LIBRARY MANAGEMENT SYSTEM

PON RASHMI G R
ABHIRAMI C VARIER
PRANAV R R
RANJITH D

Description:

This project is a console-based library management system written in Python using PostgreSQL as the database. The system allows users to add books, view books, borrow and return books, register and log in as a user or administrator, and view a list of users. The system also includes features to promote or demote users between user and administrator roles, and to delete users from the system. The system provides authentication for both users and administrators, and different functionalities are available to each type of user. Additionally, the system includes a search function that allows users to search for books by either the book title or the author name

.

Github Repository link:

PonRashmiGR/Library-Management_Python (github.com)

Demo Video Link(s):

Call with Python Team-20230306_220158-Meeting Recording.mp4

Functionalities:

- Register
- Login
- Admin operations
- User operations

Requirements:

1. Python 3.8.13 is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures combined with dynamic typing and dynamic binding makes it very attractive for Rapid Application Development.
2. PostgreSQL 2.5.12 is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads.
3. pgAdmin 6.19 is a web-based Graphical User Interface (GUI) management application used to communicate with Postgres and derivative relational databases on both local and remote servers.
4. GitHub Desktop 3.1.15 is an application that enables you to interact with GitHub using a GUI instead of the command line or a web browser.
5. Visual Studio Code 1.74.3 is a code editor redefined and optimized for building and debugging modern web and cloud applications.

Installation - on Mac :

Python 3.8.13

Step 1: Download the installer package from Python's official website. https://www.python.org/downloads/

Step 2: Wait for the download to complete. Once it's finished, double-click the package to start the installation process.

Step 3: Once the installation is complete, the installer will automatically open Python's installation directory in a new Finder window.

PostgreSQL

Step 1: Open https://postgresapp.com/downloads.html and download the latest release under the downloads tab.

Step 3: To install Postgres.app, just drag it to your Applications folder and double click. Postgres.app must be placed in the /Applications folder.

pgAdmin

Step 1: Open https://www.postgresql.org/ftp/pgadmin/pgadmin4/v6.19/macos/ and download pgadmin4-6.19.dmg under files.

Step 3: To install pgAdmin.app, just drag it to your Applications folder and double click. pgAdmin will be installed.

Visual Studio Code

Step 1: Open https://code.visualstudio.com/download

Step 2: Click on Apple Silicon under the Mac icon to download VS Code's package installer for Mac in a ZIP file.

Step 3: Open finder in your mac and drag Visual Studio Code.app to the Applications folder, making it available in the macOS Launchpad.

GitHub Desktop

Step 1: Open https://desktop.github.com/

Step 2: Click *Download for macOS.*

Step 3: In the downloads tab, double click the GitHub Desktop zip file.

Step 4: After the file has been unzipped, double click the GitHub Desktop.

LIBRARY MANAGEMENT SYSTEM – CONSOLE

```
ranjithd@sivas-old-mbp Library-Management_Python % /usr/local/bin/python3 /Users/ranjithd/Documents/Library-Management_Python/main.py
Welcome to the Library Management System!
What would you like to do?
1. Register as a new user
2. Log in
3. Exit
Enter your choice (1-3): 2
```

Admin menu:

```
What would you like to do?
1. Add Books
2. List Books
3. List borrowed books
4. List users
5. Add a user
6. Promote a user to admin
7. Demote an admin to user
8. Delete a user
9. Logout
```

User menu:

```
What would you like to do?
1. List Books
2. Borrow a book
3. Return a book
4. Search
5. List borrowed books
6. Logout
Enter your choice (1-4):
```

Login/Registration:

Login():

```python
# Login function
def login():
    username = input("Enter your username: ")
    password = getpass.getpass("Enter your password: ")
    cur = conn.cursor()
    cur.execute("SELECT id, role FROM users WHERE username = %s AND password = %s", (username, password))
    row = cur.fetchone()
    cur.close()
    if not row:
        print("Invalid username or password.")
        return None
    user_id, role = row
    return (user_id, role)
```

The **login()** function allows a user to log in to their existing account by prompting them to enter their email address and password. The function then verifies that the email and password match a row in the **users** table in the database. If the login is successful, the function returns the user's ID, first name, and last name, which can be used to identify the logged-in user in other parts of the program.

