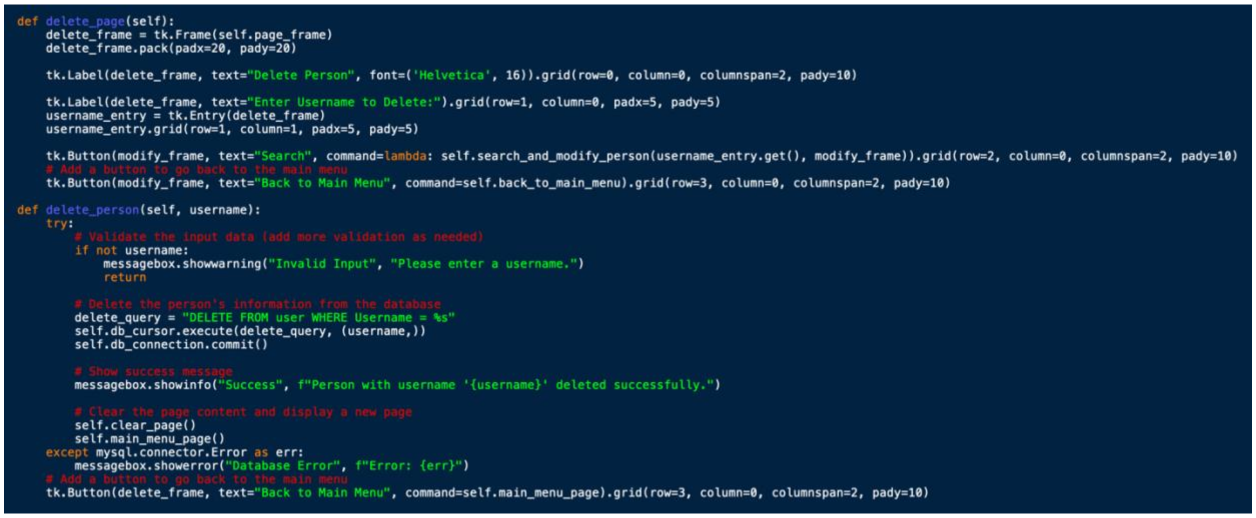


Figure 7. CMS Application Insert Page

Delete Page:

The delete user page allows administrators to remove user accounts from the system. It prompts the user to enter a username to be deleted. At the bottom of the page is a button which allows the user to return to the main menu.



```python
def delete_page(self):
    delete_frame = tk.Frame(self.page_frame)
    delete_frame.pack(padx=20, pady=20)

    tk.Label(delete_frame, text="Delete Person", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    tk.Label(delete_frame, text="Enter Username to Delete:").grid(row=1, column=0, padx=5, pady=5)
    username_entry = tk.Entry(delete_frame)
    username_entry.grid(row=1, column=1, padx=5, pady=5)

    tk.Button(modify_frame, text="Search", command=lambda: self.search_and_modify_person(username_entry.get(), modify_frame)).grid(row=2, column=0, columnspan=2, pady=10)
    # Add a button to go back to the main menu
    tk.Button(modify_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=3, column=0, columnspan=2, pady=10)

def delete_person(self, username):
    try:
        # Validate the input data (add more validation as needed)
        if not username:
            messagebox.showwarning("Invalid Input", "Please enter a username.")
            return

        # Delete the person's information from the database
        delete_query = "DELETE FROM user WHERE Username = %s"
        self.db_cursor.execute(delete_query, (username,))
        self.db_connection.commit()

        # Show success message
        messagebox.showinfo("Success", f"Person with username '{username}' deleted successfully.")

        # Clear the page content and display a new page
        self.clear_page()
        self.main_menu_page()
    except mysql.connector.Error as err:
        messagebox.showerror("Database Error", f"Error: {err}")
    # Add a button to go back to the main menu
    tk.Button(delete_frame, text="Back to Main Menu", command=self.main_menu_page).grid(row=3, column=0, columnspan=2, pady=10)
```

Figure 8. Delete Page Python Script

This is a Python code snippet that defines a class method delete_page and another method delete_person. The delete_page method creates a frame and adds a label and two buttons to it. The delete_person method takes a username parameter, validates it, and deletes the person's information from the database if the username exists. If the deletion is successful, a success message is displayed, and the page is cleared, and the main menu is displayed. If the username is not found, a warning message is displayed. If there is an error in the database, an error message is displayed.

