

NOTESAPP
A MINI PROJECT REPORT

Submitted by

Mithul Aaditya A 230701185

Mithesh Tharun S 230701184

In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE AND ENGINEERING

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)

THANDALAM

CHENNAI-602105

2024- 25

BONAFIDE CERTIFICATE

Certified that this project report “**STUDENT MANAGEMENT SYSTEM**” is the bonafide work of “**“Mithul Aaditya A (230701185), Mithesh Tharun S(230701184)”**” who carried out the project work under my supervision.

Submitted for the Practical Examination held on _____

SIGNATURE

Mrs.K. MAHESMEENA
Assistant Professor,
Computer Science and Engineering
Rajalakshmi Engineering College,
(Autonomous),
Thandalam, Chennai - 602 105

INTERNAL EXAMINER

EXTERNAL EXAMINER

ABSTRACT

This project aims to develop a **Notes Application** using Python and MySQL, designed to streamline and automate the process of creating, managing, and organizing personal and professional notes. The system leverages Python for application logic and MySQL for robust data storage and retrieval.

The Notes Application provides a comprehensive solution for managing notes effectively. Key functionalities include user registration, secure login, creating and editing notes, categorizing notes, and tagging them for easy retrieval. Users can filter and search their notes based on categories, tags, or content, ensuring a seamless organization and accessibility experience.

The system ensures data integrity and security by implementing password hashing and user authentication mechanisms. It supports multi-user environments, allowing individuals to manage their notes privately and efficiently. The Notes Application is designed to enhance productivity by offering a user-friendly and reliable platform for managing information systematically.

TABLE OF CONTENTS

S.NO	TITLE	PAGE. NO
1.	INTRODUCTION	2
	1.1. OBJECTIVES	2
	1.2. MODULE	2
2.	SURVEY OF TECHNOLOGIES	3
	2.1.SOFTWARE DESCRIPTION	3
	2.2. LANGUAGE	3
	2.2.1. SQL	3
	2.2.2. PYTHON	4
3.	REQUIREMENTS AND ANALYSIS	4
	3.1 REQUIREMENT SPECIFICATIONS ⁵	5
	3.2 HARDWARE AND SOFTWARE REQUIREMENTS ⁶	5
	3.3 DATA FLOW DIAGRAM ⁶	6
	3.4 DATA DICTIONARY ⁶	7
	3.5 ER-DIAGRAM ⁶	8
	3.6 NORMALIZATION ⁶	8
4.	PROGRAM CODE	11
5.	RESULT AND DISCUSSION	24
6.	TESTING	29
7.	CONCLUSION	30
8.	FUTURE ENHANCEMENTS	30

1. INTRODUCTION

A NotesApp is an essential tool for individuals or organizations to efficiently manage and organize notes, tasks, and resources. By utilizing Python for its robust programming capabilities and SQL for powerful database management, this system can streamline processes such as note creation, categorization, tagging, sharing, and version control. Python's versatility ensures seamless integration with SQL databases, facilitating secure and efficient storage, retrieval, and manipulation of note data. Implementing a NotesApp with these technologies not only enhances the user experience but also improves the overall accessibility, organization, and sharing of notes and related information.

1.1. OBJECTIVES

The Notes Application is designed to provide users with a secure and efficient platform to create, manage, and organize their notes. Built using Python and MySQL, it ensures robust data handling and user-friendly interaction.

Key features include user authentication for secure access, allowing users to register and log in with hashed passwords for enhanced security. Users can create, update, delete, and retrieve notes seamlessly. The app supports organizing notes by categories and tags, enabling better classification and quick access.

Users can search and filter notes based on titles, content, categories, or tags, making the system highly intuitive and efficient. The database structure is designed to handle multiple users, ensuring unique and private data storage for each account.

The application aims to simplify note-taking and organization, ensuring productivity and ease of use for both personal and professional purposes. It offers a scalable and reliable solution for managing notes while maintaining data integrity and security.

1.2. MODULES

- Database Module
- Student Management Module

- Course Management Module
- Grade Management Module
- Attendance Management Module
- Enroll Management Module

2. SURVEY OF TECHNOLOGIES

2.1. SOFTWARE DESCRIPTION

PyCharm is an Integrated Development Environment (IDE) specifically designed for Python development. PyCharm provides a comprehensive set of tools for coding, debugging, testing, and deploying Python applications. It offers a user-friendly interface and a wide range of features. One of the key features of PyCharm is its powerful code editor, which supports syntax highlighting, code completion, and code analysis, helping developers write clean and error-free code more efficiently. It also comes with built-in support for version control systems like Git, making it easy to manage code repositories directly from the IDE.

2.2. LANGUAGE

2.2.1. SQL

SQL, or Structured Query Language, is a powerful and widely-used programming language designed for managing and manipulating relational databases. It provides a standardized way to interact with databases, allowing users to perform various tasks such as querying data, updating records, and defining database structures. SQL is essential for creating, retrieving, updating, and deleting data from databases. One of the key features of SQL is its ability to perform complex queries on large datasets efficiently. Using SQL, users can write queries to retrieve specific information from databases based on specified criteria, such as selecting all students enrolled in a particular course or calculating the average grade for a group of students. SQL also provides mechanisms for data manipulation, including inserting new records into a

database, updating existing records, and deleting unwanted data. This makes it a versatile tool for managing data throughout its lifecycle. Additionally, SQL allows for the creation and management of database structures such as tables, indexes, views, and stored procedures. These structures help organize and optimize data storage and retrieval, ensuring efficient operation of the database system.

2.2.2. PYTHON

Python is a high-level, interpreted programming language known for its simplicity and readability. Guido van Rossum released Python in 1991, and since then, it has gained immense popularity in various fields, including web development, data analysis, artificial intelligence, and scientific computing. One of Python's key strengths lies in its clean and concise syntax, which allows developers to write code that is easy to understand and maintain. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming, making it versatile for a wide range of applications. It comes with a comprehensive standard library that provides ready-to-use modules and functions for tasks such as file I/O, networking, and data processing, reducing the need for external dependencies. Python's dynamic typing and automatic memory management simplify development, as programmers do not need to declare variable types explicitly, and memory allocation and deallocation are handled by the interpreter. This feature contributes to Python's flexibility and ease of use, especially for beginners.

3. REQUIREMENTS AND ANALYSIS

3.1. REQUIREMENT SPECIFICATION

The Student Management System (SMS) aims to streamline administrative

HARDWARE AND SOFTWARE REQUIREMENTS

The Notes Application (NA) aims to provide users with a secure and efficient platform to manage and organize their notes, categories, and tags for personal or professional use.

