Library Management System

AMINE AZENKOUK

202353080076

Computer Science

I. Abstract :

This Library Management System is designed to streamline and automate library operations for educational institutions, addressing the need for efficient book tracking, user management, and feedback collection while reducing manual administrative workload and improving user experience through digital access to library resources. The system is built using Flask web framework with Python for backend logic, SQLAlchemy ORM for database management with SQLite (Library.db), and responsive HTML/CSS/JavaScript for the frontend interface, implementing a role-based access control system that separates student and administrator functionalities, utilizing RESTful API endpoints for data operations, and incorporating features such as user authentication, book reservation tracking, borrowing history management, and feedback submission systems. The project successfully delivers a fully functional web-based library management platform that enables students to browse books, make reservations with time slots, view their borrowing history, submit feedback with ratings, and manage their profile settings, while administrators can efficiently manage the book inventory through CRUD operations, monitor all user reservations and borrowing patterns, review submitted feedback, update student information, and generate comprehensive reports, all data being persistently stored in a relational database with proper model relationships between Students, Books, Reservations, and Feedback entities.

II. Keywords :

- *Library Management System,*
- *Flask Web Application,*
- *Database Management,*
- *User Authentication,*
- *Book Reservation*

III. Introduction :

The primary objective of designing this Library Management System is to create a comprehensive digital solution that automates and streamlines library operations in educational institutions by enabling efficient book

inventory management, tracking student borrowing activities in real-time, facilitating seamless book reservations with time-slot scheduling, collecting user feedback for service improvement, and providing administrators with powerful tools to monitor and manage all library resources and user interactions through a centralized web-based platform that eliminates manual record-keeping, reduces administrative burden, minimizes book loss or misplacement, improves accessibility for students to browse and reserve books remotely, and generates detailed reports on borrowing patterns and library usage statistics. Traditional library management systems developed by others typically employ either desktop-based applications using technologies like Java Swing or C# Windows Forms with local database systems such as Microsoft Access or MySQL, or web-based solutions built with PHP and MySQL, ASP.NET with SQL Server, or Django with PostgreSQL, implementing standard CRUD operations for book management, barcode scanning for book identification, and basic user authentication mechanisms with role-based access control for librarians and members. However, these existing approaches face several critical problems including limited scalability when user base grows, lack of real-time synchronization across multiple access points, poor mobile responsiveness making it difficult for users to access the system from smartphones or tablets, complex deployment procedures requiring extensive server configuration and maintenance, insufficient user engagement features such as feedback systems or personalized borrowing history, outdated user interfaces that don't meet modern web design standards, and inadequate integration capabilities with other institutional systems. In this project, I adopted Flask as the lightweight yet powerful Python web framework for backend development due to its simplicity and flexibility, implemented SQLAlchemy ORM for database abstraction enabling easy database migrations and query optimization, utilized SQLite for efficient data storage with a well-structured relational schema containing Students, Books, Reservations, and Feedback tables with proper foreign key relationships, designed RESTful API endpoints for all data operations ensuring clean separation between frontend and backend logic, created a responsive user interface using HTML, CSS3, and JavaScript that adapts seamlessly to different screen sizes, implemented secure user authentication with session management distinguishing between student and administrator roles with different access privileges, developed a comprehensive book reservation system allowing students to select specific time slots for borrowing books with automatic conflict