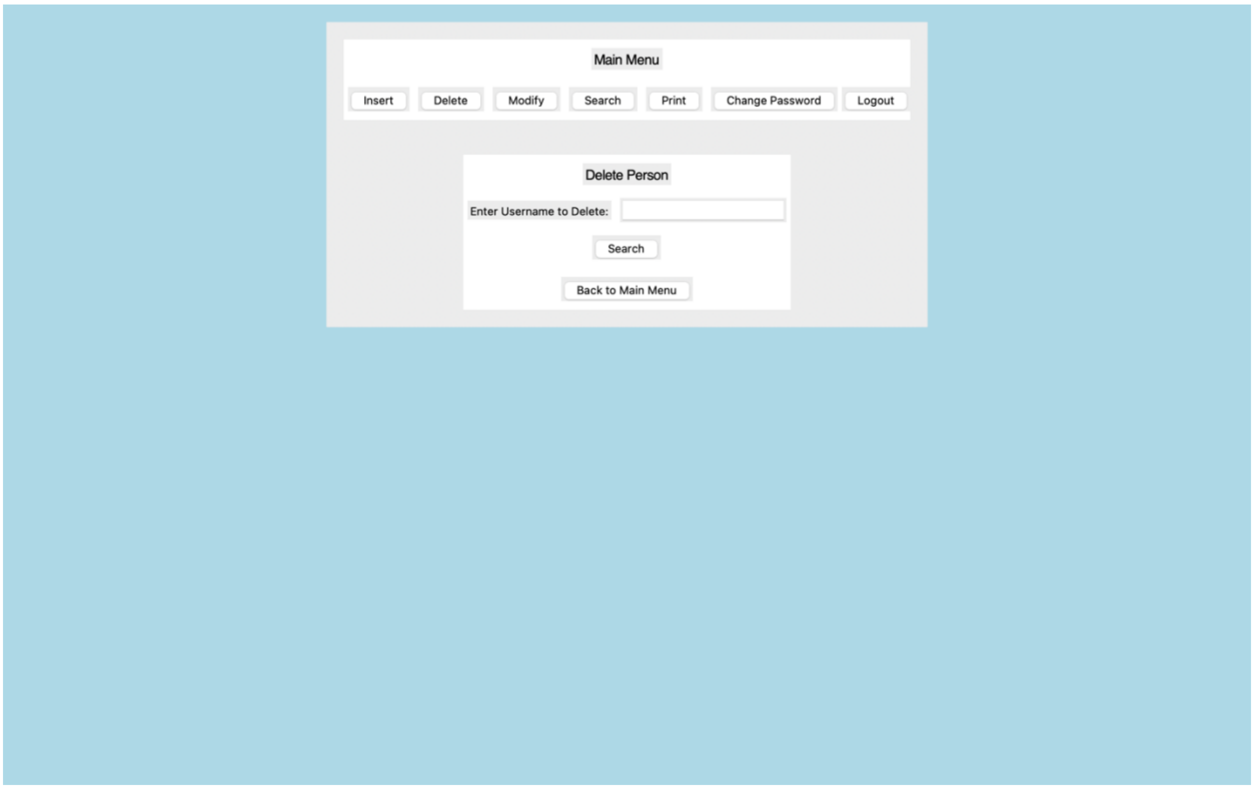
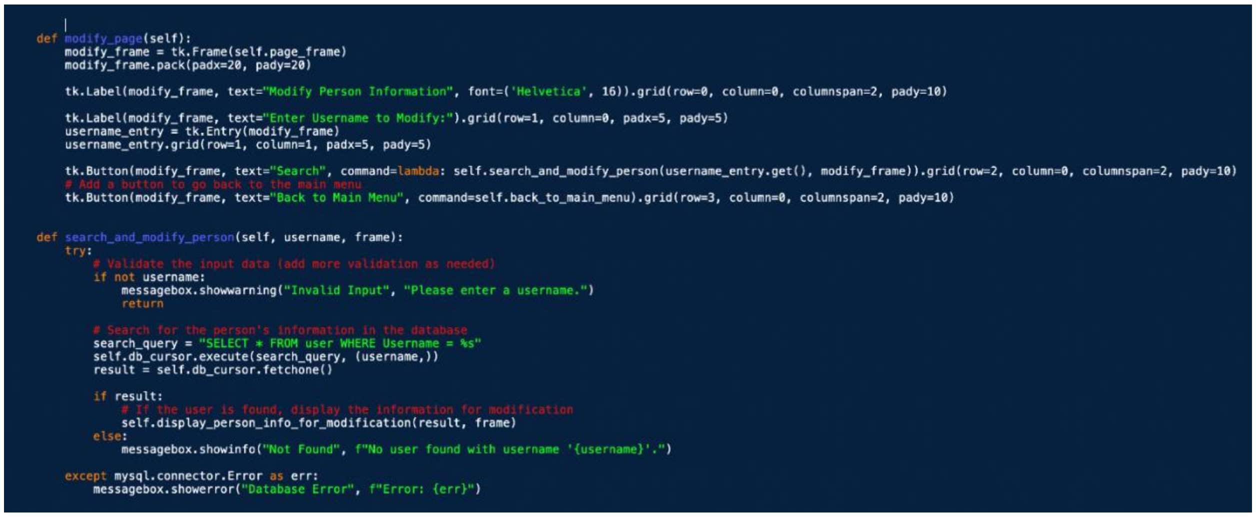


Figure 9. CMS Application Delete Page

Modify Page:

The modify page gives users the ability to change attributes of a specific user. This page prompts the user to enter a username to modify which will then give the user capabilities to modify different attributes of the user.



```python
def modify_page(self):
    modify_frame = tk.Frame(self.page_frame)
    modify_frame.pack(padx=20, pady=20)

    tk.Label(modify_frame, text="Modify Person Information", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    tk.Label(modify_frame, text="Enter Username to Modify:").grid(row=1, column=0, padx=5, pady=5)
    username_entry = tk.Entry(modify_frame)
    username_entry.grid(row=1, column=1, padx=5, pady=5)

    tk.Button(modify_frame, text="Search", command=lambda: self.search_and_modify_person(username_entry.get(), modify_frame)).grid(row=2, column=0, columnspan=2, pady=10)
    # Add a button to go back to the main menu
    tk.Button(modify_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=3, column=0, columnspan=2, pady=10)


def search_and_modify_person(self, username, frame):
    try:
        # Validate the input data (add more validation as needed)
        if not username:
            messagebox.showwarning("Invalid Input", "Please enter a username.")
            return

        # Search for the person's information in the database
        search_query = "SELECT * FROM user WHERE Username = %s"
        self.db_cursor.execute(search_query, (username,))
        result = self.db_cursor.fetchone()

        if result:
            # If the user is found, display the information for modification
            self.display_person_info_for_modification(result, frame)
        else:
            messagebox.showinfo("Not Found", f"No user found with username '{username}'.")

    except mysql.connector.Error as err:
        messagebox.showerror("Database Error", f"Error: {err}")
```

Figure 10. Modify Page Python Script

This is a Python code snippet that defines a class method modify_page and another method search_and_modify_person. The modify_page method creates a frame and adds a label, two labels, an entry, and two buttons to it. The search_and_modify_person method takes a username parameter, validates it, and searches for the person’s information in the database. If the user is found, the information is displayed for modification. If the username is not found, a message is displayed. If there is an error in the database, an error message is displayed.