- **User**

The system shall allow users to register and log in securely. User records shall include information such as username, email, and a hashed password for authentication.

- **NoteManagement:**

Users shall be able to add, edit, and delete notes. Each note shall include details such as title, content, timestamp, and optional category or tags. Notes shall be stored securely and organized for easy retrieval.

- **CategoryManagement:**

The system shall support categorizing notes by allowing users to assign notes to categories. Categories shall include details such as a category name and a unique ID. Users can filter notes based on assigned categories.

- **TagManagement:**

Users shall have the ability to assign tags to notes for further classification. Tags shall be managed separately, and users can associate multiple tags with a single note for enhanced searchability.

- **SearchandFiltering:**

The application shall provide functionality for users to search notes by title, content, or associated tags. Notes can also be filtered by categories for better organization and accessibility.

- **DataSecurity:**

All user credentials and sensitive data shall be securely stored using password hashing and encrypted database connections.

- **ScalabilityandPerformance:**

The system shall handle multiple users and support seamless retrieval of notes, even for users with a large volume of data.

- **ReportsandInsights(Optional):**

The system may allow users to generate simple reports, such as the total number of notes, notes by category, or tag usage statistics.

This Notes Application is intended to enhance productivity by offering a simple yet powerful tool for managing notes effectively.

Software Requirements

- Operating System Windows 11
- Front End: Python
- Back End: MySQL

Hardware Requirements

- Desktop PC or a Laptop
- Operating System – Windows 10 , 64-bit operating system
- Intel® Core™ i3-6006U CPU @ 2.00GHz
- 4.00 GB RAM
- Keyboard and Mouse

3.2. DATA FLOW DIAGRAM

Column Name	Data Type	Description
enrollment_id	INT	Primary key, auto-incremented enrollment ID
student_id	INT	Foreign key referencing students(student_id)
course_id	INT	Foreign key referencing courses(course_id)
enrollment_date	DATE	Date of enrollment

- **grades**

Column Name	Data Type	Description
grade_id	INT	Primary key, auto-incremented grade ID
student_id	INT	Foreign key referencing students(student_id)
course_id	INT	Foreign key referencing courses(course_id)
grade	CHAR(1)	Grade assigned to the student for the course

- **attendance**

Column Name	Data Type	Description
attendance_id	INT	Primary key, auto-incremented attendance ID
student_id	INT	Foreign key referencing students(student_id)
course_id	INT	Foreign key referencing courses(course_id)
attendance_date	DATE	Date of attendance record
status	ENUM('Present', 'Absent')	Attendance status (Present/Absent)

3.4. ER DIAGRAM

Users Table

Column Name	Data Type	Constraints
user_id	INT	PRIMARY KEY, AUTO_INCREMENT
username	VARCHAR(50)	NOT NULL, UNIQUE
email	VARCHAR(100)	NOT NULL, UNIQUE
password_hash	VARCHAR(255)	NOT NULL
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

2. Categories Table

Column Name	Data Type	Constraints
category_id	INT	PRIMARY KEY, AUTO_INCREMENT
user_id	INT	NOT NULL, FOREIGN KEY (users)
category_name	VARCHAR(50)	NOT NULL
description	TEXT	NULLABLE
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

3. Tags Table

Column Name	Data Type	Constraints
tag_id	INT	PRIMARY KEY, AUTO_INCREMENT

Column Name	Data Type	Constraints
user_id	INT	NOT NULL, FOREIGN KEY (users)
tag_name	VARCHAR(50)	NOT NULL
description	TEXT	NULLABLE
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

4. Notes Table

Column Name	Data Type	Constraints
note_id	INT	PRIMARY KEY, AUTO_INCREMENT
user_id	INT	NOT NULL, FOREIGN KEY (users)
title	VARCHAR(100)	NULLABLE
content	TEXT	NULLABLE
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
category_id	INT	NULLABLE, FOREIGN KEY (categories)

5. Note_Tags Table

Column Name	Data Type	Constraints
note_id	INT	PRIMARY KEY (Composite), FOREIGN KEY (notes)
tag_id	INT	PRIMARY KEY (Composite), FOREIGN KEY (tags)

6. Shared_Notes Table

Column Name	Data Type	Constraints
note_id	INT	PRIMARY KEY (Composite), FOREIGN KEY (notes)
shared_with_user_id	INT	PRIMARY KEY (Composite), FOREIGN KEY (users)
shared_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

4. PROGRAM CODE

PYTHON CODE:

```
import mysql.connector

from mysql.connector import Error

import hashlib
```



```
# Database connection function
```

```
def connect_to_database():
```

```
    try:
```

```
        connection = mysql.connector.connect(
```

```
            host="localhost",
```

```
            user="root", # Replace with your MySQL username
```

```
            password="mithu133", # Replace with your MySQL password
```

```
            database="NotesApp"
```

```
        )
```

```
        return connection
```

```
    except Error as e:
```

```
        print(f"Error connecting to database: {e}")
```

```
        return None
```

```
# User authentication
```

```
def register_user(username, email, password):
```

```
    conn = connect_to_database()
```

```
    if not conn:
```

```
        return "Database connection failed!"
```

```

try:

    cursor = conn.cursor()

    hashed_password = hashlib.sha256(password.encode()).hexdigest()

    cursor.execute("INSERT INTO users (username, email, password_hash)
VALUES (%s, %s, %s)",

                (username, email, hashed_password))

    conn.commit()

    return "User registered successfully!"

except mysql.connector.Error as e:

    return f"Error: {e}"

finally:

    conn.close()

```

```

def login_user(username, password):

    conn = connect_to_database()

    if not conn:

        return None

    try:

        cursor = conn.cursor(dictionary=True)

        hashed_password = hashlib.sha256(password.encode()).hexdigest()

```

```

        cursor.execute("SELECT * FROM users WHERE username = %s AND
password_hash = %s",

                        (username, hashed_password))

    user = cursor.fetchone()

    return user

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return None

finally:

    conn.close()


# Notes management

def add_note(user_id, title, content, category_id=None):

    conn = connect_to_database()

    if not conn:

        return "Database connection failed!"

    try:

        cursor = conn.cursor()

        cursor.execute("INSERT INTO notes (user_id, title, content, category_id)
VALUES (%s, %s, %s, %s)",