If the login fails (e.g., if the user enters an incorrect email or password), the program will display an error message and prompt the user to try aga

Register():

```
# Register function
def register():
    username = input("Enter a username: ")
    password = input("Enter a password: ")
    confirm_password = input("Confirm password: ")
    if password != confirm_password:
        print("Passwords do not match. Please try again.")
        return
    cur = conn.cursor()
    cur.execute("SELECT id FROM users WHERE username = %s", (username,))
    if cur.fetchone() is not None:
        print("Username already taken. Please choose another one.")
        return
    cur.execute("INSERT INTO users (username, password, role) VALUES (%s, %s, %s)", (username, password, "user"))
    conn.commit()
    cur.close()
    print("Registration successful.")
```

The `register()` function allows a user to create a new account by prompting them to enter their first name, last name, email, password, and confirm password. The function then verifies that the password and confirm password match and that the email address has not already been registered. If the user input passes these checks, the function inserts a new row into the `users` table with the user's information, along with a generated unique user ID.

Note that if the user enters an email address that has already been registered, or if the password and confirm password do not match, the program will display an error message and prompt the user to try again.

User_menu():

```
def user_menu(user_id):
    while True:
        print("What would you like to do?")
        print("1. List Books")
        print("2. Borrow a book")
        print("3. Return a book")
        print("4. Search")
        print("5. List borrowed books")
        print("6. Logout")
        choice = input("Enter your choice (1-4): ")
        if choice == "1":
            book.list_books()
        elif choice == "2":
            usermenu.borrow_book(user_id)
        elif choice == "3":
            usermenu.return_book(user_id)
        elif choice == "4":
            book.search_book()
        elif choice == "5":
            book.list_borrowed()

        elif choice == "6":
            print("Logged out successfully.")
            break
        else:
            print("Invalid choice. Please try again.")
```

List_books():

```
# List all books in the database
def list_books():
    cur = conn.cursor()
    cur.execute("SELECT id, title, author, genre, publication_date, available FROM books")
    rows = cur.fetchall()
    cur.close()
    if not rows:
        print("No books found.")
    else:
        print("ID\tTitle\tAuthor\tgenre\tPublication Date\tAvailable")
        for row in rows:
            book_id, title, author, genre, publication_date, available = row
            print(f"{book_id}\t{title}\t{author}\t{genre}\t{publication_date}\t{available}")
```

The **list_users()** function retrieves all user information from the **users** table in the database and displays it to the user in a nicely formatted table. The table shows the user ID, first name, last name, email address, and registration date for each user.

If there are no users in the database, the program will display a message indicating that there are no users to display.

Borrow_book():

```
# Borrow a book
def borrow_book(user_id):
    book_id = input("Enter the ID of the book you want to borrow: ")
    cur = conn.cursor()
    cur.execute("SELECT available FROM books WHERE id = %s", (book_id,))
    row = cur.fetchone()
    if not row:
        print("Book not found.")
        return
    available = row[0]
    if not available:
        print("Book is not available.")
        return
    cur.execute("UPDATE books SET available = FALSE WHERE id = %s", (book_id,))
    loan_date = str(date.today())
    cur.execute("INSERT INTO loans (book_id, borrower_id, loan_date) VALUES (%s, %s, %s)", (book_id, user_id, loan_date)
    conn.commit()
    cur.close()
```

The **borrow_book()** function allows a logged-in user to borrow a book from the library by prompting them to enter the book ID and quantity they wish to borrow. The function then checks that the requested book exists in the **books** table and that there are enough copies available to satisfy the user's request. If the book is available, the function inserts a new row into the **loans** table with the user's ID, book ID, borrowed date, and due date.

Note that if the user enters a non-integer quantity value, the program will throw a **ValueError** and exit. If the requested book is not available, the program will display an error message and prompt the user to try again.