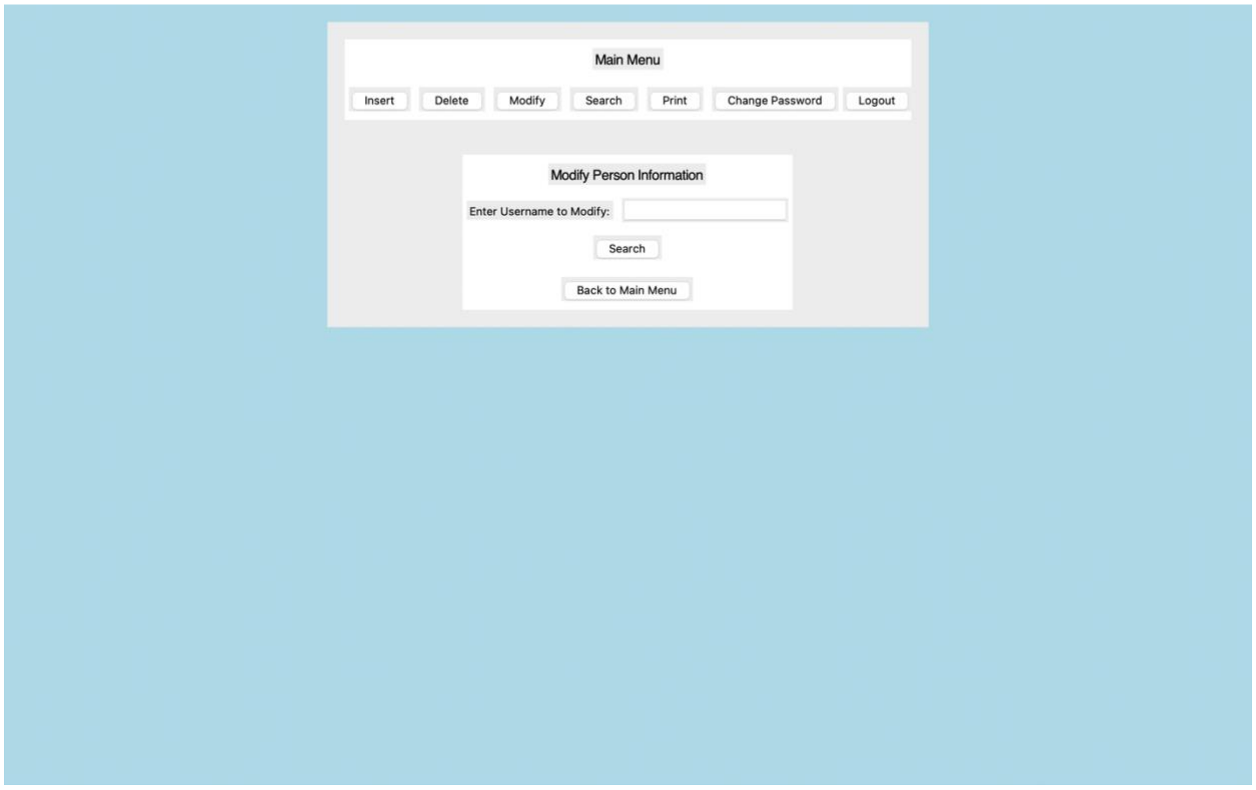
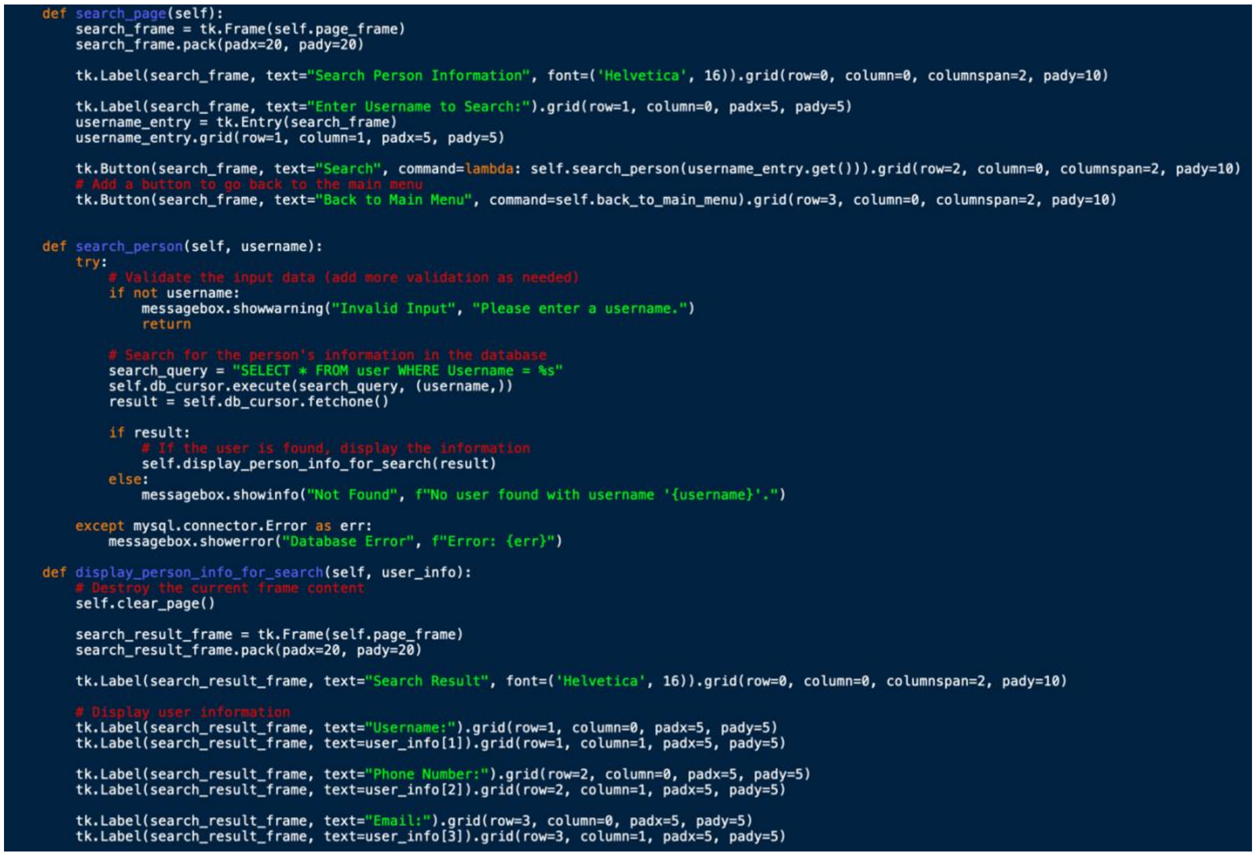


Figure 11. CMS Application Modify Page

Search Page:

The search page enables users to locate specific information within the CMS. Users can input search criteria (Username), and the system returns relevant results, enhancing efficiency in finding desired content or user profiles.



```python
    def search_page(self):
        search_frame = tk.Frame(self.page_frame)
        search_frame.pack(padx=20, pady=20)

        tk.Label(search_frame, text="Search Person Information", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

        tk.Label(search_frame, text="Enter Username to Search:").grid(row=1, column=0, padx=5, pady=5)
        username_entry = tk.Entry(search_frame)
        username_entry.grid(row=1, column=1, padx=5, pady=5)

        tk.Button(search_frame, text="Search", command=lambda: self.search_person(username_entry.get())).grid(row=2, column=0, columnspan=2, pady=10)
        # Add a button to go back to the main menu
        tk.Button(search_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=3, column=0, columnspan=2, pady=10)


    def search_person(self, username):
        try:
            # Validate the input data (add more validation as needed)
            if not username:
                messagebox.showwarning("Invalid Input", "Please enter a username.")
                return

            # Search for the person's information in the database
            search_query = "SELECT * FROM user WHERE Username = %s"
            self.db_cursor.execute(search_query, (username,))
            result = self.db_cursor.fetchone()

            if result:
                # If the user is found, display the information
                self.display_person_info_for_search(result)
            else:
                messagebox.showinfo("Not Found", f"No user found with username '{username}'.")

        except mysql.connector.Error as err:
            messagebox.showerror("Database Error", f"Error: {err}")

    def display_person_info_for_search(self, user_info):
        # Destroy the current frame content
        self.clear_page()

        search_result_frame = tk.Frame(self.page_frame)
        search_result_frame.pack(padx=20, pady=20)

        tk.Label(search_result_frame, text="Search Result", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

        # Display user information
        tk.Label(search_result_frame, text="Username:").grid(row=1, column=0, padx=5, pady=5)
        tk.Label(search_result_frame, text=user_info[1]).grid(row=1, column=1, padx=5, pady=5)

        tk.Label(search_result_frame, text="Phone Number:").grid(row=2, column=0, padx=5, pady=5)
        tk.Label(search_result_frame, text=user_info[2]).grid(row=2, column=1, padx=5, pady=5)

        tk.Label(search_result_frame, text="Email:").grid(row=3, column=0, padx=5, pady=5)
        tk.Label(search_result_frame, text=user_info[3]).grid(row=3, column=1, padx=5, pady=5)
```