```

```

        (user_id, title, content, category_id))

    conn.commit()

    return "Note added successfully!"

except mysql.connector.Error as e:

    return f"Error: {e}"

finally:

    conn.close()


def get_user_notes(user_id):

    conn = connect_to_database()

    if not conn:

        return []

    try:

        cursor = conn.cursor(dictionary=True)

        cursor.execute("SELECT * FROM notes WHERE user_id = %s ORDER BY
updated_at DESC", (user_id,))

        return cursor.fetchall()

    except mysql.connector.Error as e:

        print(f"Error: {e}")

        return []

```

finally:

conn.close()

def get_note_by_id(note_id):

conn = connect_to_database()

if not conn:

return None

try:

cursor = conn.cursor(dictionary=True)

cursor.execute("SELECT * FROM notes WHERE note_id = %s", (note_id,))

return cursor.fetchone()

except mysql.connector.Error as e:

print(f"Error: {e}")

return None

finally:

conn.close()

def get_notes_by_category(user_id, category_id):

conn = connect_to_database()

```

if not conn:

    return []

try:

    cursor = conn.cursor(dictionary=True)

    cursor.execute("""

        SELECT * FROM notes WHERE user_id = %s AND category_id = %s

    """, (user_id, category_id))

    return cursor.fetchall()

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return []

finally:

    conn.close()

```

```

def update_note(note_id, new_title, new_content):

    conn = connect_to_database()

    if not conn:

        return False

    try:

```

```

        cursor = conn.cursor()

        cursor.execute("""

            UPDATE notes

            SET title = %s, content = %s

            WHERE note_id = %s

            """, (new_title, new_content, note_id))

        conn.commit()

        return True # Return True if update is successful

    except mysql.connector.Error as e:

        print(f"Error: {e}")

        return False

    finally:

        conn.close()

def get_tags_for_note(note_id):

    conn = connect_to_databas()

    if not conn:

        return []

    try:

        cursor = conn.cursor(dictionary=True)

```

```

cursor.execute("""

SELECT t.tag_name

FROM tags t

JOIN note_tags nt ON t.tag_id = nt.tag_id

WHERE nt.note_id = %s

""", (note_id,))

tags = cursor.fetchall()

return [tag['tag_name'] for tag in tags]

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return []

finally:

    conn.close()

```

```

def update_note_category(note_id, category_id):

    conn = connect_to_database()

    if not conn:

        return

    try:

```



```

        cursor = conn.cursor()

        cursor.execute("""

            UPDATE notes SET category_id = %s WHERE note_id = %s

        """, (category_id, note_id))

        conn.commit()

    except mysql.connector.Error as e:

        print(f"Error: {e}")

    finally:

        conn.close()

def update_note_tags(note_id, tag_ids):

    conn = connect_to_database()

    if not conn:

        return

    try:

        cursor = conn.cursor()

        # Delete existing tags for this note

        cursor.execute("""

            DELETE FROM note_tags WHERE note_id = %s

```

```

        """, (note_id,))

# Insert new tags

for tag_id in tag_ids:

    cursor.execute("""

        INSERT INTO note_tags (note_id, tag_id) VALUES (%s, %s)

        """, (note_id, tag_id))

    conn.commit()

except mysql.connector.Error as e:

    print(f"Error: {e}")

finally:

    conn.close()

def get_notes_for_tag(user_id, tag_id):

    conn = connect_to_database()

    if not conn:

        return []

    try:

        cursor = conn.cursor(dictionary=True)

```

```

        cursor.execute("""

            SELECT notes.note_id, notes.title, notes.content

            FROM notes

            JOIN note_tags ON notes.note_id = note_tags.note_id

            WHERE note_tags.tag_id = %s AND notes.user_id = %s

            """, (tag_id, user_id))

        return cursor.fetchall()

    except mysql.connector.Error as e:

        print(f"Error: {e}")

        return []

    finally:

        conn.close()

def get_category_by_id(user_id, category_id):

    conn = connect_to_database()

    if not conn:

        return None

    try:

        cursor = conn.cursor(dictionary=True)

        cursor.execute("""

```

```

        SELECT category_id, category_name

        FROM categories

        WHERE user_id = %s AND category_id = %s

        """ , (user_id, category_id))

    category = cursor.fetchone()

    return category

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return None

finally:

    conn.close()


def view_notes_for_category(user_id, category_id):

    conn = connect_to_database()

    if not conn:

        return []

    try:

        cursor = conn.cursor(dictionary=True)

        cursor.execute("""

            SELECT n.note_id, n.title, n.content, n.updated_at

```

```

        FROM notes n

        JOIN categories c ON n.category_id = c.category_id

        WHERE n.user_id = %s AND n.category_id = %s

        ORDER BY n.updated_at DESC

        """ , (user_id, category_id))

    notes = cursor.fetchall()

    return notes

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return []

finally:

    conn.close()

```

```

def get_category_name_for_note(note_id):

    conn = connect_to_database()

    if not conn:

        return None

    try:

        cursor = conn.cursor(dictionary=True)

```

```

        cursor.execute("""

            SELECT category_name FROM categories

            WHERE category_id = (SELECT category_id FROM notes WHERE
note_id = %s)

            """, (note_id,))

        category = cursor.fetchone()

        return category['category_name'] if category else None

    except mysql.connector.Error as e:

        print(f"Error: {e}")

        return None

    finally:

        conn.close()

def update_tags_for_note(note_id, tags):

    conn = connect_to_database()

    if not conn:

        return

    try:

        cursor = conn.cursor()

        # First, delete all existing tags for the note

```

```

cursor.execute("""

DELETE FROM note_tags WHERE note_id = %s

""", (note_id,))

# Then, insert the new tags

for tag in tags:

    # Check if tag exists, if not, add it

    cursor.execute("""

        SELECT tag_id FROM tags WHERE tag_name = %s

        """, (tag,))

    tag_id = cursor.fetchone()

    if not tag_id:

        # Add the tag if it doesn't exist

        cursor.execute("""

            INSERT INTO tags (tag_name) VALUES (%s)

            """, (tag,))

        cursor.execute("""

            SELECT tag_id FROM tags WHERE tag_name = %s

            """, (tag,))

```

```

        tag_id = cursor.fetchone()

        cursor.execute("""

            INSERT INTO note_tags (note_id, tag_id) VALUES (%s, %s)

            """, (note_id, tag_id['tag_id']))

        conn.commit()

    except mysql.connector.Error as e:

        print(f"Error: {e}")

    finally:

        conn.close()