Return_book():

```
# Return a book
def return_book(user_id):
    book_id = input("Enter the ID of the book you want to return: ")
    cur = conn.cursor()
    cur.execute("SELECT borrower_id, loan_date, return_date FROM loans WHERE book_id = %s AND return_date IS NULL", (boo
    row = cur.fetchone()
    if not row:
        print("Book not found.")
        return
    borrower_id, loan_date, return_date = row
    if user_id != borrower_id:
        print("You did not borrow this book.")
        return
    cur.execute("UPDATE books SET available = TRUE WHERE id = %s", (book_id,))
    return_date = str(date.today())
    cur.execute("UPDATE loans SET return_date = %s WHERE book_id = %s AND borrower_id = %s", (return_date, book_id, borr
    conn.commit()
    cur.close()
```

The **return_book()** function allows a logged-in user to return a borrowed book by prompting them to enter the loan ID of the book they wish to return. The function then checks that the requested loan exists in the **loans** table, updates the **books** table to increase the available quantity of the returned book, and removes the loan information from the **loans** table.

Note that if the user enters a non-integer loan ID value, the program will throw a **ValueError** and exit. If the requested loan ID does not exist in the **loans** table, the program will display an error message and prompt the user to try again.

Search_book():

```
# Function to search for a book by title or author
def search_book():
    search = input("Enter search term: ")
    cursor = conn.cursor()
    cursor.execute("SELECT id, title, author, genre publication_date, available FROM books WHERE title LIKE %s OR author
    results = cursor.fetchall()
    cursor.close()
    if not results:
        print("No results found")
    else:
        print("ID\tTitle\tAuthor\tPublication Date\tAvailable")
        for row in results:
            book_id, title, author, publication_date, available = row
            print(f"{book_id}\t{title}\t{author}\t{publication_date}\t{available}")
```

The **search_books()** function prompts the user to enter a search term (which can be either a book title or an author name), constructs a SQL query that searches for all books that match the search term using the **ILIKE** operator and then executes the query using the **psycopg2** library. If there are no books that match the search term, the function prints a message indicating that no books were found. Otherwise, the function prints a message indicating how many books were found, and Note that this function assumes that the **books** table in the database has columns named **title** and **author** that store the title and author name of each book, respectively

List_borrowed():

```
def list_borrowed():
    cur = conn.cursor()
    cur.execute("SELECT loans.book_id, books.title, users.username, loans.loan_date, loans.return_date FROM loans JOIN b
    rows = cur.fetchall()
    cur.close()
    if not rows:
        print("No borrowed books found.")
    else:
        print("Book ID\tTitle\tBorrower\tLoan Date\tReturn Date")
        for row in rows:
            book_id, title, borrower, loan_date, return_date = row
            print(f"{book_id}\t{title}\t{borrower}\t{loan_date}\t{return_date}")
```

The **list_borrowed()** function retrieves all loan information from the **loans** table in the database and displays it to the user in a nicely formatted table. The table shows the loan ID, borrower ID, borrower name, book ID, book title, date borrowed, and due date for each loan.

If there are no loans in the database, the program will display a message indicating that there are no loans to display.

Admin menu():

```
def admin_menu(admin_id):
    while True:
        print("What would you like to do?")
        print("1. Add Books")
        print("2. List Books")
        print("3. List borrowed books")
        print("4. List users")
        print("5. Add a user")
        print("6. Promote a user to admin")
        print("7. Demote an admin to user")
        print("8. Delete a user")
        print("9. Logout")
        choice = input("Enter your choice (1-7): ")
        if choice == "1":
            adminmenu.add_book()
        elif choice == "2":
            book.list_books()
        elif choice == "3":
            book.list_borrowed()
        elif choice == "4":
            adminmenu.list_users(admin_id)
        elif choice == "5":
            adminmenu.add_user()
        elif choice == "6":
            adminmenu.promote_user(admin_id)
        elif choice == "7":
            adminmenu.demote_admin(admin_id)
        elif choice == "8":
            adminmenu.delete_user(admin_id)
        elif choice == "9":
            print("Logged out successfully.")
            break
        else:
            print("Invalid choice. Please try again.")
```