Figure 12. Search Page Python Script

The search_person method takes a username parameter, validates it, and searches for the person’s information in the database. If the user is found, the information is displayed for modification. If the username is not found, a message is displayed. If there is an error in the database, an error message is displayed.

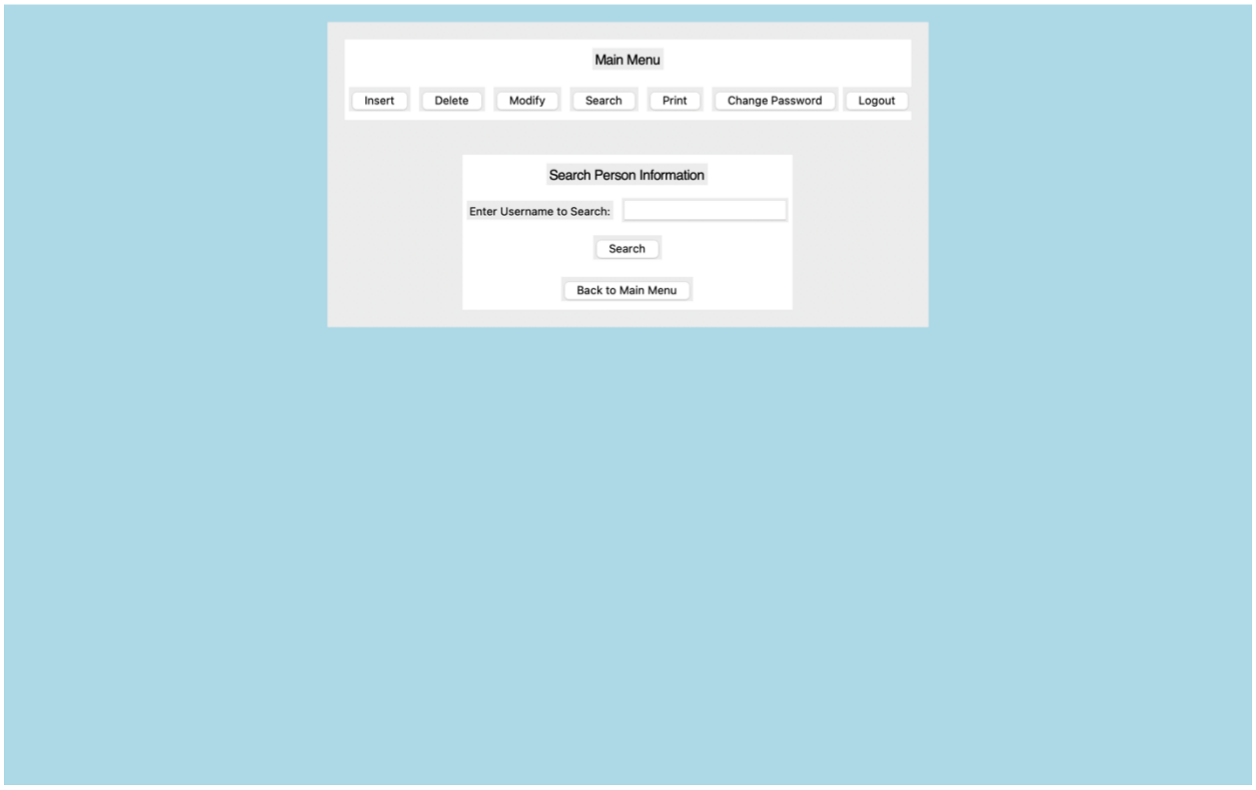
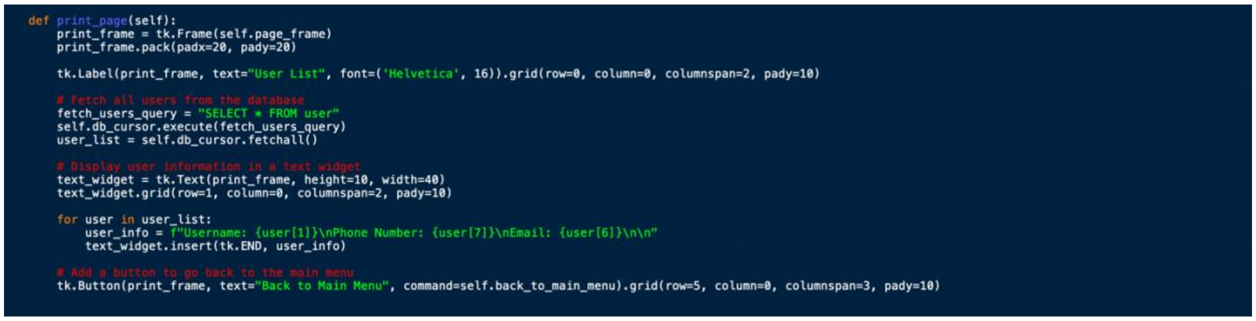