```

Searching notes

```
def search_notes_by_term(user_id, search_term):
```

```
    conn = connect_to_database()
```

```
    if not conn:
```

```
        return []
```

```
    try:
```



```

cursor = conn.cursor(dictionary=True)

cursor.execute("""

        SELECT * FROM notes WHERE user_id = %s AND (title LIKE %s OR
content LIKE %s)

        """, (user_id, f"% {search_term} %", f"% {search_term} %"))

# Fetch and return the results

notes = cursor.fetchall()

return notes

except mysql.connector.Error as e:

    print(f"Error: {e}")

    return []

finally:

    cursor.close() # Explicitly close the cursor

    conn.close()

# Categories management

def add_category(user_id, category_name):

    conn = connect_to_database()

    if not conn:

```

```

        return "Database connection failed!"

    try:

        cursor = conn.cursor()

        cursor.execute("INSERT INTO categories (user_id, category_name)
VALUES (%s, %s)",

                        (user_id, category_name))

        conn.commit()

        return "Category added successfully!"

    except mysql.connector.Error as e:

        return f"Error: {e}"

    finally:

        conn.close()


def get_categories(user_id):

    conn = connect_to_database()

    if not conn:

        return []

    try:

        cursor = conn.cursor(dictionary=True)

        cursor.execute("SELECT * FROM categories WHERE user_id = %s",

```

```
(user_id,))
```

```
    return cursor.fetchall()
```

```
except mysql.connector.Error as e:
```

```
    print(f"Error: {e}")
```

```
    return []
```

```
finally:
```

```
    conn.close()
```

```
# Tags management
```

```
def add_tag(user_id, tag_name):
```

```
    conn = connect_to_database()
```

```
    if not conn:
```

```
        return "Database connection failed!"
```

```
    try:
```

```
        cursor = conn.cursor()
```

```
        cursor.execute("INSERT INTO tags (user_id, tag_name) VALUES (%s,  
%s)",
```

```
                        (user_id, tag_name))
```

```
        conn.commit()
```

```
        return "Tag added successfully!"
```

```
except mysql.connector.Error as e:
```

```
    return f"Error: {e}"
```

```
finally:
```

```
    conn.close()
```

```
def get_tags(user_id):
```

```
    conn = connect_to_database()
```

```
    if not conn:
```

```
        return []
```

```
    try:
```

```
        cursor = conn.cursor(dictionary=True)
```

```
        cursor.execute("SELECT * FROM tags WHERE user_id = %s", (user_id,))
```

```
        return cursor.fetchall()
```

```
    except mysql.connector.Error as e:
```

```
        print(f"Error: {e}")
```

```
        return []
```

```
    finally:
```

```
        conn.close()
```

Note-Tags Relationship

```
def add_tags_to_note(note_id, tag_ids):

    conn = connect_to_database()

    if not conn:

        return "Database connection failed!"

    try:

        cursor = conn.cursor()

        for tag_id in tag_ids:

            cursor.execute("INSERT INTO note_tags (note_id, tag_id) VALUES (%s, %s)", (note_id, tag_id))

        conn.commit()

        return "Tags added to note successfully!"

    except mysql.connector.Error as e:

        return f"Error: {e}"

    finally:

        conn.close()
```

Sharing notes

```
def share_note(user_id, note_id, target_username):
```

```

conn = connect_to_database()

if not conn:

    return "Database connection failed!"

try:

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT * FROM users WHERE username = %s",
(target_username,))

    target_user = cursor.fetchone()

    if not target_user:

        return "Target user not found!"

    cursor.execute("INSERT INTO shared_notes (user_id, note_id,
shared_with_user_id) VALUES (%s, %s, %s)",

        (user_id, note_id, target_user['user_id']))

    conn.commit()

    return "Note shared successfully!"

except mysql.connector.Error as e:

    return f"Error: {e}"

finally:

    conn.close()

import tkinter as tk

```

```

from tkinter import messagebox, simpledialog

from database import register_user, login_user, add_note,
get_user_notes, get_notes_for_tag, update_note, search_notes_by_term, add_category,
get_categories, add_tag, get_tags,
add_tags_to_note, get_note_by_id, get_notes_by_category, get_tags_for_note, update_note_
category, update_tags_for_note, update_note_tags, get_category_name_for_note


class NotesApp:

    def __init__(self, root):

        self.root = root

        self.root.title("Notes Application")

        self.current_user = None

        self.setup_login_screen()

    def setup_login_screen(self):

        self.clear_screen()

        tk.Label(self.root, text="Login", font=("Arial", 16)).pack(pady=10)

        self.username_label = tk.Label(self.root, text="Username")

        self.username_label.pack()

        self.username_entry = tk.Entry(self.root)

```

```
self.username_entry.pack(pady=5)
```

```
self.password_label = tk.Label(self.root, text="Password")
```

```
self.password_label.pack()
```

```
self.password_entry = tk.Entry(self.root, show="*")
```

```
self.password_entry.pack(pady=5)
```

```
self.login_button = tk.Button(self.root, text="Login", command=self.login)
```

```
self.login_button.pack(pady=10)
```

```
self.signup_button = tk.Button(self.root, text="Sign Up",  
command=self.setup_signup_screen)
```

```
self.signup_button.pack()
```

```
def setup_signup_screen(self):
```

```
    self.clear_screen()
```

```
    tk.Label(self.root, text="Sign Up", font=("Arial", 16)).pack(pady=10)
```

```
    self.signup_username_label = tk.Label(self.root, text="Username")
```

```
    self.signup_username_label.pack()
```



```
self.signup_username_entry = tk.Entry(self.root)

self.signup_username_entry.pack(pady=5)


self.signup_email_label = tk.Label(self.root, text="Email")

self.signup_email_label.pack()

self.signup_email_entry = tk.Entry(self.root)

self.signup_email_entry.pack(pady=5)


self.signup_password_label = tk.Label(self.root, text="Password")

self.signup_password_label.pack()

self.signup_password_entry = tk.Entry(self.root, show="*")

self.signup_password_entry.pack(pady=5)


self.signup_button = tk.Button(self.root, text="Sign Up",
command=self.signup)

self.signup_button.pack(pady=10)


self.back_button = tk.Button(self.root, text="Back to Login",
command=self.setup_login_screen)

self.back_button.pack()
```

```

def signup(self):

    username = self.signup_username_entry.get()

    email = self.signup_email_entry.get()

    password = self.signup_password_entry.get()


    if not username or not email or not password:

        messagebox.showerror("Error", "All fields are required.")

        return


    result = register_user(username, email, password)

    messagebox.showinfo("Sign Up", result)


    if "successfully" in result:

        self.setup_login_screen()


def login(self):

    username = self.username_entry.get()

    password = self.password_entry.get()


    if not username or not password:

```

```

        messagebox.showerror("Error", "Both fields are required.")

    return

user = login_user(username, password)

if user:

    self.current_user = user

    self.setup_notes_screen()

else:

    messagebox.showerror("Login Failed", "Invalid username or password.")


def view_or_edit_note(self, note):

    popup = tk.Toplevel(self.root)

    popup.title(f"Edit Note: {note['title']}")


    # Title and content fields

    title_label = tk.Label(popup, text="Title")

    title_label.pack()


    title_entry = tk.Entry(popup)

    title_entry.insert(0, note['title']) # Pre-fill with current title

```

```

title_entry.pack(pady=5)


content_label = tk.Label(popup, text="Content")

content_label.pack()


content_text = tk.Text(popup, height=10, width=30)

content_text.insert("1.0", note['content']) # Pre-fill with current content

content_text.pack(pady=5)


# Category selection

category_label = tk.Label(popup, text="Category")

category_label.pack()


categories = get_categories(self.current_user['user_id'])

category_options = [cat['category_name'] for cat in categories]

category_var = tk.StringVar(popup)

category_var.set(next((cat['category_name'] for cat in categories if
cat['category_id'] == note['category_id']), categories[0]['category_name']))

category_dropdown = tk.OptionMenu(popup, category_var,
*category_options)

category_dropdown.pack(pady=5)

```

```

# Tags selection (Allow user to choose tags for the note)

tag_label = tk.Label(popup, text="Tags")

tag_label.pack()


tags = get_tags(self.current_user['user_id'])

tag_options = [tag['tag_name'] for tag in tags]

tag_var = tk.StringVar(popup)

tag_var.set(", ".join([tag['tag_name'] for tag in
get_tags_for_note(note['note_id'])])) # Pre-fill with current tags


tag_dropdown = tk.OptionMenu(popup, tag_var, *tag_options)

tag_dropdown.pack(pady=5)


def save_edited_note():

    updated_title = title_entry.get()

    updated_content = content_text.get("1.0", "end-1c")

    updated_category = category_var.get()

    updated_tags = tag_var.get().split(", ") # Get tags from dropdown
(comma-separated)

```

```

        if updated_title and updated_content:

            # Update the note details

            category_id = next((cat['category_id'] for cat in categories if
cat['category_name'] == updated_category), None)

            if category_id:

                update_note(note['note_id'], updated_title, updated_content,
category_id)

                # Update tags

                update_tags_for_note(note['note_id'], updated_tags)

                messagebox.showinfo("Note Updated", "The note has been updated
successfully.")

                popup.destroy()

            else:

                messagebox.showerror("Error", "Invalid category selected.")

        else:

            messagebox.showerror("Error", "Title and content cannot be empty.")

save_button = tk.Button(popup, text="Save", command=save_edited_note)

save_button.pack(pady=10)

cancel_button = tk.Button(popup, text="Cancel", command=popup.destroy)

```

```
cancel_button.pack(pady=5)
```

```
def setup_notes_screen(self):
```

```
    self.clear_screen()
```

```
    tk.Label(self.root, text=f"Welcome, {self.current_user['username']}!",  
font=("Arial", 16)).pack(pady=10)
```

```
    self.notes_frame = tk.Frame(self.root)
```

```
    self.notes_frame.pack(pady=10)
```

```
    self.display_notes()
```

```
    tk.Button(self.root, text="Add Note",  
command=self.add_note_popup).pack(pady=5)
```

```
    tk.Button(self.root, text="Search Notes",  
command=self.search_notes).pack(pady=5)
```

```
    tk.Button(self.root, text="Categories",  
command=self.manage_categories).pack(pady=5)
```

```
    tk.Button(self.root, text="Tags", command=self.manage_tags).pack(pady=5)
```

```
    tk.Button(self.root, text="Logout", command=self.logout).pack()
```

```

def display_notes(self):

    for widget in self.notes_frame.winfo_children():

        widget.destroy()

    notes = get_user_notes(self.current_user['user_id'])

    for note in notes:

        note_frame = tk.Frame(self.notes_frame)

        note_frame.pack(pady=5, fill="x")

        note_label = tk.Label(note_frame, text=note['title'], font=("Arial", 12))

        note_label.pack(side="left", padx=5)

        category_name = get_category_name_for_note(note['note_id'])

        category_label = tk.Label(note_frame, text=f"Category:
{category_name}")

        category_label.pack(side="left", padx=5)

        # Tags display

        tags = get_tags_for_note(note['note_id'])

        tag_label = tk.Label(note_frame, text=f"Tags: {' '.join(tags)}")

```



```

tag_label.pack(side="left", padx=5)

# Edit buttons

edit_button = tk.Button(note_frame, text="Edit", command=lambda
note=note: self.view_or_edit_note(note))

edit_button.pack(side="right", padx=5)

edit_category_button = tk.Button(note_frame, text="Edit Category",
command=lambda note=note: self.edit_category(note))

edit_category_button.pack(side="right", padx=5)

edit_tags_button = tk.Button(note_frame, text="Edit Tags",
command=lambda note=note: self.edit_tags(note))

edit_tags_button.pack(side="right", padx=5)

def edit_category(self, note):

    popup = tk.Toplevel(self.root)

    popup.title("Edit Category")

# Get available categories

categories = get_categories(self.current_user['user_id'])

```

```

category_options = [cat['category_name'] for cat in categories]

category_var = tk.StringVar(popup)


# Set the current category as default

current_category = get_category_name_for_note(note['note_id'])

category_var.set(current_category)


category_dropdown = tk.OptionMenu(popup, category_var,
*category_options)

category_dropdown.pack(pady=10)


def save_category():

    selected_category = category_var.get()

    category_id = next((cat['category_id'] for cat in categories if
cat['category_name'] == selected_category), None)

    if category_id:

        update_note_category(note['note_id'], category_id)

        popup.destroy()

        self.display_notes()


save_button = tk.Button(popup, text="Save", command=save_category)

