ART GALLERY MANAGEMENT SYSTEM

A MINI PROJECT REPORT

Submitted by

VARUN KUMAR V	**220701311**
VARUN G	**220701310**

In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

RAJALAKSHMI ENGINEERING COLLEGE,

CHENNAI-602105

2024

BONAFIDE CERTIFICATE

Certified that this project report "**ART GALLERY MANAGEMENT SYSTEM**" is the bonafide work of **"VARUN KUMAR V (220701311), VARUN G (220701310) "** who carried out the project work under my supervision.

Submitted for the Practical Examination held on ________________

SIGNATURE

Dr.R.SABITHA
Professor and II Year Academic Head
Computer Science and Engineering,
Rajalakshmi Engineering College
(Autonomous),
Thandalam, Chennai - 602 105

SIGNATURE

Ms.V.JANANEE
Assistant Professor (SG),
Computer Science and Engineering,
Rajalakshmi Engineering College,
(Autonomous),
Thandalam, Chennai - 602 105

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

Art galleries play a significant role as cultural hubs, preserving and exhibiting artworks for public enjoyment. Nevertheless, administering the operations of an art gallery entails numerous hurdles, such as artwork inventory supervision, artist cooperation, exhibition scheduling, and patron interaction. Present methodologies frequently depend on manual procedures, resulting in inefficiencies, inaccuracies, and restricted accessibility. To overcome these obstacles and contemporize gallery administration, there arises a necessity for an integrated Art Gallery Management System. This system would streamline operations, improve collaboration, and elevate the visitor experience, thus revitalizing gallery management practices.

The objective of the Art Gallery Management System project is to create an all-encompassing and intuitive web platform that facilitates the efficient functioning of art galleries. It aims to improve collaboration with artists, streamline exhibition planning, and elevate the overall visitor experience. By harnessing contemporary technologies, the system endeavors to modernize gallery management practices, stimulate artistic involvement, and cultivate a dynamic and inclusive art community.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 INTRODUCTION

1.2 OBJECTIVES

1.3 MODULES

2. SURVEY OF TECHNOLOGIES

2.1 SOFTWARE DESCRIPTION

2.2 LANGUAGES

2.2.1 SQL

2.2.2 PYTHON

3. REQUIREMENTS AND ANALYSIS

3.1 REQUIREMENT SPECIFICATION

3.2 HARDWARE AND SOFTWARE REQUIREMENTS

3.3 ARCHITECTURE DIAGRAM

3.4 ER DIAGRAM

3.5 NORMALIZATION

4. PROGRAM CODE

5. RESULTS AND DISCUSSION

6. CONCLUSION

7. REFERENCES

CH 1. INTRODUCTION

1.1 Introduction

The Canvas stands as Art Gallery Management System crafted to streamline the administration and exhibition of artworks within art galleries. This comprehensive solution harnesses contemporary web technologies to furnish gallery owners, curators, artists, and visitors with a fluid and captivating experience.

1.2 Objective

The main objective of the Art Gallery Management System is to efficiently manage various aspects of gallery operations, including inventory, sales, exhibitions, customer relationships, and administrative tasks. It aims to streamline and automate the cataloging of artworks with detailed information, track their location, and handle sales transactions, including invoicing and receipts. The system facilitates the planning and organization of exhibitions and events, managing guest lists and RSVPs. Additionally, it maintains a comprehensive database of customers, artists, and collectors, tracking interactions to enhance engagement and personalized communication. It generates reports on sales, inventory, and customer data, providing valuable insights for decision-making. The system ensures secure data storage and compliance with legal requirements, manages the gallery's online presence, and promotes artists and their works. Designed for administrative use, it focuses on improving operational efficiency, data security, and user experience by automating routine tasks and maintaining data integrity.

1.3 MODULE

- ADD NEW ALBUMS
- ADD PICTURES
- ADD DESCRIPTION
- EDIT ALBUM
- DROP ALBUM

CH 2. SURVEY OF TECHNOLOGY

2.1 SOFTWARE DESCRIPTION

Visual studio Code Visual Studio Code combines the simplicity of a source code editor with powerful developer tooling, like IntelliSense code completion and debugging. First and foremost, it is an editor that gets out of your way. The delightfully frictionless edit-build-debug cycle means less time fiddling with your environment, and more time executing on your ideas.

2.2 LANGUAGES

2.2.1 SQL

SQL, or Structured Query Language, is the go-to language for communicating with relational databases. Imagine it as a set of instructions for a giant filing cabinet. SQL empowers you to organize and manage information stored in tables with rows and columns. Need to find something specific? SQL lets you search and filter the database with ease. You can even add, edit, and delete data entries to keep everything organized and current. SQL's ability to control access ensures only authorized users can tinker with valuable information. As a widespread standard across database systems, mastering SQL is a valuable skill for anyone who works with data.

2.2.2 PYTHON

Python is a versatile and beginner-friendly programming language. With its clear and concise syntax, it's easy to learn and write code that reads almost like plain English. This makes it a favorite for beginners. But its power extends beyond introductory lessons. Python boasts a vast library of tools for web development, data analysis, machine learning, and more, making it a valuable asset for experienced programmers as well. Plus, as a free and open-source language, Python offers a supportive community and endless resources for learning and development.

CH 3. REQUIREMENT AND ANALYSIS

3.1 REQUIREMENTS SPECIFICATION

3.1.1 USER REQUIREMENTS

The art gallery system allows users to explore and manage artwork through search, view, and management functionalities.

3.1.2 SYSTEM REQUIREMENTS

There should be a database backup of the library management system. Operating system should be WindowsXP or a higher version of windows.

3.2 HARDWARE AND SOFTWARE REQUIREMENTS

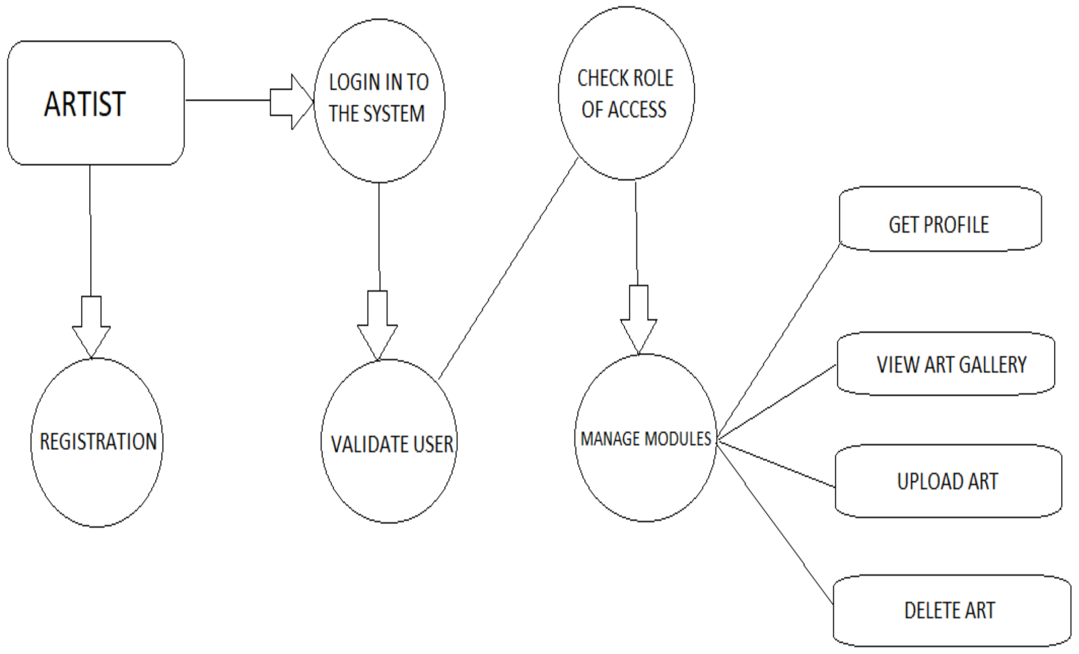
3.2.1 SOFTWARE REQUIREMENTS

- Operating System Windows 10
- Front End : PYTHON
- Back End : SQL

3.2.2 HARDWARE REQUIREMENTS

- Desktop PC or a Laptop
- Printer
- Operating System – Windows 10
- Intel® CoreTM i3-6006U CPU @ 2.00GHz
- 4.00 GB RAM
- 64-bit operating system, x64 based processor
- 1024 x 768 monitor resolution
- Keyboard and Mouse

3.3 ARCHITECTURE



Artist Login

- Login to the System: This section allows artists to enter their login credentials to access the system.
- Registration: New artists can register themselves on the platform.

Manage User

- Check Role of Access: This section allows users to determine their access level within the system, which could be artist, administrator or gallery owner (roles not explicitly shown in the diagram).
- Validate User: This functionality might be for system administrators to validate new user registrations.

Manage Modules

- Get Profile: This allows artists to access and update their profile information.
- View Art Gallery: This section provides artists with an overview of the art gallery, possibly showcasing other artists and their work.