Figure 13. CMS Application Search Page

Print page:

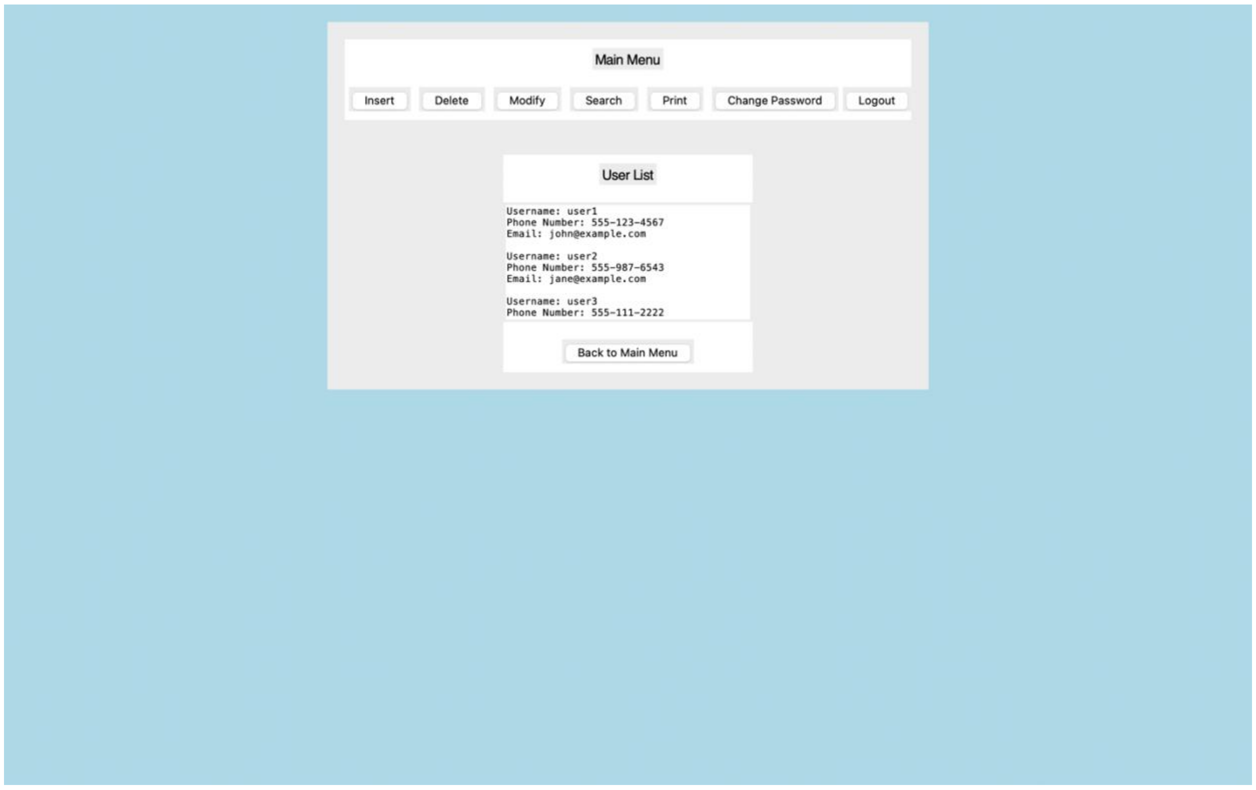
The print page displays a comprehensive list of registered users. It may include key details about each user, providing administrators with an overview of the user base and facilitating easy navigation.



```python
def print_page(self):
    print_frame = tk.Frame(self.page_frame)
    print_frame.pack(padx=20, pady=20)

    tk.Label(print_frame, text="User List", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    # Fetch all users from the database
    fetch_users_query = "SELECT * FROM user"
    self.db_cursor.execute(fetch_users_query)
    user_list = self.db_cursor.fetchall()

    # Display user information in a text widget
    text_widget = tk.Text(print_frame, height=10, width=40)
    text_widget.grid(row=1, column=0, columnspan=2, pady=10)

    for user in user_list:
        user_info = f"Username: {user[1]}\nPhone Number: {user[7]}\nEmail: {user[6]}\n\n"
        text_widget.insert(tk.END, user_info)

    # Add a button to go back to the main menu
    tk.Button(print_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=5, column=0, columnspan=3, pady=10)
```

Figure 14. Print Page Python Script

This is a Python code snippet that defines a class method print_page. The method creates a frame and adds a label, a text widget, and a button to it. It fetches all users from the database and displays their information in the text widget. The information includes the username, phone number, and email of each user. Finally, it adds a button to go back to the main menu.