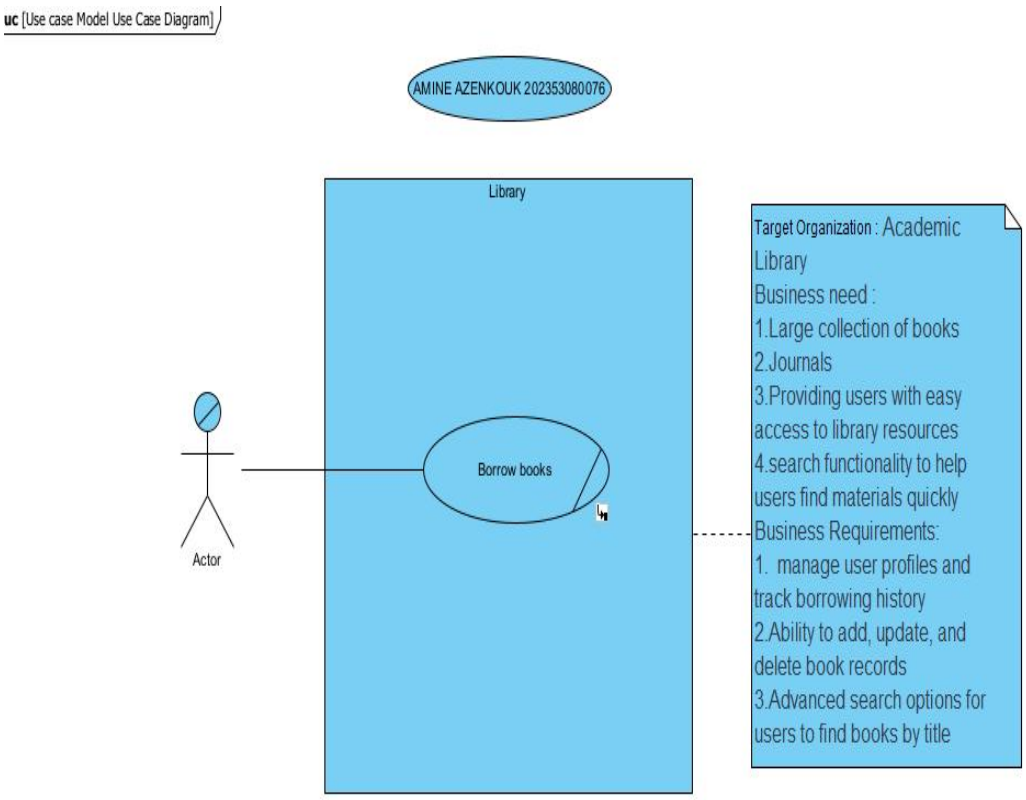
detection, built an intuitive feedback collection mechanism enabling students to rate library services and submit comments which administrators can review through a dedicated dashboard, integrated dynamic avatar generation for user profiles using UI Avatars API as fallback when custom images aren't available, and established a modular code structure with clear separation of concerns between models, routes, forms, and templates making the system easily maintainable and extensible for future enhancements.

IV. Data :

The Library Management System utilizes SQLite as its relational database with the main database file "library.db" stored in the instance directory, managed through SQLAlchemy ORM which provides an abstraction layer for converting Python objects into database tables. The database schema consists of four interconnected tables: the Student table stores user information including id, student_id, name, email, hashed password, gender, and image_path , the Book table maintains the library inventory with fields for id, book_id, title, author, publication_date, isbn, category and total_copies to track book availability in real-time; the Reservation table acts as a junction table recording all borrowing transactions with fields for id, student_id , book_id , student_name, book_title, start_time, end_time, and status, establishing many-to-many relationships between students and books, and the Feedback table stores user ratings and comments with fields for id, student_id , student_name, rating, comment, and submitted_at timestamp. Database operations follow SQLAlchemy's session management pattern using db.session.add() for insertions and db.session.commit() for saving changes, while queries utilize methods like Model.query.filter_by() for filtering records and Model.query.all() for retrieving all entries. The init_db.py script initializes the database using db.create_all() and populates it with sample data including an admin account and test student records. Student profile images are stored as physical files in the static/uploads directory with only the filename saved in the database, and when no image exists, the system generates dynamic avatars using the UI Avatars API with gender-based color coding, ensuring all data persistence and file management are handled efficiently through Flask's application context and proper error handling with transaction rollbacks when operations fail.