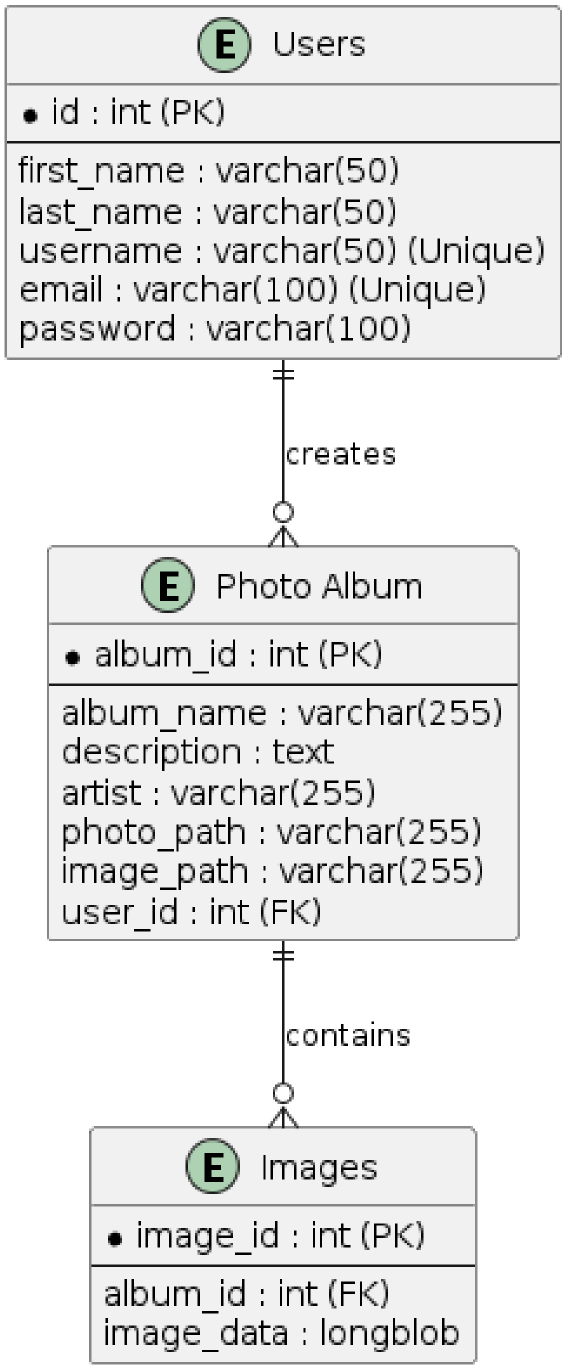
Upload Art

- Upload Art: This function allows artists to upload their artwork to the online gallery.
- Delete Art: This function allows artists to delete their artwork from the online gallery.

Order Management (separate box)

- Order Management: This section likely refers to the functionality for users to purchase artwork, though it's not explicitly connected to the artist functionalities in the diagram

3.4 ER DIAGRAM



Users (Entity)

- **id**: (Primary Key) Unique identifier for the user (int)
- **first_name** : User's first name (varchar(50))
- **last_name** : User's last name (varchar(50))
- **username** : Username for login (varchar(50)) (Unique)
- **email** : User's email address (varchar(100)) (Unique)
- **password** : User's password (varchar(100))

Photo Album (Entity)

- **album_id** : (Primary Key) Unique identifier for the album (int)
- **album_name** : Name of the album (varchar(255))
- **description** : Description of the album (text)
- **artist** : Name of the artist who created the album (varchar(255))
- **user_id** : Foreign Key referencing the user who created the album (int)

Images (Entity)

- **image_id** : (Primary Key) Unique identifier for the image (int)
- **album_id** : Unique identifier for the album (int)
- **image_data** : data of the image for each album id (longblob)

RELATIONSHIP

User and Photo Album Relationship:

- This is a one-to-many relationship, where one user can create multiple photo albums, but each photo album is associated with only one user.
- The **user_id** column in the **photo_album** entity acts as a foreign key that references the **id** column in the **user** entity.
- This relationship allows you to track which user created each photo album.

Photo Album and Images Relationship:

- This is a one-to-many relationship, where one photo album can have multiple images associated with it, but each image belongs to only one photo album.
- The **album_id** column in the **images** entity acts as a foreign key that references the **album_id** column in the **photo_album** entity.
- This relationship allows you to store multiple images for each photo album, and each image is associated with a specific album.

3.5 NORMALIZATION

FIRST NORMAL FORM (1NF):

- The entities are already in the first normal form since they don't have any multi-valued attributes or composite primary keys.

SECOND NORMAL FORM (2NF):

- The entities are already in the second normal form since they don't have any partial dependencies on the primary keys.

THIRD NORMAL FORM (3NF):

- The **photo_album** entity is not in the third normal form because the **artist** column is a non-key attribute that does not depend on the full primary key (**album_id**). This can lead to update anomalies.

To achieve the third normal form, we need to separate the **artist** column from the **photo_album** entity and create a new entity called **Artist**.

Artist (Entity)

- **artist_id** : (Primary Key) Unique identifier for the artist (int)
- **artist_name** : Name of the artist (varchar(255))

Now, the photo_album entity becomes,

Photo Album (Entity)

- **album_id** : (Primary Key) Unique identifier for the album (int)
- **album_name**: Name of the album (varchar(255))
- **description**: Description of the album (text)
- **user_id** : Foreign Key referencing the user who created the album (int)
- **artist_id** : Foreign Key referencing the artist associated with the album (int)

Now, the **photo_album** entity is in the third normal form, and the relationship between **photo_album** and **Artist** is established through the **artist_id** foreign key.