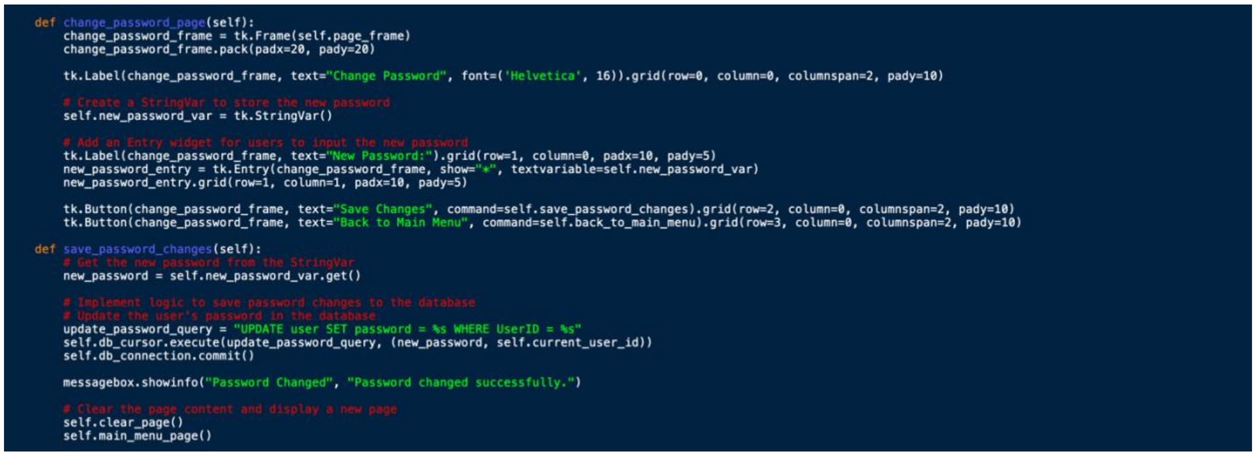


```
Username: user1
Phone Number: 555-123-4567
Email: john@example.com

Username: user2
Phone Number: 555-987-6543
Email: jane@example.com

Username: user3
Phone Number: 555-111-2222
```

Figure 15. CMS Application Print Page

Change Password Page:

The change password page is a dedicated space for users to change their current password. It prompts the user to enter their desired new password which is saved using a button at the bottom of the page that reads “Save Changes”. Below this button is the option to return to the main menu.



```python
def change_password_page(self):
    change_password_frame = tk.Frame(self.page_frame)
    change_password_frame.pack(padx=20, pady=20)

    tk.Label(change_password_frame, text="Change Password", font=('Helvetica', 16)).grid(row=0, column=0, columnspan=2, pady=10)

    # Create a StringVar to store the new password
    self.new_password_var = tk.StringVar()

    # Add an Entry widget for users to input the new password
    tk.Label(change_password_frame, text="New Password:").grid(row=1, column=0, padx=10, pady=5)
    new_password_entry = tk.Entry(change_password_frame, show="*", textvariable=self.new_password_var)
    new_password_entry.grid(row=1, column=1, padx=10, pady=5)

    tk.Button(change_password_frame, text="Save Changes", command=self.save_password_changes).grid(row=2, column=0, columnspan=2, pady=10)
    tk.Button(change_password_frame, text="Back to Main Menu", command=self.back_to_main_menu).grid(row=3, column=0, columnspan=2, pady=10)

def save_password_changes(self):
    # Get the new password from the StringVar
    new_password = self.new_password_var.get()

    # Implement logic to save password changes to the database
    # Update the user's password in the database
    update_password_query = "UPDATE user SET password = %s WHERE UserID = %s"
    self.db_cursor.execute(update_password_query, (new_password, self.current_user_id))
    self.db_connection.commit()

    messagebox.showinfo("Password Changed", "Password changed successfully.")

    # Clear the page content and display a new page
    self.clear_page()
    self.main_menu_page()
```

Figure 16. Change Password Page Python Script

This is a Python code snippet that defines two class methods change_password_page and save_password_changes. The change_password_page method creates a frame and adds a label, an entry, and two buttons to it. The save_password_changes method gets the new password from the StringVar, updates the user’s password in the database, shows a success message, clears the page content, and displays the main menu.