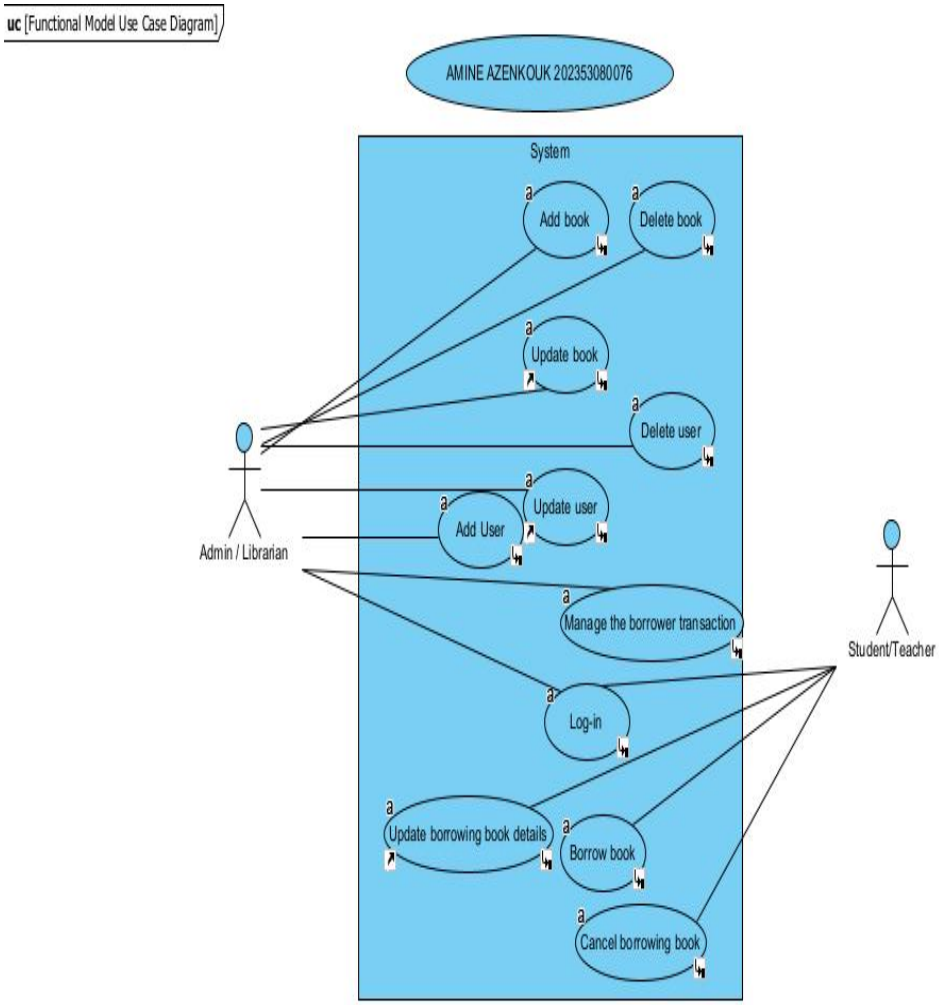
V. Overall Framework of the Program Design :