Add_books():

```
# Add a new book to the database
def add_book():
    title = input("Enter the title of the book: ")
    author = input("Enter the author of the book: ")
    genre = input("Enter the genre of the book: ")
    publication_date = input("Enter the publication date of the book (YYYY-MM-DD): ")
    cur = conn.cursor()
    cur.execute("INSERT INTO books (title, author, genre, publication_date) VALUES (%s, %s, %s,%s)", (title, author,ge
    conn.commit()
    cur.close()
```

The **add_book()** function allows an admin user to add a new book to the database by prompting them to enter the book's title, author, genre, publication, and quantity. The function then inserts a new row into the books table with this information, setting the available quantity to the same as the total quantity (as no books have been loaned out yet).

Note that if the user enters a non-integer quantity value, the program will throw a **ValueError** and exit.

List_users():

```
# List all users (admin only)
def list_users(admin_id):
    cur = conn.cursor()
    cur.execute("SELECT id,username,role FROM users WHERE id != %s", (admin_id,))
    rows = cur.fetchall()
    cur.close()
    if not rows:
        print("No users found.")
    else:
        print("ID\tUsername\tRole")
        for row in rows:

            user_id, username, role = row
            print(f"{user_id}\t{username}\t{role}")
```

The **list_users()** function retrieves all user information from the **users** table in the database and displays it to the user in a nicely formatted table. The table shows the user ID, first name, last name, email address, and registration date for each user.

If there are no users in the database, the program will display a message indicating that there are no users to display.

Add_user():

```python
# Add a user (admin only)
def add_user():
    username = input("Enter a username: ")
    password = input("Enter a password: ")
    confirm_password = input("Confirm password: ")
    if password != confirm_password:
        print("Passwords do not match. Please try again.")
        return
    role = input("Enter the role of the user (user or admin): ")
    if role not in ["user", "admin"]:
        print("Invalid role. Please enter 'user' or 'admin'.")
        return
    cur = conn.cursor()
    cur.execute("SELECT id FROM users WHERE username = %s", (username,))
    if cur.fetchone() is not None:
        print("Username already taken. Please choose another one.")
        return
    cur.execute("INSERT INTO users (username, password, role) VALUES (%s, %s, %s)", (username, password, role))
    conn.commit()
    cur.close()
    print("User added successfully.")
```

The **add_user()** function allows an administrator to add a new user to the **users** table in the database. The function prompts the administrator to enter the user's first name, last name, email address, and password. The function then verifies that the email address is not already registered, and inserts a new row into the **users** table with the user's information, along with a generated unique user ID.

Promote_user():

```python
# Promote a user to admin (admin only)
def promote_user(admin_id):
    # if not is_admin(admin_id):
    #     print("Only administrators can promote users.")
    #     return
    user_id = input("Enter the ID of the user you want to promote to admin: ")
    cur = conn.cursor()
    cur.execute("SELECT id FROM users WHERE id = %s AND role = 'user'", (user_id,))
    if cur.fetchone() is None:
        print("Invalid user ID or user is already an admin.")
        return
    cur.execute("UPDATE users SET role = 'admin' WHERE id = %s", (user_id,))
    conn.commit()
    cur.close()
    print("User promoted to admin successfully.")
```

The **promote_user()** function allows an administrator to promote a regular user to an administrator by prompting the administrator to enter the user ID of the user to promote. The function verifies that the user ID exists in the **users** table and updates the **is_admin** field for that user to **True.**

Demote_user():

```
#user (admin only)
def demote_admin(admin_id):
    # if not is_admin(admin_id):
    #     print("Only administrators can demote other admins.")
    #     return
    user_id = input("Enter the ID of the admin you want to demote to user: ")
    cur = conn.cursor()
    cur.execute("SELECT id FROM users WHERE id = %s AND role = 'admin'", (user_id,))
    if cur.fetchone() is None:
        print("Invalid user ID or user is not an admin.")
        return
    cur.execute("UPDATE users SET role = 'user' WHERE id = %s", (user_id,))
    conn.commit()
    cur.close()
    print("Admin demoted to user successfully.")
```