```

```
save_button.pack(pady=10)
```

```
def edit_tags(self, note):
```

```
    popup = tk.Toplevel(self.root)
```

```
    popup.title("Edit Tags")
```

```
    # Get available tags
```

```
    tags = get_tags(self.current_user['user_id'])
```

```
    tag_options = [tag['tag_name'] for tag in tags]
```

```
    tag_var = tk.StringVar(popup)
```

```
    # Set current tags for this note
```

```
    current_tags = get_tags_for_note(note['note_id'])
```

```
    tag_var.set(", ".join(current_tags))
```

```
    tag_dropdown = tk.OptionMenu(popup, tag_var, *tag_options)
```

```
    tag_dropdown.pack(pady=10)
```

```
def save_tags():
```

```
    selected_tags = tag_var.get().split(", ") # Split by commas if multiple tags
```

selected

```

tag_ids = [tag['tag_id'] for tag in tags if tag['tag_name'] in selected_tags]

update_note_tags(note['note_id'], tag_ids)

popup.destroy()

self.display_notes()


save_button = tk.Button(popup, text="Save", command=save_tags)

save_button.pack(pady=10)


def edit_note_popup(self, note):

    popup = tk.Toplevel(self.root)

    popup.title("Edit Note")


    title_label = tk.Label(popup, text="Title")

    title_label.pack()


    # Pre-fill the title field with the current note's title

    title_entry = tk.Entry(popup)

    title_entry.insert(0, note['title']) # Default text as current note's title

    title_entry.pack()

```

```

content_label = tk.Label(popup, text="Content")

content_label.pack()


# Pre-fill the content field with the current note's content

content_text = tk.Text(popup, height=10, width=30)

content_text.insert("1.0", note['content']) # Default text as current note's
content

content_text.pack()


def save_changes():

    new_title = title_entry.get()

    new_content = content_text.get("1.0", "end-1c")


    if new_title and new_content:

        update_result = update_note(note['note_id'], new_title, new_content)

        if update_result:

            popup.destroy()

            self.display_notes() # Refresh the notes display

        else:

            messagebox.showerror("Error", "Failed to update the note.")

    else:

```

```
messagebox.showerror("Error", "Both title and content are required.")
```

```
save_button = tk.Button(popup, text="Save", command=save_changes)
```

```
save_button.pack()
```

```
def add_note_popup(self):
```

```
    popup = tk.Toplevel(self.root)
```

```
    popup.title("Add Note")
```

```
    title_label = tk.Label(popup, text="Title")
```

```
    title_label.pack()
```

```
    title_entry = tk.Entry(popup)
```

```
    title_entry.pack()
```

```
    content_label = tk.Label(popup, text="Content")
```

```
    content_label.pack()
```

```
    content_text = tk.Text(popup, height=10, width=30)
```

```

content_text.pack()

# Category selection

category_label = tk.Label(popup, text="Category")

category_label.pack()

categories = get_categories(self.current_user['user_id'])

category_options = [cat['category_name'] for cat in categories]

if category_options: # Only show category dropdown if categories exist

    category_var = tk.StringVar(popup)

    category_var.set("") # Default empty value

    category_dropdown = tk.OptionMenu(popup, category_var,
*category_options)

    category_dropdown.pack()

else:

    category_label.pack_forget() # Hide category label if no categories

    message = tk.Label(popup, text="No categories available.")

    message.pack()

# Tag selection

```

```

tag_label = tk.Label(popup, text="Tags")

tag_label.pack()


tags = get_tags(self.current_user['user_id'])

tag_options = [tag['tag_name'] for tag in tags]


if tag_options: # Only show tag dropdown if tags exist

    tag_var = tk.StringVar(popup)

    tag_var.set("") # Default empty value

    tag_dropdown = tk.OptionMenu(popup, tag_var, *tag_options)

    tag_dropdown.pack()

else:

    tag_label.pack_forget() # Hide tag label if no tags

    message = tk.Label(popup, text="No tags available.")

    message.pack()


def save_note():

    title = title_entry.get()

    content = content_text.get("1.0", "end-1c")

    category_name = category_var.get() if category_options else None

```



```

        category_id = next((cat['category_id'] for cat in categories if
cat['category_name'] == category_name), None)

        tags = [tag_var.get()] if tag_options else []

        if title and content:

            note_id = add_note(self.current_user['user_id'], title, content,
category_id)

            add_tags_to_note(note_id, tags)

            popup.destroy()

        save_button = tk.Button(popup, text="Save", command=save_note)

        save_button.pack()

    def view_category_notes(self, category):

        self.clear_screen()

        tk.Label(self.root, text=f"Notes in {category['category_name']}",
font=("Arial", 16)).pack(pady=10)

        notes = get_notes_by_category(self.current_user['user_id'],
category['category_id'])

        if notes:

```

```

        for note in notes:

            note_button = tk.Button(self.root, text=note['title'], command=lambda
note=note: self.view_or_edit_note(note))

            note_button.pack(pady=5)

        else:

            messagebox.showinfo("No Notes", f"There are no notes in the
{category['category_name']} category.")

            tk.Button(self.root, text="Back to Categories",
command=self.manage_categories).pack(pady=10)

def search_notes(self):

    # Ask the user for the search term

    search_term = simpledialog.askstring("Search Notes", "Enter search term:")

    if search_term: # Ensure the search term is not empty

        notes = search_notes_by_term(self.current_user['user_id'], search_term) #
Get search results from the database

        self.display_search_results(notes) # Display search results

    else:

        messagebox.showinfo("Search", "Please enter a search term.")

```

```

def display_search_results(self, notes):

    # Clear existing notes

    for widget in self.notes_frame.winfo_children():

        widget.destroy()

    if notes: # If notes are found, display them

        for note in notes:

            note_label = tk.Label(self.notes_frame, text=note['title'], font=("Arial",
12))

            note_label.pack(pady=5)

        else: # If no notes match the search, show a message

            messagebox.showinfo("Search Results", "No notes found for your search
term.")

def manage_categories(self):

    self.clear_screen()

    tk.Label(self.root, text="Manage Categories", font=("Arial",
16)).pack(pady=10)

    categories = get_categories(self.current_user['user_id'])

```

```

if categories:

    for category in categories:

        category_button = tk.Button(self.root, text=category['category_name'],
command=lambda c=category: self.view_notes_for_category(c['category_id']))

        category_button.pack(pady=5)

    else:

        tk.Label(self.root, text="No categories available").pack(pady=5)


# Add new category

self.new_category_entry = tk.Entry(self.root)

self.new_category_entry.pack(pady=5)


tk.Button(self.root, text="Add Category",
command=self.add_category).pack(pady=10)


tk.Button(self.root, text="Back to Notes",
command=self.setup_notes_screen).pack()


def add_category(self):

    category_name = self.new_category_entry.get()

    if category_name:

        result = add_category(self.current_user['user_id'], category_name)