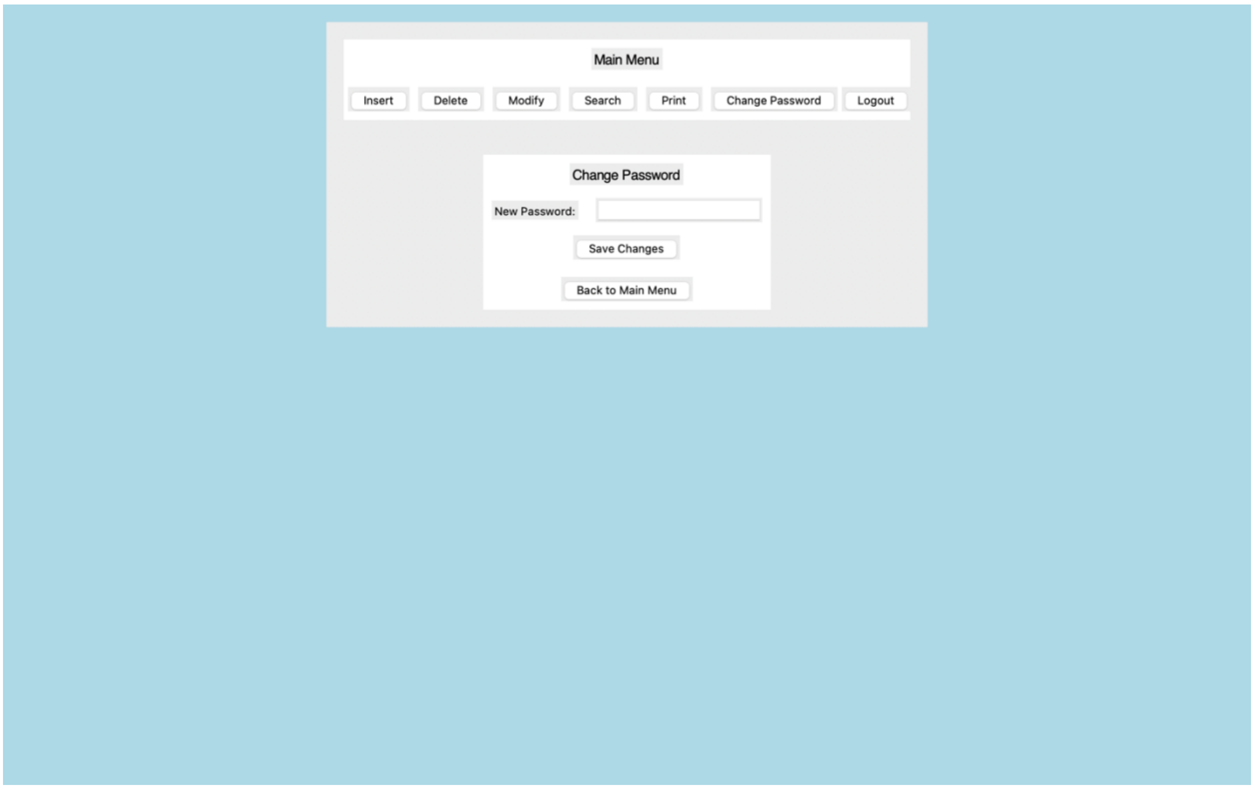


Figure 17. CMS Application Change Password Page

Conclusion and Future Work:

What we Learned:

This project has proven to be an exceptional learning journey. Beyond gaining a solid understanding of database management fundamentals, we immersed ourselves in the collaborative dynamics of team-based project work, a valuable experience with significant implications for future employment. Our initial phase involved meticulously outlining our database structure by identifying pertinent entities and attributes essential to our CMS. Subsequently, delving into the intricacies of Entity-Relationship diagrams provided a framework for constructing our SQL database. The creation of an Enhanced Entity Relationship diagram further elucidated the intricate connections between different entities, guiding us in comprehending how each entity interrelated. This process led to a crucial realization – the importance of simplicity in database design, a lesson that significantly streamlined our approach. Moving forward, the implementation phase required us to translate our conceptualized CMS into SQL, fostering proficiency and hands-on experience with SQL itself. Finally, establishing a connection between the database and a Python GUI was a challenging yet enlightening process, contributing to our enhanced proficiency in both SQL and basic Python.

Features we can add:

Although the application that we have made is adequate, there are still some features that we can include to make it even better. One way that we could improve our apllication is to make an email list. This could be very useful when trying to contact a whole team. A user can use the table of email to look up a group ID and email a certain team they are trying to contact all at once. Another feature that could be useful is instead of looking up the username, when searching for a person's email, they can just look up the name or last name. This is a better way to search for someone you are trying to contact because they might not have any clue what that person's username is. The way that we have it running now is probably better for an administrator's use. Also including a feature such as printing out all the information in our database is a security issue. A feature that we could include to the print option is to only print out information that is not harmful to the regular employee users' end. Otherwise someone could hack into our system and print out all the information in our database and alter it.

References:

Section 1.1

Source 1: https://www.zendesk.com/sell/crm/contact-management-software/

Source 2: https://www.getapp.com/customer-management-software/a/pipedrive/reviews/

Source 3: https://crm.org/news/streak-crm-review