CH 4. PROGRAM CODE

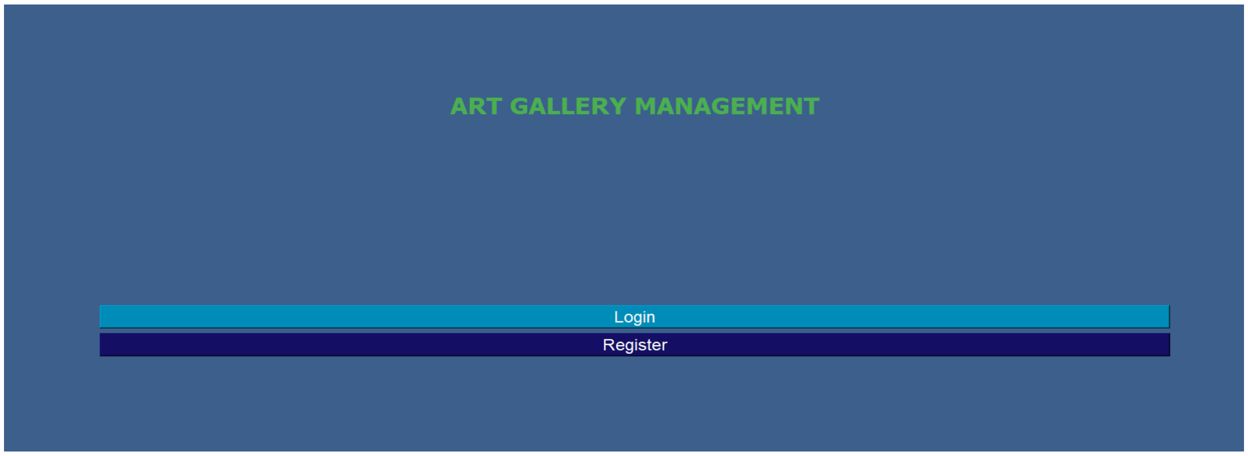
```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton,
QLabel, QLineEdit, QDialog, QMessageBox, QListWidget, QFileDialog, QHBoxLayout,
QScrollArea, QGridLayout
from PyQt5.QtGui import QFont, QPixmap
from PyQt5.QtCore import Qt
import mysql.connector

class UserListWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle('User List')
        self.resize(400, 300)
        layout = QVBoxLayout()
        label = QLabel('User List')
        label.setFont(QFont('Arial', 16))
        layout.addWidget(label)
        self.list_widget = QListWidget()
        layout.addWidget(self.list_widget)
        self.setLayout(layout)

    def loadUsers(self):
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )

            cursor = conn.cursor()
            cursor.execute("SELECT * FROM users")
            rows = cursor.fetchall()
            self.list_widget.clear()
            for row in rows:
                user_info = f"{row[0]} - {row[1]} {row[2]} ({row[3]})"
                self.list_widget.addItem(user_info)
            conn.close()
        except mysql.connector.Error as e:
            print(f'Failed to retrieve users: {e}')

class ArtGalleryManagement(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Art Gallery Management')
        self.resize(600, 400)
        self.setStyleSheet('background-color: #40618E;')
        layout = QVBoxLayout()
        label = QLabel('ART GALLERY MANAGEMENT')
        label.setStyleSheet('color: #4CAF50; font-size: 34px; font-weight: bold;')
        label.setAlignment(Qt.AlignCenter)
        layout.setContentsMargins(250, 20, 250, 300)
        login_button = QPushButton('Login')
        login_button.setStyleSheet('background-color: #008CBA; color: white;')
        login_button.setFont(QFont('Arial', 14))
        login_button.clicked.connect(self.showLogin)
        registration_button = QPushButton('Register')
        registration_button.setStyleSheet('background-color: #171065; color: white;')
        registration_button.setFont(QFont('Arial', 14))
        registration_button.clicked.connect(self.showRegistration)
        layout.addWidget(label)
        layout.addWidget(login_button)
        layout.addWidget(registration_button)
        self.setLayout(layout)

    def showLogin(self):
        login_form = LoginForm(self)
        if login_form.exec_() == QDialog.Accepted:
            username = login_form.username_edit.text()
            self.showHomePage(username)

    def showRegistration(self):
        registration_form = RegistrationForm(self)
        registration_form.exec_()

    def showHomePage(self, username):
        self.home_page = HomePage(username)
        self.home_page.show()

class RegistrationForm(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle('Registration Form')
        self.first_name_edit = QLineEdit()
        self.last_name_edit = QLineEdit()
        self.username_edit = QLineEdit()
        self.email_edit = QLineEdit()
        self.password_edit = QLineEdit()
        self.confirm_password_edit = QLineEdit()
        self.first_name_edit.setStyleSheet('color: white;')
        self.last_name_edit.setStyleSheet('color: white;')
        self.username_edit.setStyleSheet('color: white;')
        self.email_edit.setStyleSheet('color: white;')
        self.password_edit.setStyleSheet('color: white;')
        self.confirm_password_edit.setStyleSheet('color: white;')
        self.password_edit.setEchoMode(QLineEdit.Password)
        self.confirm_password_edit.setEchoMode(QLineEdit.Password)
        layout = QVBoxLayout()
        self.addFormField(layout, 'First Name:', self.first_name_edit)
        self.addFormField(layout, 'Last Name:', self.last_name_edit)
        self.addFormField(layout, 'Username:', self.username_edit)
        self.addFormField(layout, 'Email:', self.email_edit)
        self.addFormField(layout, 'Password:', self.password_edit)
        self.addFormField(layout, 'Confirm Password:', self.confirm_password_edit)
        self.setLayout(layout)
        self.ok_button = QPushButton('OK')
        self.cancel_button = QPushButton('Cancel')
        self.ok_button.setStyleSheet('color: white;')
        self.cancel_button.setStyleSheet('color: white;')
        layout.addWidget(self.ok_button)
        layout.addWidget(self.cancel_button)
        self.ok_button.clicked.connect(self.register)
        self.cancel_button.clicked.connect(self.reject)

    def addFormField(self, layout, label_text, edit_widget):
        label = QLabel(label_text)
        label.setStyleSheet('color: white;')
        layout.addWidget(label)
        layout.addWidget(edit_widget)

    def register(self):
        first_name = self.first_name_edit.text()
        last_name = self.last_name_edit.text()
        username = self.username_edit.text()
        email = self.email_edit.text()
        password = self.password_edit.text()
        confirm_password = self.confirm_password_edit.text()

        if not (first_name and last_name and username and email and password and
confirm_password):
            QMessageBox.critical(self, 'Error', 'All fields are required!')
            return

        if password != confirm_password:
            QMessageBox.critical(self, 'Error', 'Passwords do not match!')
            return

        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )

            cursor = conn.cursor()
            cursor.execute("INSERT INTO users (first_name, last_name, username, email,
password) VALUES (%s, %s, %s, %s, %s)",
                           (first_name, last_name, username, email, password))
            conn.commit()
            conn.close()
            QMessageBox.information(self, 'Success', 'Registration Successful!')
            self.accept()
        except mysql.connector.Error as e:
            QMessageBox.critical(self, 'Error', f'Failed to register user: {e}')

class LoginForm(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle('Login')
        self.username_edit = QLineEdit()
        self.password_edit = QLineEdit()
        self.username_edit.setStyleSheet('color: white;')
        self.password_edit.setStyleSheet('color: white;')
        self.password_edit.setEchoMode(QLineEdit.Password)
        layout = QVBoxLayout()
        self.addFormField(layout, 'Username:', self.username_edit)
        self.addFormField(layout, 'Password:', self.password_edit)
        self.setLayout(layout)
        self.ok_button = QPushButton('Login')
        self.cancel_button = QPushButton('Cancel')
        self.ok_button.setStyleSheet('color: white;')
        self.cancel_button.setStyleSheet('color: white;')
        layout.addWidget(self.ok_button)
        layout.addWidget(self.cancel_button)
        self.ok_button.clicked.connect(self.login)
        self.cancel_button.clicked.connect(self.reject)

    def addFormField(self, layout, label_text, edit_widget):
        label = QLabel(label_text)
        label.setStyleSheet('color: white;')
        layout.addWidget(label)
        layout.addWidget(edit_widget)

    def login(self):
        username = self.username_edit.text()
        password = self.password_edit.text()

        if not (username and password):
            QMessageBox.critical(self, 'Error', 'Username and password are required!')
            return
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )

            cursor = conn.cursor()
            cursor.execute("SELECT * FROM users WHERE username = %s AND password
= %s", (username, password))
            row = cursor.fetchone()

            if row:
                QMessageBox.information(self, 'Success', 'Login Successful!')
                self.accept()
            else:
                QMessageBox.critical(self, 'Error', 'Invalid username or password!')

            conn.close()
        except mysql.connector.Error as e:
            QMessageBox.critical(self, 'Error', f'Failed to authenticate user: {e}')

class HomePage(QWidget):
    def __init__(self, username):
        super().__init__()
        self.setWindowTitle('Home Page')
        self.resize(800, 600)
        self.username = username
        self.header_layout = QVBoxLayout()
        self.username_label = QLabel(f"Logged in as: {self.username}")
        self.header_layout.addWidget(self.username_label)
        self.add_album_button = QPushButton('Add Album')
        self.add_album_button.clicked.connect(self.addAlbum)
        self.header_layout.addWidget(self.add_album_button)
        self.albums_layout = QVBoxLayout()
        self.header_layout.addLayout(self.albums_layout)
        self.loadAlbums()
        self.setLayout(self.header_layout)
```