```

```

        messagebox.showinfo("Category", result)

        self.manage_categories()

def manage_tags(self):

    self.clear_screen()

    tk.Label(self.root, text="Manage Tags", font=("Arial", 16)).pack(pady=10)

    tags = get_tags(self.current_user['user_id'])

    if tags:

        for tag in tags:

            tag_button = tk.Button(self.root, text=tag['tag_name'],
command=lambda t=tag: self.view_notes_for_tag(t['tag_id']))

            tag_button.pack(pady=5)

        else:

            tk.Label(self.root, text="No tags available").pack(pady=5)

    self.new_tag_entry = tk.Entry(self.root)

    self.new_tag_entry.pack(pady=5)

    tk.Button(self.root, text="Add Tag", command=self.add_tag).pack(pady=10)

    tk.Button(self.root, text="Back to Notes",
command=self.setup_notes_screen).pack()

```

```

def view_notes_for_tag(self, tag_id):

    self.clear_screen()

    tk.Label(self.root, text="Notes with this Tag", font=("Arial",
16)).pack(pady=10)

    notes = get_notes_for_tag(self.current_user['user_id'], tag_id)

    if notes:

        for note in notes:

            note_button = tk.Button(self.root, text=note['title'], command=lambda
n=note: self.view_and_edit_note(n['note_id']))

            note_button.pack(pady=5)

        else:

            tk.Label(self.root, text="No notes found for this tag").pack(pady=5)

            tk.Button(self.root, text="Back to Tags",
command=self.manage_tags).pack(pady=10)

def add_tag(self):

    tag_name = self.new_tag_entry.get()

    if tag_name:

        result = add_tag(self.current_user['user_id'], tag_name)

```

```

        messagebox.showinfo("Tag", result)

    self.manage_tags()

def logout(self):

    self.current_user = None

    self.setup_login_screen()

def clear_screen(self):

    for widget in self.root.winfo_children():

        widget.destroy()

if __name__ == "__main__":

    root = tk.Tk()

    app = NotesApp(root)

    root.mainloop()

```

MYSQL QUERIES:

```

CREATE DATABASE student_management;

USE student_management;

CREATE TABLE students (

    student_id INT AUTO_INCREMENT PRIMARY KEY,

    first_name VARCHAR(100),

```

```

last_name VARCHAR(100),

date_of_birth DATE,

gender ENUM('Male', 'Female', 'Other'),

email VARCHAR(100),

phone_number VARCHAR(15)

);

CREATE TABLE courses (

    course_id INT AUTO_INCREMENT PRIMARY KEY,

    course_name VARCHAR(100),

    description TEXT,

    credits INT

);

CREATE TABLE enrollments (

    enrollment_id INT AUTO_INCREMENT PRIMARY KEY,

    student_id INT,

    course_id INT,

    enrollment_date DATE,

    FOREIGN KEY (student_id) REFERENCES students(student_id),

    FOREIGN KEY (course_id) REFERENCES courses(course_id)

);


```



```
CREATE TABLE grades (  
  
    grade_id INT AUTO_INCREMENT PRIMARY KEY,  
  
    student_id INT,  
  
    course_id INT,  
  
    grade CHAR(1),  
  
    FOREIGN KEY (student_id) REFERENCES students(student_id),  
  
    FOREIGN KEY (course_id) REFERENCES courses(course_id)  
  
);
```

```
CREATE TABLE attendance (  
  
    attendance_id INT AUTO_INCREMENT PRIMARY KEY,  
  
    student_id INT,  
  
    course_id INT,  
  
    attendance_date DATE,  
  
    status ENUM('Present', 'Absent'),  
  
    FOREIGN KEY (student_id) REFERENCES students(student_id),  
  
    FOREIGN KEY (course_id) REFERENCES courses(course_id)  
  
);
```