1. Use case Model Diagram :



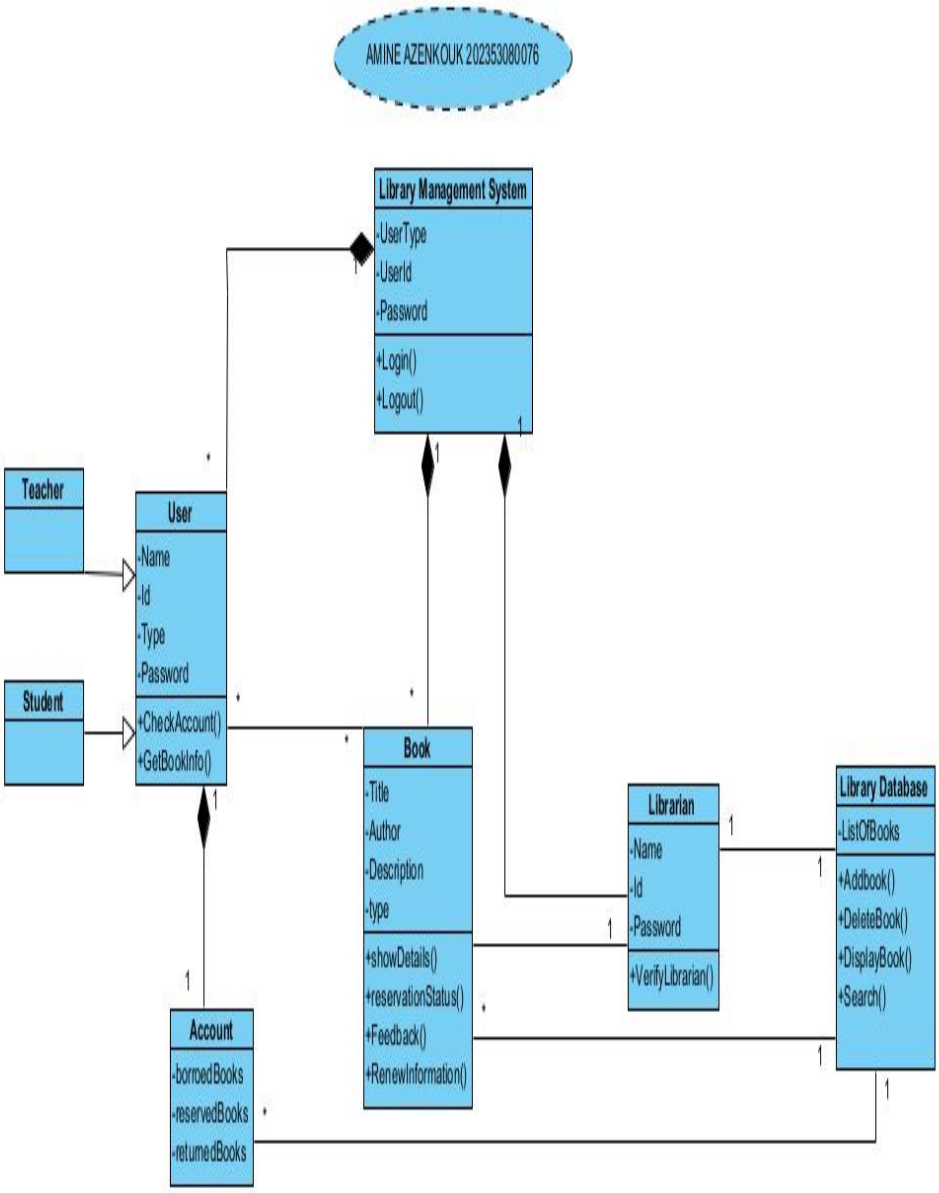
This use case model outlines the functional requirements for an academic library management system, centering on the core "Borrow books" use case. The system is designed to address key business needs such as managing a large collection of books and journals, providing users with easy digital access to resources, and enabling efficient search functionality. Key business requirements include user profile and borrowing history management, the ability to add, update, and delete book records, and advanced search options for locating materials by title. The model highlights the interaction between the system and an actor—typically a student or librarian, demonstrating how the system streamlines library operations, improves accessibility, and supports effective resource management within an educational institution.

2. Functional Model Use Case Diagram :



This functional model use case diagram defines the roles and interactions within a library management system, distinguishing between Admin/Librarian and Student/Teacher actors. The system provides administrative functions such as adding and updating books, managing user information, and handling borrowing transactions. The Admin can log in, process book borrowing, and cancel borrowing activities, ensuring operational control. Meanwhile, Student/Teacher users interact with the system primarily to borrow books or manage their borrowing status. This model highlights a clear separation of responsibilities, ensuring that administrative management and user-driven borrowing operations are systematically supported within a structured workflow.

3. Onject Model Class Diagram :



This diagram represents a Library Management System with six main components. The Library Management System class serves as the central controller, handling user authentication (Login/Logout) with UserId and Password attributes. User is the base class that both Teacher and Student inherit from, containing common attributes (Name, Id, Type, Password) and methods for account management and book information retrieval. Each user has an Account that tracks borrowed, reserved, and returned books. The Book class stores book details (Title, Author, Description, type) and provides functionality for displaying details, checking reservation status, providing feedback, and renewing books. The Librarian class has special privileges to verify other librarians and interact with the Library Database, which maintains the list of books and provides CRUD operations (Add, Delete, Display books) and search

functionality. The system establishes relationships where multiple users can interact with multiple books, librarians manage the database, and all components connect through the central Library Management System.