The **demote_user()** function allows an administrator to demote an administrator to a regular user by prompting the administrator to enter the user ID of the user to demote. The function verifies that the user ID exists in the **users** table and updates the **is_admin** field for that user to **False.**

Delete_user():

```
#Delete a user (admin only)
def delete_user(admin_id):
    # if not is_admin(admin_id):
    #     print("Only administrators can delete users.")
    #     return
    user_id = input("Enter the ID of the user you want to delete: ")
    cur = conn.cursor()
    cur.execute("SELECT id FROM users WHERE id = %s AND role = 'user'", (user_id,))
    if cur.fetchone() is None:
        print("Invalid user ID or user is an admin.")
        return
    cur.execute("DELETE FROM users WHERE id = %s", (user_id,))
    conn.commit()
    cur.close()
    print("User deleted successfully.")
```

The **delete_user()** function allows an administrator to delete a user from the **users** table by prompting the administrator to enter the user ID of the user to delete. The function verifies that the user ID exists in the **users** table and deletes the corresponding row.

Note that for each of these functions, only administrators are allowed to use them. If a non-administrator tries to use any of these functions, the program will display an error message and exit. Also, for the **promote_user()** and **demote_user()** functions, if the administrator enters a non-integer user ID value, or if the requested user ID does not exist in the **users** table, the program will display an error message and prompt the administrator to try again. For the **delete_user()** function, if the administrator enters a non-integer user ID value, or if the requested user ID does not exist in the **users** table, the program will display an error message and exit.

Database tables:

Books :

	id [PK] integer	title character varying (100)	author character varying (100)	publication_date date	available boolean	genre character varying (255)
1	1	c programming	mark	2000-12-20	true	programming
2	2	Wings of Fire	Abdul kalam	1987-02-12	true	Biography
3	3	I am malala	malala	1998-02-12	true	Biography

The **books** table has the following columns:

- **id**: a unique identifier for each book
- **title**: the title of the book
- **author**: the author of the book
- **publication_date**: the publication date of the book
- **available**: a boolean value indicating whether the book is available for borrowing or not

Users:

	id [PK] integer	username character varying (50)	password character varying (50)	role character varying (10)
1	1	ranjith	ranjith	admin
2	2	pranav	pranav	user
3	3	rashmi	rashmi	user
4	5	abhirami	abhirami	user

The **users** table has the following columns:

- **id**: a unique identifier for each user
- **username**: the username of the user
- **password**: the password of the user
- **role**: the role of the user, which can be either "admin" or "user"

Loans:

	id [PK] integer	book_id integer	borrower_id integer	loan_date date	return_date date
1	1	1	2	2023-03-06	2023-03-06

The **loans** table has the following columns:

- **id**: a unique identifier for each loan

- **book_id**: a foreign key referencing the **id** column of the **books** table, indicating the book that was borrowed
- **borrower_id**: a foreign key referencing the **id** column of the **users** table, indicating the user who borrowed the book
- **loan_date**: the date when the book was borrowed
- **return_date**: the date when the book was returned (NULL if the book has not been returned yet)

Note that we have set up referential integrity constraints on the **book_id** and **borrower_id** columns of the **loans** table, which ensures that only valid book and user IDs can be inserted into these columns. We have also set up a cascading delete constraint on the **books** and **users** tables, which ensures that when a book or user is deleted, all corresponding loans are also deleted.

Conclusion:

this console-based library management system is a simple yet effective tool for managing books and user information. The use of PostgreSQL as the database ensures that the system can handle large amounts of data and provides scalability for future growth. The system's functionality is designed to be intuitive and easy to use for both users and administrators, and the inclusion of a search function makes it easy for users to find books of interest. Overall, this project can serve as a starting point for anyone looking to create a more complex library management system or similar project in the future.