5. RESULT AND DISCUSSION

 Notes Application

—

□

×

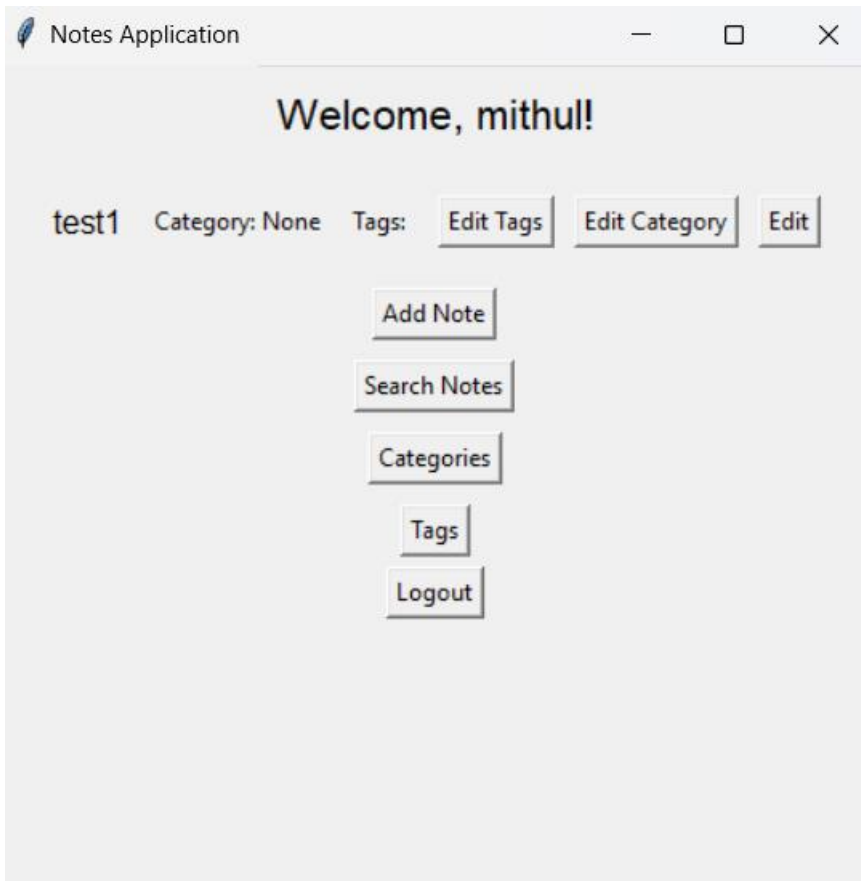
Login

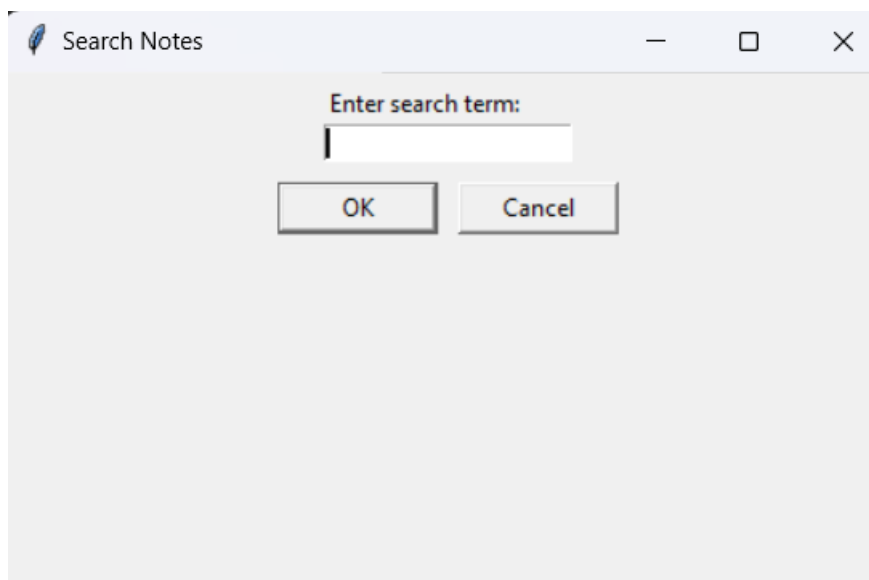
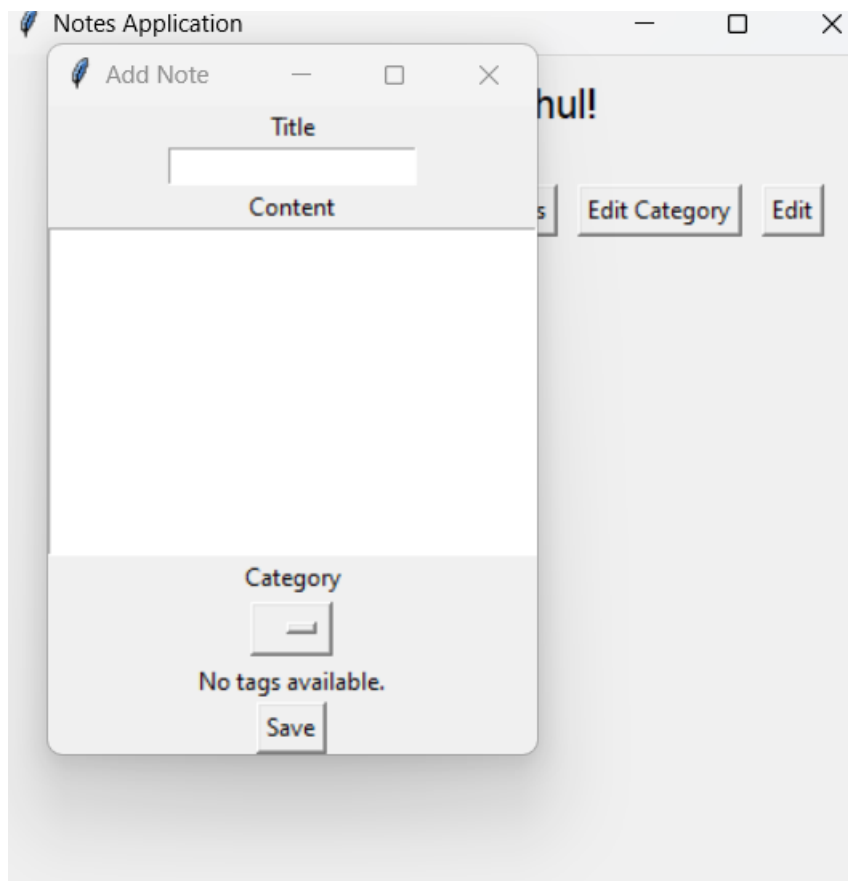
Username

Password

Login

Sign Up





TESTING

6.1 Unit Testing

Unit testing is a testing technique in which modules are tested individually. Small individual units of source code are tested to determine whether they function as intended. For the Notes Management System, individual modules such as user registration, note creation, note tagging, and category management were tested in isolation to ensure their correctness and reliability.

6.2 Integration Testing

Integration testing ensures that individual components or modules work together correctly when combined. After successful unit testing, modules such as user authentication, note creation, and sharing functionality were integrated and tested as a whole to validate data flow and interactions between them.

6.3 System Testing

System testing was conducted on the entire Notes Management System to verify that it meets its requirements. This included testing functionalities such as creating, editing, categorizing, tagging, and sharing notes. The system was deployed in different environments, and errors or bugs encountered during these tests were fixed.

6.4 Acceptance Testing

User acceptance testing was carried out to certify that the Notes Management System meets the agreed-upon requirements. Feedback was collected from potential end-users, such as students or professionals, to ensure that the system is user-friendly and functional. This was the final phase of testing before the application was deployed for production use.

7. CONCLUSION

In conclusion, the development of a Notes Management System using Python and SQL

provides a robust and user-friendly solution for organizing and managing notes. This system simplifies the process of creating, categorizing, tagging, and sharing notes, making it a valuable tool for personal and collaborative use. By leveraging Python's versatility for backend logic and SQL's reliability for database management, the system ensures data security, integrity, and accessibility. The integration of these technologies enhances the user experience, laying the foundation for future expansions, such as cloud-based accessibility and advanced analytics. This project highlights the potential of combining Python and SQL to address real-world problems and create practical applications.

8. FUTURE ENHANCEMENTS

- **Integration with Productivity Tools:** Seamless integration with productivity tools such as Google Keep, Microsoft OneNote, or Trello for a more connected workflow.
- **Mobile Application Development:** Developing a mobile app to provide users with access to their notes anytime and anywhere.
- **Advanced Search and Filtering:** Implementing advanced search capabilities using natural language processing (NLP) to make finding notes easier and faster.
- **Cloud-Based Solutions:** Migrating to a cloud-based infrastructure for enhanced scalability, accessibility, and real-time collaboration across different devices and locations.
- **Collaboration Features:** Adding features for collaborative note-taking, allowing multiple users to edit and comment on notes simultaneously.
- **Integration with AI Tools:** Incorporating AI tools for summarizing notes, suggesting tags, or even converting voice memos into text-based notes.