VI. Program code :

1. Models.py

```
from library import db
from datetime import datetime

class Student(db.Model):

    name = db.Column(db.String(length=30), nullable=False )
    id = db.Column(db.Integer(), primary_key=True )
    passport = db.Column(db.Integer(), nullable=False , unique=True )
    classs = db.Column(db.String(length=30), nullable=False )
    type = db.Column(db.String(length=30), nullable=False )
    gender = db.Column(db.String(length=30), nullable=False )
    password = db.Column(db.String(length=30), nullable=False )


    def __repr__(self) :
        return f'Student {self.name}'

class Book(db.Model):

    id = db.Column(db.Integer(), primary_key=True )
    title = db.Column(db.String(length=30), nullable=False)
    author = db.Column(db.String(length=30), nullable=False)
    type = db.Column(db.String(length=30), nullable=False)
    description = db.Column(db.String(length=80), nullable=False)
    image_path = db.Column(db.String(200))

    def __repr__(self) :
        return f'Book {self.title}'
```

```
class Reservation(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    book_id = db.Column(db.Integer(), db.ForeignKey('book.id'),
nullable=False)
    student_id = db.Column(db.Integer(), db.ForeignKey('student.id'),
nullable=False)
    start_time = db.Column(db.DateTime, nullable=False,
default=datetime.utcnow)
    end_time = db.Column(db.DateTime, nullable=False)
```

```
    book = db.relationship('Book', backref='reservations')
    student = db.relationship('Student', backref='reservations')

    def __repr__(self):
        return f'Reservation {self.id}'
```

```
class Feedback(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    student_id = db.Column(db.Integer(), db.ForeignKey('student.id'),
nullable=False)
    rating = db.Column(db.Integer(), nullable=False)
    feedback_text = db.Column(db.Text, nullable=False)
    created_at = db.Column(db.DateTime, nullable=False,
default=datetime.utcnow)


    student = db.relationship('Student', backref='feedbacks')

    def __repr__(self):
        return f'Feedback {self.id}'
```

Explanation : *This file defines the database schema for a library management system using SQLAlchemy ORM, consisting of four interconnected models. The Student model stores user information including credentials, identification details (passport number), class, user type (Admin or regular student), and gender. The Book model maintains the library catalog with attributes like title, author, type, description, and an image path for book covers. The Reservation model establishes a many-to-many relationship between students and books, tracking when books are borrowed (start_time) and when they're due (end_time), with foreign key relationships linking back to both Student and Book tables. Finally, the Feedback model allows students to submit ratings and written feedback about the library system, storing a timestamp for when each feedback was created, and it's linked to the Student model through a foreign key relationship to track who submitted each review.*