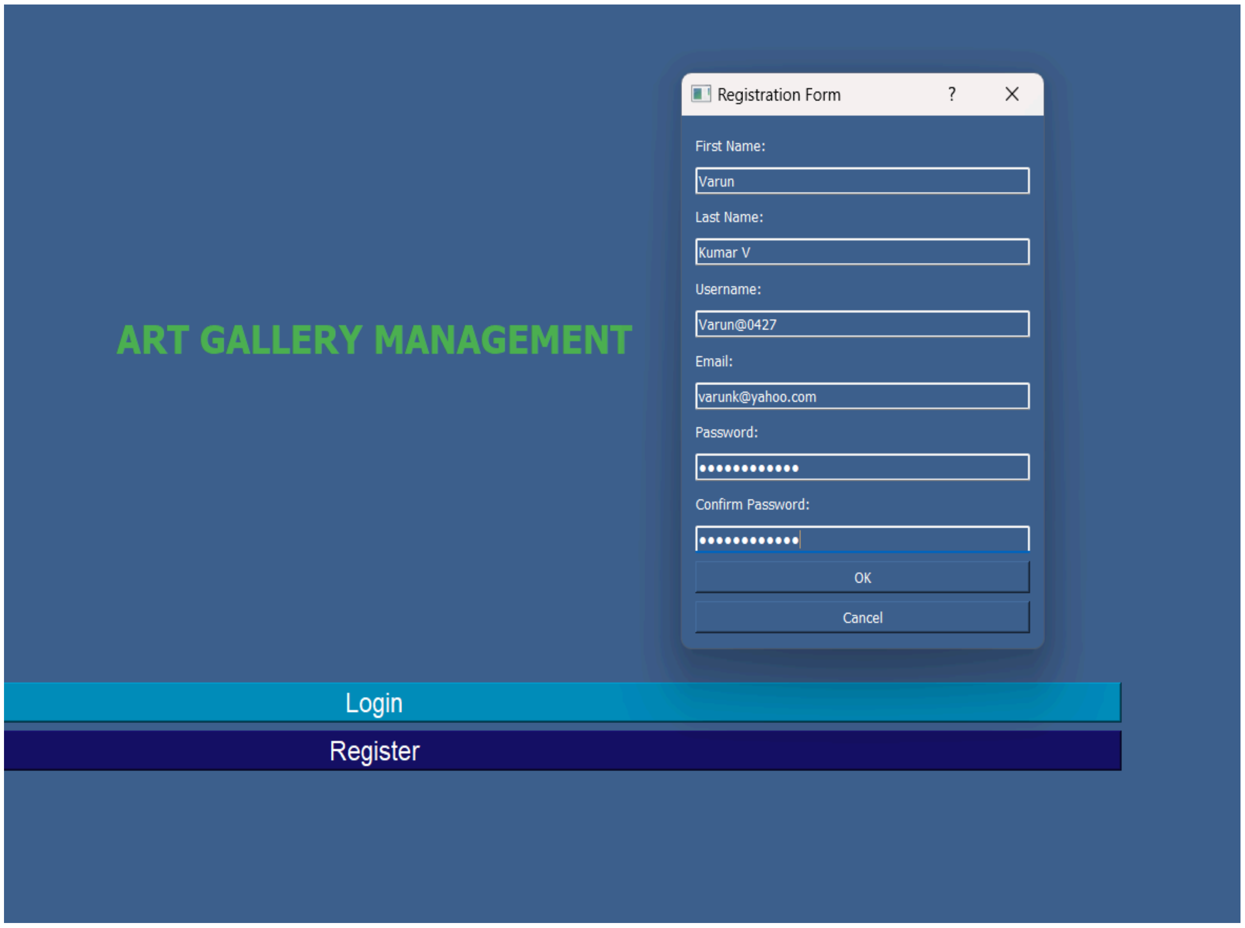
```
def loadAlbums(self):
    try:
        conn = mysql.connector.connect(
            host="localhost",
            user="gallery_user",
            password="varun2711",
            database="art_gallery"
        )
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM photo_album")
        albums = cursor.fetchall()
        for i in reversed(range(self.albums_layout.count())):
            widget = self.albums_layout.itemAt(i).widget()
            if widget:
                widget.deleteLater()
        for album in albums:
            album_widget = self.createAlbumWidget(album)
            self.albums_layout.addWidget(album_widget)

        conn.close()
    except mysql.connector.Error as e:
        print(f'Failed to load albums: {e}')

def createAlbumWidget(self, album):
    album_widget = QWidget()
    layout = QVBoxLayout()
    album_name_label = QLabel(f"Album Name: {album[1]}")
    description_label = QLabel(f"Description: {album[2]}")
    artist_label = QLabel(f"Artist: {album[3]}")

    layout.addWidget(album_name_label)
    layout.addWidget(description_label)
    layout.addWidget(artist_label)
    add_images_button = QPushButton('Add Images')
    add_images_button.clicked.connect(lambda: self.addImages(album[0]))
    layout.addWidget(add_images_button)
    scroll_area = QScrollArea()
    scroll_area.setWidgetResizable(True)
    scroll_widget = QWidget()
    grid_layout = QGridLayout()
    scroll_widget.setLayout(grid_layout)
```

```
        scroll_area.setWidget(scroll_widget)
        layout.addWidget(scroll_area)

        album_widget.scroll_area = scroll_area
        album_widget.grid_layout = grid_layout
        self.loadImages(album[0], grid_layout)
        edit_button = QPushButton('Edit')
        edit_button.clicked.connect(lambda: self.editAlbum(album[0]))

        delete_button = QPushButton('Delete')
        delete_button.clicked.connect(lambda: self.deleteAlbum(album[0]))

        button_layout = QHBoxLayout()
        button_layout.addWidget(edit_button)
        button_layout.addWidget(delete_button)

        layout.addLayout(button_layout)

        album_widget.setLayout(layout)
        return album_widget

    def get_images_for_album(album_id):
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )
            cursor = conn.cursor()
            query = "SELECT image_data FROM images WHERE album_id = %s"
            cursor.execute(query, (album_id,))
            images = cursor.fetchall()

            conn.close()
            return images

        except mysql.connector.Error as e:
            print(f'Failed to retrieve images: {e}')
            return None
#Example Usage
```

```
    album_id = 1
    images = get_images_for_album(album_id)
    if images:
        for i, image in enumerate(images):
            with open(f'image_{i}.jpg', 'wb') as f:
                f.write(image[0])

        def addImages(self, album_id):
            file_paths, _ = QFileDialog.getOpenFileNames(self, 'Select Images', '', 'Image
Files (*.png *.jpg *.bmp)')

            if file_paths:
                try:
                    conn = mysql.connector.connect(
                        host="localhost",
                        user="gallery_user",
                        password="varun2711",
                        database="art_gallery"
                    )
                    cursor = conn.cursor()

                    for file_path in file_paths:
                        with open(file_path, 'rb') as file:
                            image_data = file.read()
                            cursor.execute("INSERT INTO images (album_id, image_data)
VALUES (%s, %s)", (album_id, image_data))
                    conn.commit()
                    conn.close()
                    for album_widget in self.findChildren(QWidget):
                        if hasattr(album_widget, 'scroll_area') and hasattr(album_widget,
'grid_layout'):
                            self.loadImages(album_id, album_widget.grid_layout)

                except mysql.connector.Error as e:
                    print(f'Failed to add images: {e}')


    def loadImages(self, album_id, grid_layout):
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )
            cursor = conn.cursor()
            cursor.execute("SELECT image_data FROM images WHERE album_id = %s",
(album_id,))
            images = cursor.fetchall()
            for i in reversed(range(grid_layout.count())):
                item = grid_layout.itemAt(i)
                if item:
                    widget = item.widget()
                    if widget:
                        widget.deleteLater()
                    grid_layout.removeItem(item)
            row, col = 0, 0
            for image in images:
                image_label = QLabel()
                pixmap = QPixmap()
                pixmap.loadFromData(image[0])
                scaled_pixmap = pixmap.scaled(200, 200, Qt.KeepAspectRatio)
                image_label.setPixmap(scaled_pixmap)
                grid_layout.addWidget(image_label, row, col)
                col += 1
                if col >= 3:
                    row += 1
                    col = 0

            conn.close()
        except mysql.connector.Error as e:
            print(f'Failed to load images: {e}')

    def addAlbum(self):
        dialog = AddAlbumDialog(self)
        if dialog.exec_() == QDialog.Accepted:
            album_data = dialog.getAlbumData()
            try:
                conn = mysql.connector.connect(
                    host="localhost",
                    user="gallery_user",
                    password="varun2711",
                    database="art_gallery"
                )
                cursor = conn.cursor()
                cursor.execute("INSERT INTO photo_album (album_name, description,
artist) VALUES (%s, %s, %s)",
                           (album_data['album_name'], album_data['description'],
album_data['artist']))
                conn.commit()
                conn.close()
                self.loadAlbums()
            except mysql.connector.Error as e:
                print(f'Failed to add album: {e}')

    def editAlbum(self, album_id):
        dialog = EditAlbumDialog(self, album_id)
        if dialog.exec_() == QDialog.Accepted:
            album_data = dialog.getAlbumData()
            try:
                conn = mysql.connector.connect(
                    host="localhost",
                    user="gallery_user",
                    password="varun2711",
                    database="art_gallery"
                )
                cursor = conn.cursor()
                cursor.execute("UPDATE photo_album SET album_name = %s, description
= %s, artist = %s WHERE album_id = %s",
                               (album_data['album_name'], album_data['description'],
album_data['artist'], album_id))

                conn.commit()
                conn.close()
                self.loadAlbums()
            except mysql.connector.Error as e:
                print(f'Failed to edit album: {e}')

    def deleteAlbum(self, album_id):
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
    password="varun2711",
                database="art_gallery"
            )
            cursor = conn.cursor()
            cursor.execute("DELETE FROM images WHERE album_id = %s", (album_id,))
            cursor.execute("DELETE FROM photo_album WHERE album_id = %s",
(album_id,))
            conn.commit()
            conn.close()
            self.loadAlbums()
        except mysql.connector.Error as e:
            print(f'Failed to delete album: {e}')

class AddAlbumDialog(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle('Add Album')
        self.album_name_edit = QLineEdit()
        self.description_edit = QLineEdit()
        self.artist_edit = QLineEdit()
        layout = QVBoxLayout()
        self.addFormField(layout, 'Album Name:', self.album_name_edit)
        self.addFormField(layout, 'Description:', self.description_edit)
        self.addFormField(layout, 'Artist:', self.artist_edit)
        self.setLayout(layout)
        self.ok_button = QPushButton('OK')
        self.cancel_button = QPushButton('Cancel')
        layout.addWidget(self.ok_button)
        layout.addWidget(self.cancel_button)
        self.ok_button.clicked.connect(self.accept)
        self.cancel_button.clicked.connect(self.reject)

    def addFormField(self, layout, label_text, edit_widget):
        label = QLabel(label_text)
        layout.addWidget(label)
        layout.addWidget(edit_widget)

    def getAlbumData(self):
        return {
            'album_name': self.album_name_edit.text(),
         'description': self.description_edit.text(),
            'artist': self.artist_edit.text()
        }

class EditAlbumDialog(AddAlbumDialog):
    def __init__(self, parent=None, album_id=None):
        super().__init__(parent)
        self.setWindowTitle('Edit Album')
        self.album_id = album_id
        self.loadAlbumData()

    def loadAlbumData(self):
        try:
            conn = mysql.connector.connect(
                host="localhost",
                user="gallery_user",
                password="varun2711",
                database="art_gallery"
            )
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM photo_album WHERE album_id = %s",
(self.album_id,))
            album = cursor.fetchone()
            self.album_name_edit.setText(album[1])
            self.description_edit.setText(album[2])
            self.artist_edit.setText(album[3])

            conn.close()
        except mysql.connector.Error as e:
            print(f'Failed to load album data: {e}')

app = QApplication(sys.argv)
main_window = ArtGalleryManagement()
main_window.show()
sys.exit(app.exec_())
```

CH 5. RESULTS





First Name:

Varun

Last Name:

Kumar V

Username:

Varun@0427

Email:

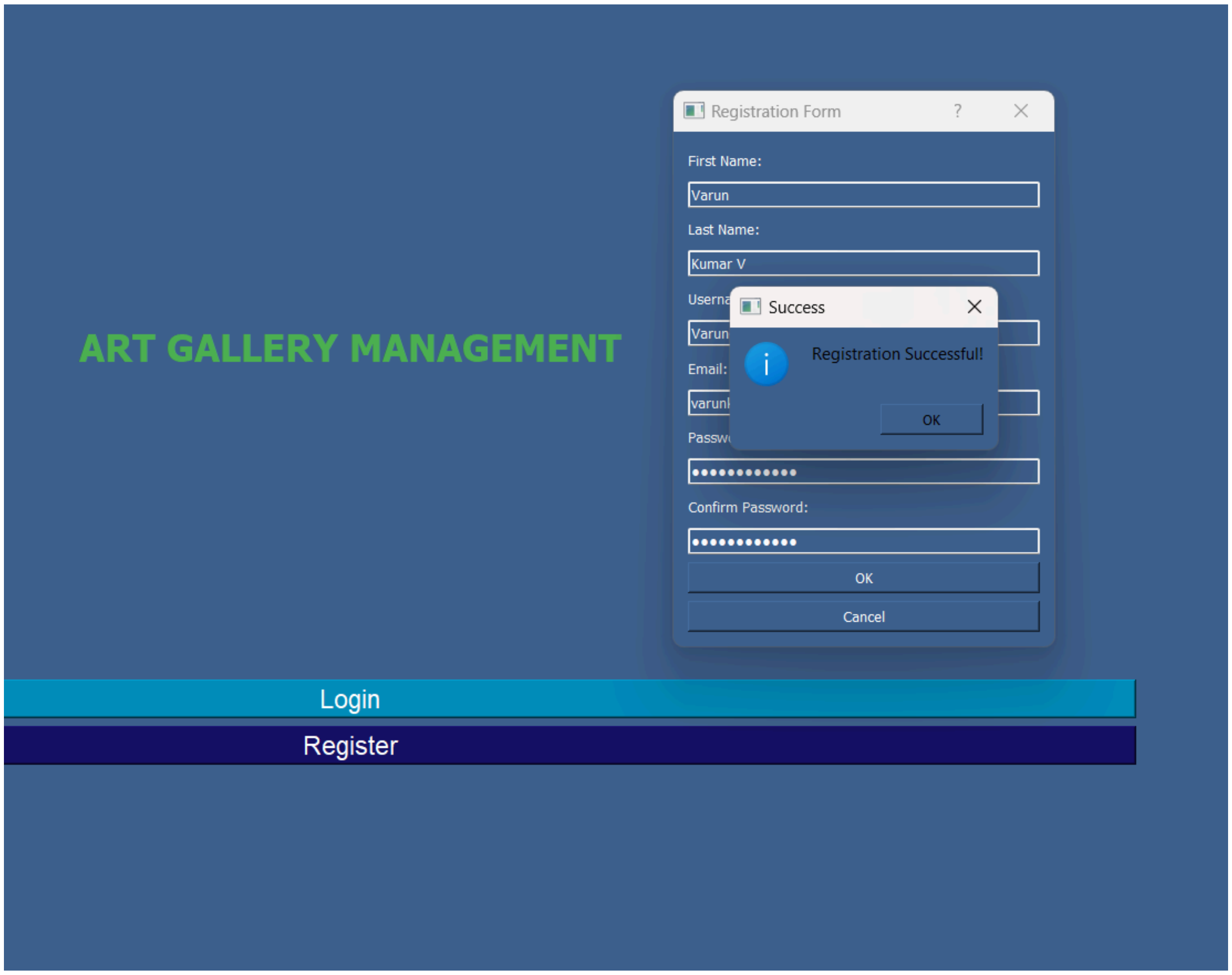
varunk@yahoo.com

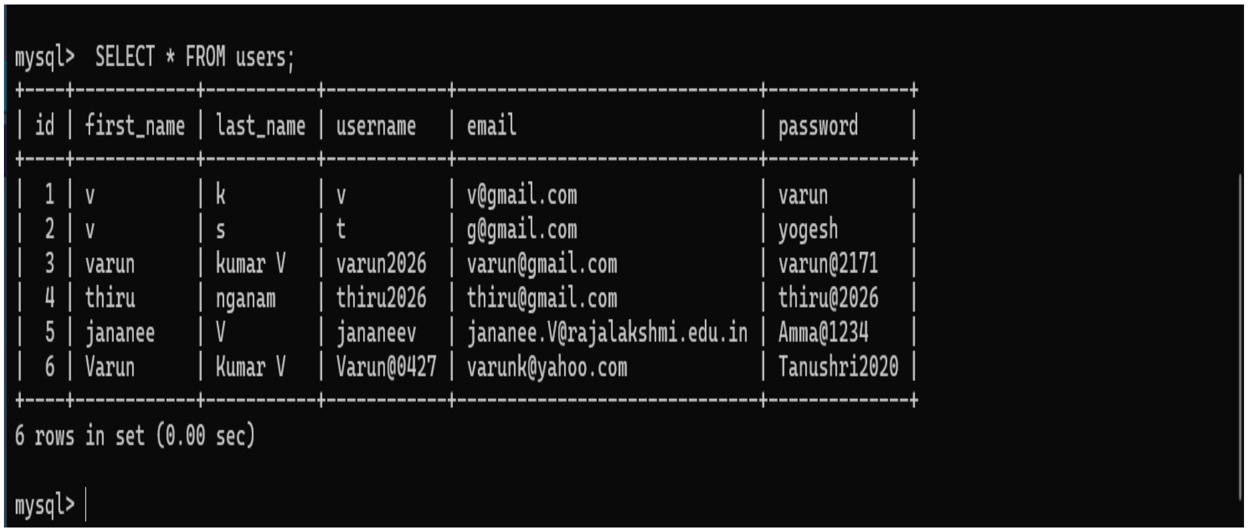
Password:

••••••••••••

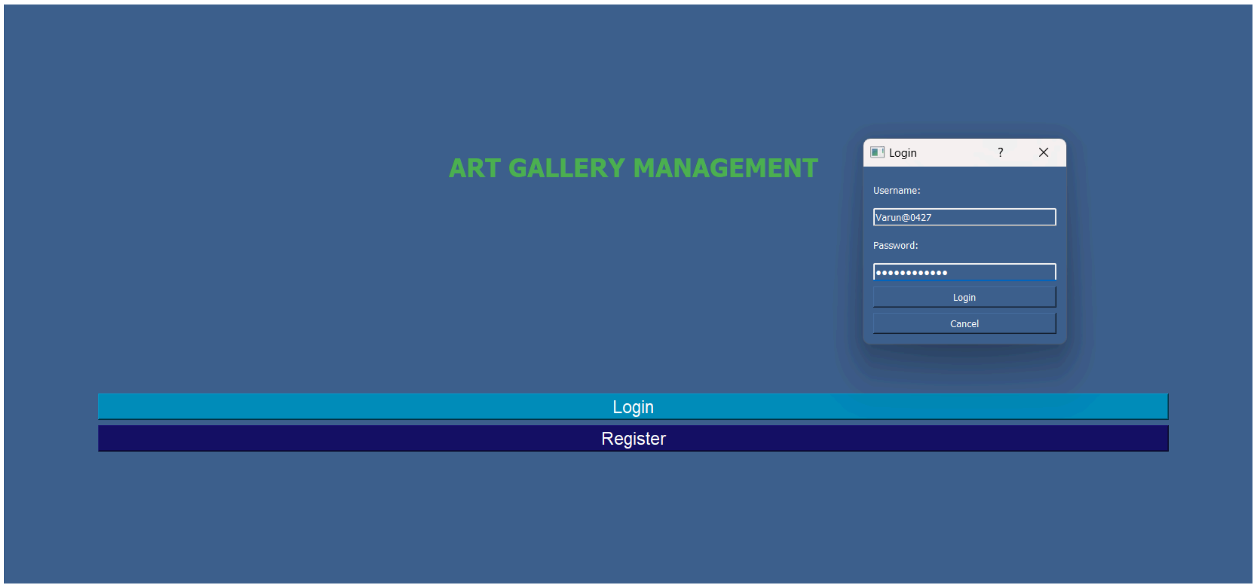
Confirm Password:

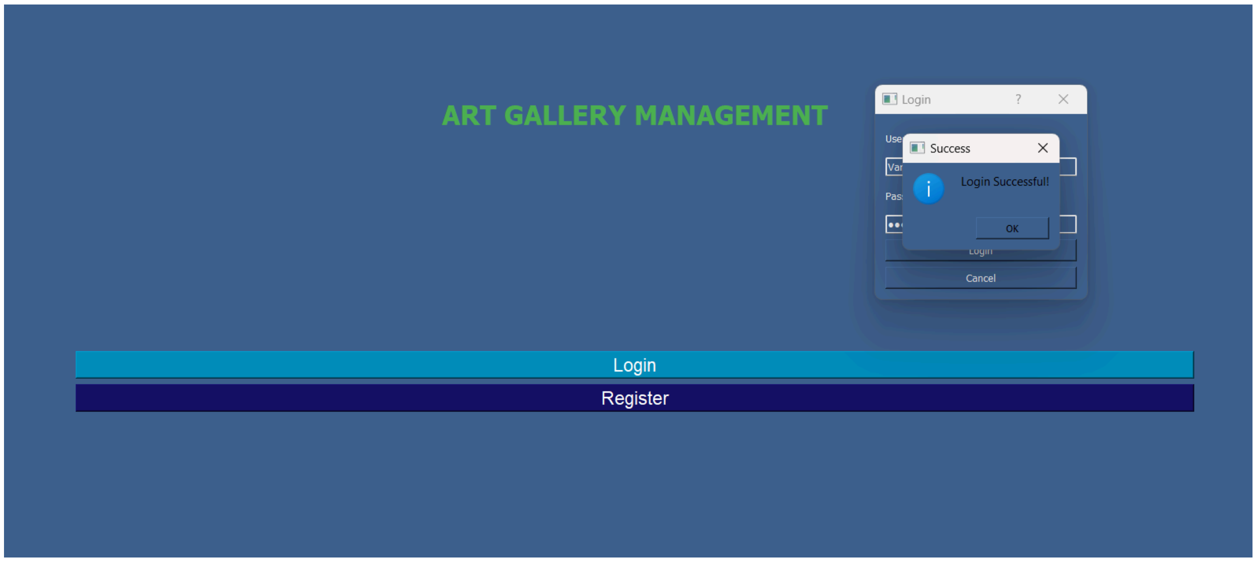
••••••••••••

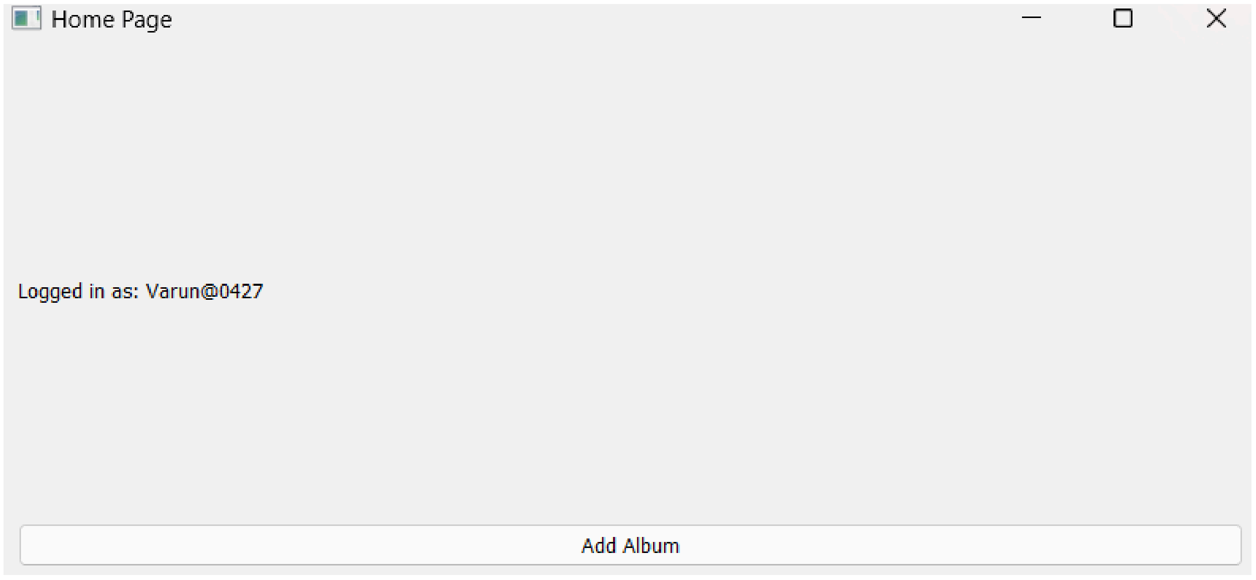




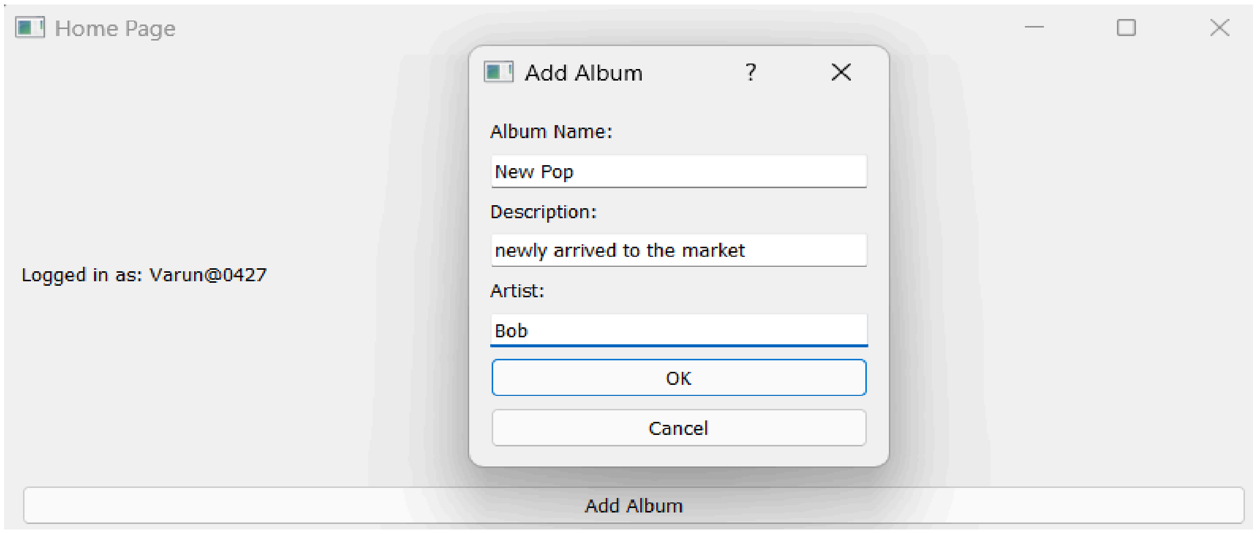
```
mysql>  SELECT * FROM users;
+----+------------+-----------+------------+------------------------------+---------------+
| id | first_name | last_name | username   | email                        | password      |
+----+------------+-----------+------------+------------------------------+---------------+
|  1 | v          | k         | v          | v@gmail.com                  | varun         |
|  2 | v          | s         | t          | g@gmail.com                  | yogesh        |
|  3 | varun      | kumar V   | varun2026  | varun@gmail.com              | varun@2171    |
|  4 | thiru      | nganam    | thiru2026  | thiru@gmail.com              | thiru@2026    |
|  5 | jananee    | V         | jananeev   | jananee.V@rajalakshmi.edu.in | Amma@1234     |
|  6 | Varun      | Kumar V   | Varun@0427 | varunk@yahoo.com             | Tanushri2020  |
+----+------------+-----------+------------+------------------------------+---------------+
6 rows in set (0.00 sec)

mysql>
```







Logged in as: Varun@0427



Logged in as: Varun@0427

Album Name:

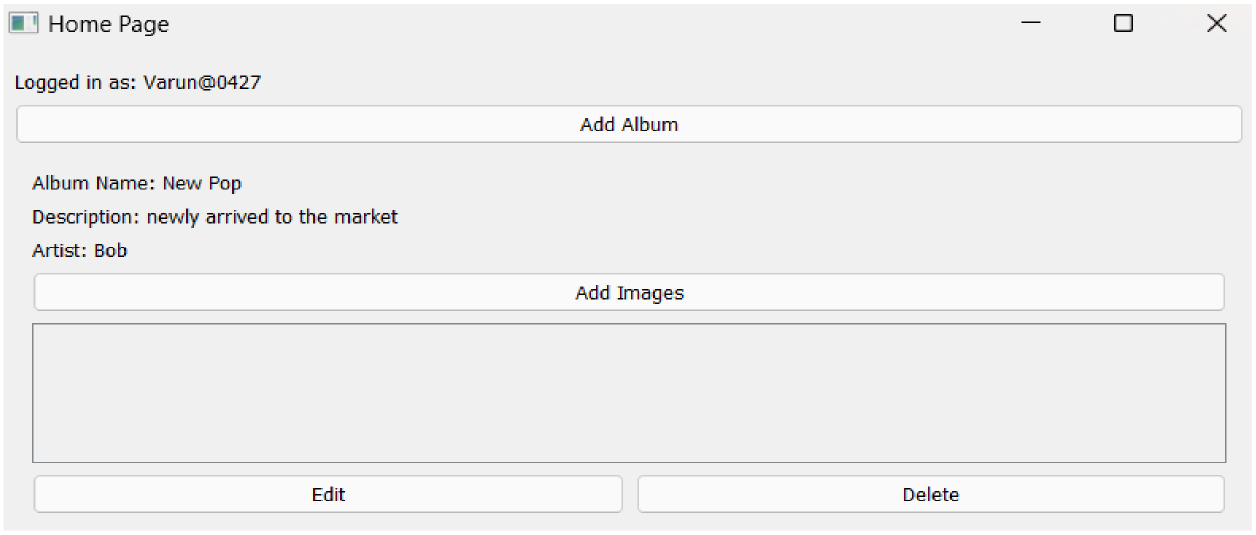
New Pop

Description:

newly arrived to the market

Artist:

Bob

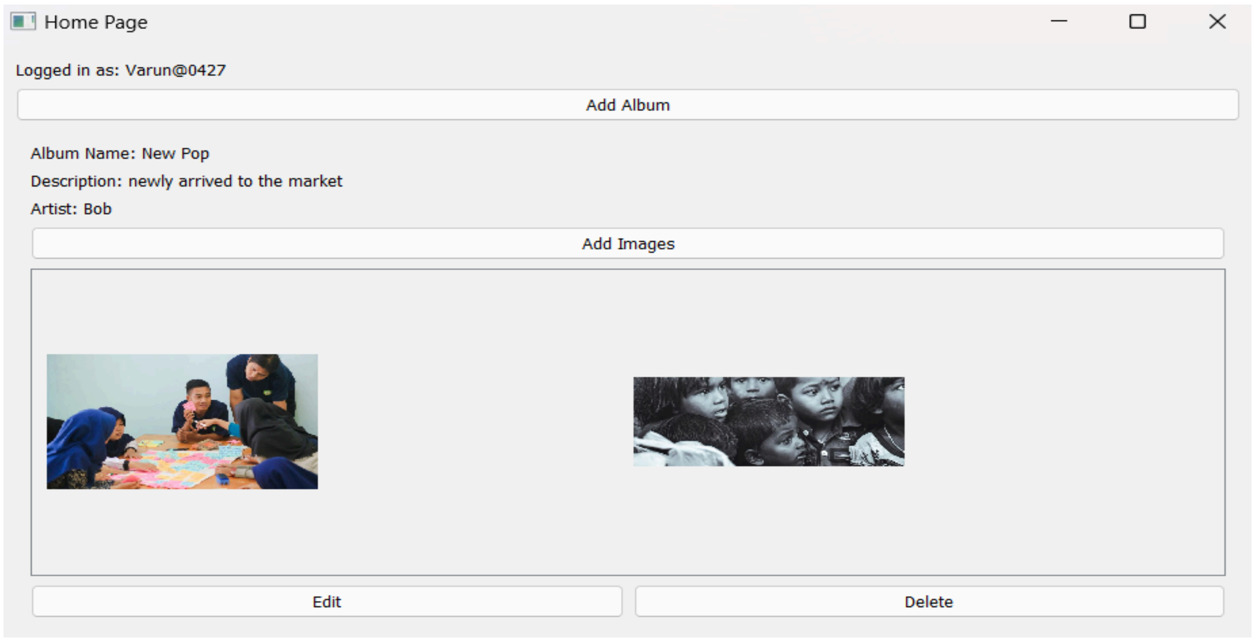


Logged in as: Varun@0427

Album Name: New Pop

Description: newly arrived to the market

Artist: Bob

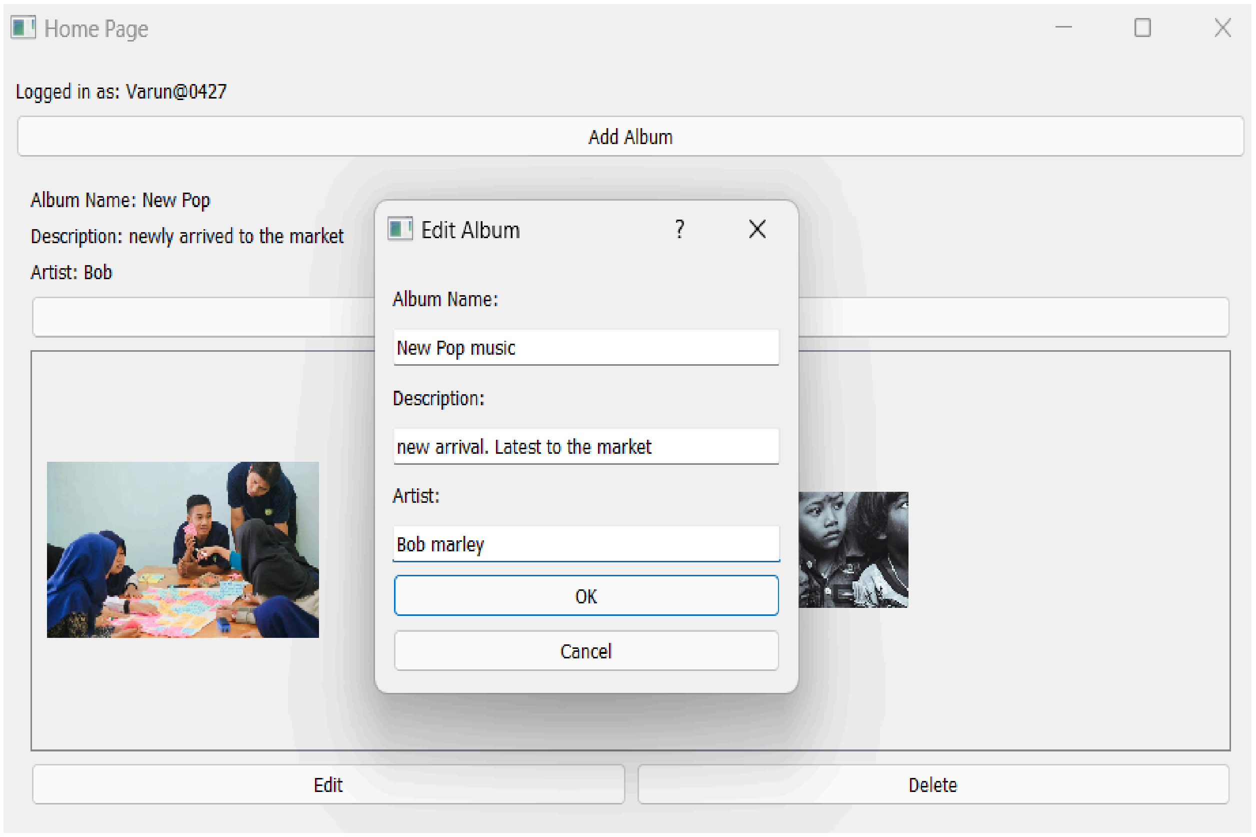


Logged in as: Varun@0427

Album Name: New Pop

Description: newly arrived to the market

Artist: Bob



Home Page

Logged in as: Varun@0427

Album Name: New Pop

Description: newly arrived to the market

Artist: Bob

Edit Album

Album Name:

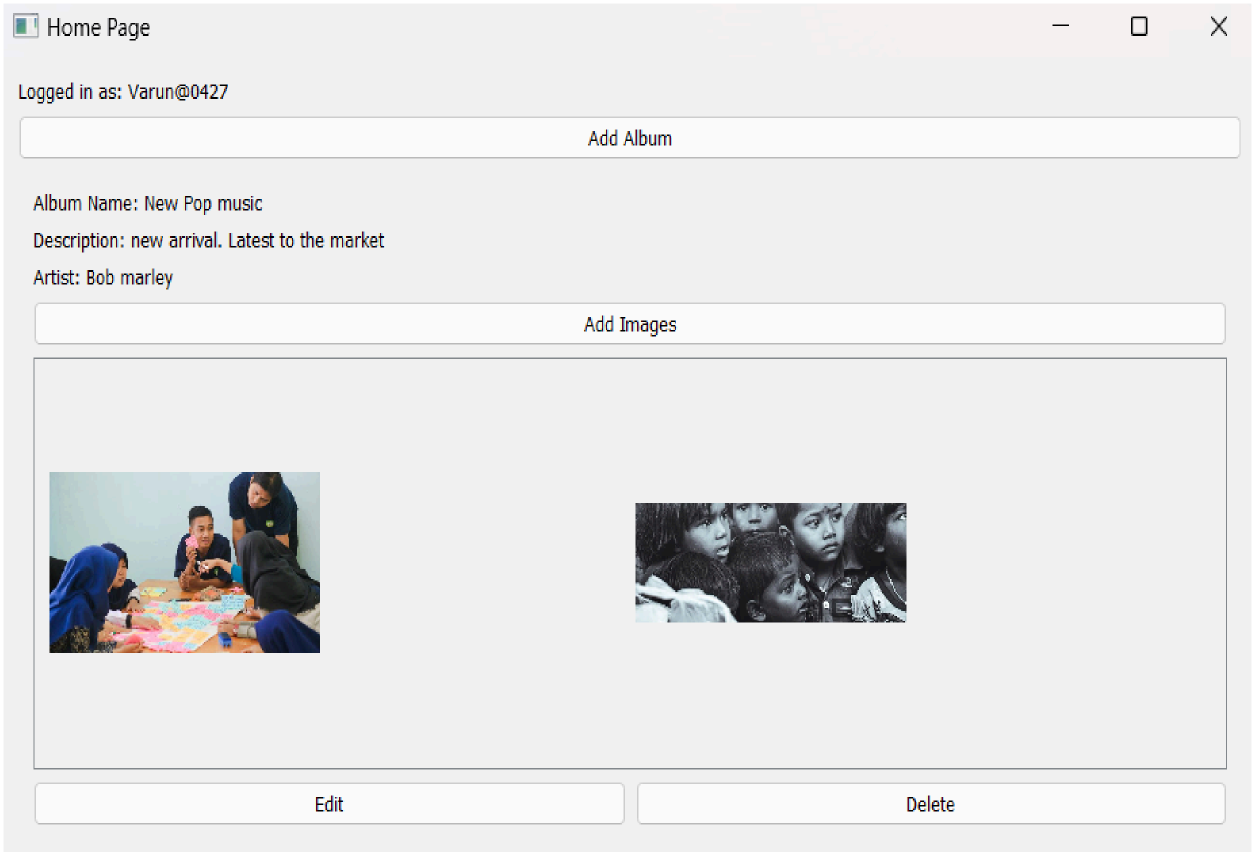
New Pop music

Description:

new arrival. Latest to the market

Artist:

Bob marley



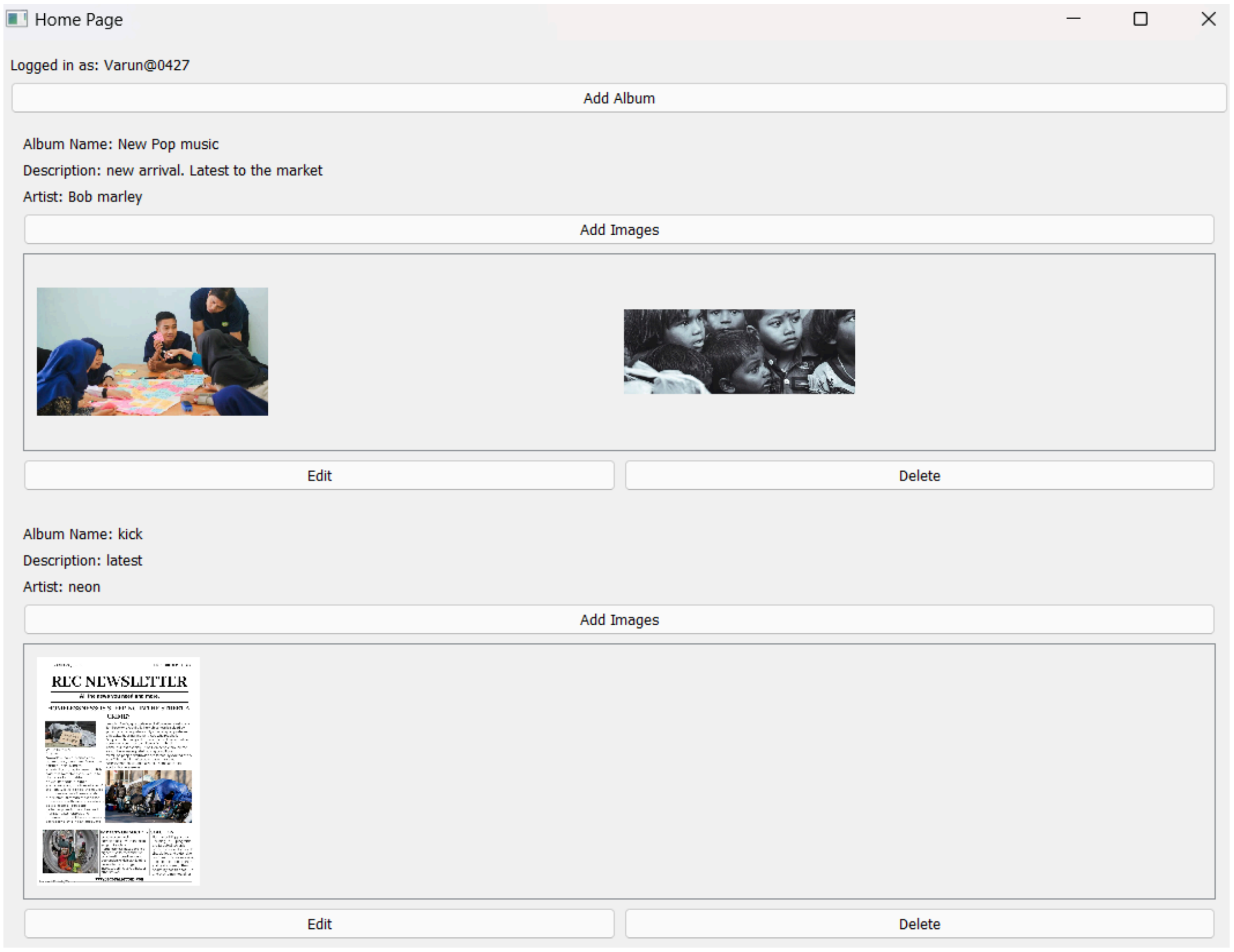
Home Page

Logged in as: Varun@0427

Album Name: New Pop music

Description: new arrival. Latest to the market

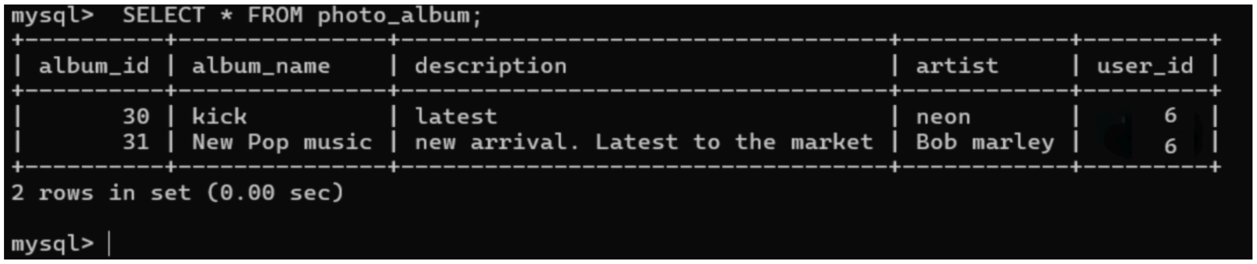
Artist: Bob marley



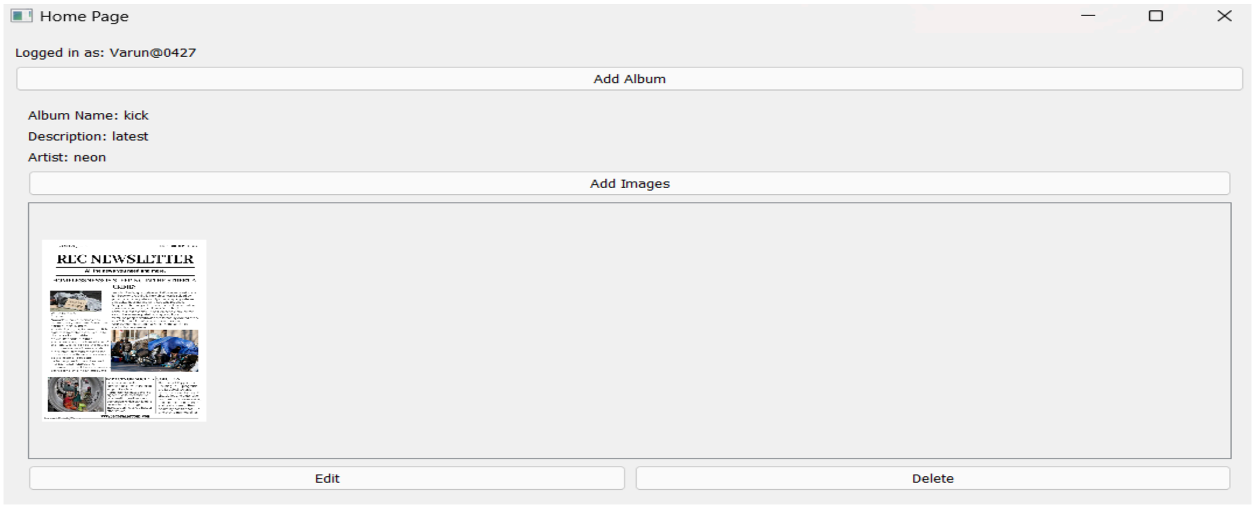
Logged in as: Varun@0427

Album Name: New Pop music
Description: new arrival. Latest to the market
Artist: Bob marley

Album Name: kick
Description: latest
Artist: neon



```
mysql>  SELECT * FROM photo_album;
+----------+----------------+------------------------------------+------------+---------+
| album_id | album_name     | description                        | artist     | user_id |
+----------+----------------+------------------------------------+------------+---------+
|       30 | kick           | latest                             | neon       |       6 |
|       31 | New Pop music  | new arrival. Latest to the market  | Bob marley |       6 |
+----------+----------------+------------------------------------+------------+---------+
2 rows in set (0.00 sec)

mysql> |
```



Logged in as: Varun@0427

Album Name: kick
Description: latest
Artist: neon

CH 6. CONCLUSION

The Art Gallery Management application successfully demonstrates the integration of Python, PyQt5, and MySQL. It offers a user-friendly interface for managing photo albums and associated images.

The application implements user authentication, allowing secure access to authorized users. Album management features enable users to create, view, edit, and delete photo albums. Image management functionality allows users to add and display images within albums.

The project leverages MySQL for persistent storage of user data, albums, and images. The code follows a modular structure, separating concerns into individual classes and methods. Error handling and input validation ensure data integrity and provide feedback to users.

Potential enhancements could include advanced search, album sharing, and image editing capabilities. Overall, the project demonstrates proficiency in Python programming, GUI development, and database integration.

CH 7. REFERENCES

This section lists the sources, references, and literature consulted during the research, design, and development phases of the FMS project. It includes academic papers, technical documentation, online resources, and industry best practices relevant to Python (PyQt), MySQL, and Art Gallery Management System.

Model-View-Controller (MVC) Pattern in PyQt:
https://www.pythoncentral.io/series/python-qt-creating-model-views-qt-model-view-code/

MySQL Connector/Python Developer Guide:
https://dev.mysql.com/doc/connector-python/en/

Python MySQL Database Access:
https://www.mysqltutorial.org/python-mysql/

PyQt5 Style Sheets:
https://doc.qt.io/qt-5/stylesheet.html