2. Routes.py :

```
from library import app
from flask import
render_template,request,redirect,url_for,session,jsonify
from library.models import Student,Book,Reservation,Feedback
from library.forms import RegisterForm
from library import db
import os
```

```
from werkzeug.utils import secure_filename
from datetime import datetime




@app.route('/books')
def main_page_books():
    books = Book.query.all()
    return render_template('main3.html', books=books)
```

```
@app.route('/borrowing-history')
def book_history():
    return render_template('bookHistoryUser.html')
```

```
@app.route('/feedback')
def feedback_page():
    return render_template('feedbackuser.html')
```

```
@app.route('/account')
def account_settings():

    user_data = {
        'id': session.get('user_id'),
        'name': session.get('user_name'),
        'type': session.get('user_type'),
        'gender': session.get('user_gender'),
        'passport': session.get('user_passport'),
        'classs': session.get('user_class')
    }
    return render_template('accountSettings.html', user=user_data)
```

```
@app.route('/api/books')
def get_books_api():
    books = Book.query.all()
    books_list = [{
        'id': book.id,
        'title': book.title,
        'author': book.author,
        'type': book.type,
        'description': book.description,
        'image': f'/static/uploads/{book.image_path}' if
book.image_path else ''
    } for book in books]
    return {'books': books_list}
```

```
@app.route('/book/<int:book_id>')
def book_details(book_id):
    book = Book.query.get_or_404(book_id)
```

```
    return render_template('book_details.html', book=book)
```

```
@app.route('/User_Data')
def user_data():
    students=Student.query.all()
    return render_template('usersData.html' , students=students)
```

```
@app.route('/Adminmain')
def Admin_page():
    return render_template('Adminmain.html')
```

```
@app.route('/Books_data')
def books_history_page():
    books=Book.query.all()
    print("Debug - Books found:", books)
    print("ss")
    print("Number of books:", len(books))
    for i, book in enumerate(books, 1):
        print(f"Book {i}: {book.title} | Image Path: {book.image_path}")
    return render_template('bookHistory.html', books=books)
```

```
@app.route('/add book', methods=['GET', 'POST'])
def Add_book_page():
    if request.method == 'POST':
        image_path = None

        if 'image' in request.files:
            image = request.files['image']
            if image.filename != '':

                upload_dir = os.path.join('library','static', 'uploads')
                os.makedirs(upload_dir, exist_ok=True)


                filename = secure_filename(image.filename)
                save_path = os.path.join(upload_dir, filename)
                image.save(save_path)


                image_path = filename

        new_book = Book(
            title=request.form.get('title'),
            author=request.form.get('author'),
            type=request.form.get('type'),
            description=request.form.get('description'),
            image_path=image_path
```

```
        )
        db.session.add(new_book)
        db.session.commit()
        return redirect(url_for('books_history_page'))

    return render_template('addBook.html')
```

```
@app.route('/User_data', methods=['GET', 'POST'])
def register():
    form = RegisterForm()
    if request.method == 'POST' :
        new_student = Student(
            name=request.form.get('Name'),
            id=request.form.get('ID'),
            passport=request.form.get('PassportID'),
            classs=request.form.get('class'),
            type=request.form.get('Type'),
            gender=request.form.get('Gender'),
            password="123456789"
        )
        db.session.add(new_student)
        db.session.commit()
        return redirect(url_for('user_data' ,_external=True))

    return render_template('usersData.html' ,form=form)
```

```
@app.route('/<int:id>/update', methods=['GET','POST'])
def update(id):
    students = Student.query.filter_by(id=id).first()
    if request.method == 'POST':

        students.name = request.form.get('Name')
        students.id=request.form.get('ID')
        students.passport = request.form.get('PassportID')
        students.classs = request.form.get('class')
        students.type = request.form.get('Type')
        students.gender = request.form.get('Gender')

        db.session.commit()

        return redirect(url_for('user_data'))

    return render_template('updateStudent.html', students=students)
```

```
@app.route('/<int:id>/delete', methods=['POST'])
def delete(id):
    students = Student.query.get_or_404(id)
    db.session.delete(students)
    db.session.commit()
    return redirect(url_for('user_data'))
```

```
@app.route('/book/<int:id>/update', methods=['GET', 'POST'])
def update_book(id):
    book = Book.query.filter_by(id=id).first()
    if request.method == 'POST':
        book.title = request.form.get('title')
        book.author = request.form.get('author')
        book.type = request.form.get('type')
        book.description = request.form.get('description')


        if 'image' in request.files:
            image = request.files['image']
            if image.filename != '':

                upload_dir = os.path.join('library', 'static',
'uploads')
                os.makedirs(upload_dir, exist_ok=True)


                filename = secure_filename(image.filename)
                save_path = os.path.join(upload_dir, filename)
                image.save(save_path)


                book.image_path = filename

        db.session.commit()
        return redirect(url_for('books_history_page'))
    return render_template('addBook.html', book=book)
```

```
@app.route('/book/<int:id>/delete', methods=['POST'])
def delete_book(id):
    book = Book.query.get_or_404(id)
```

```
    if book.image_path:
        try:
            os.remove(os.path.join('static', 'uploads',
book.image_path))
        except:
```

```
            pass

    db.session.delete(book)
    db.session.commit()
    return redirect(url_for('books_history_page'))
```

```
@app.route('/')
def home():
    return redirect(url_for('login'))
```

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        student_id = request.form.get('id')
        password = request.form.get('password')


        student = Student.query.filter_by(id=student_id).first()

        if student and student.password == password:

            session['user_id'] = student.id
            session['user_name'] = student.name
            session['user_type'] = student.type
            session['user_gender'] = student.gender
            session['user_passport'] = student.passport
            session['user_class'] = student.classs


            if student.type == 'Admin':
                return redirect(url_for('Admin_page'))
            else:
                return redirect(url_for('main_page_books'))
        else:

            return render_template('login.html', error='Invalid ID or
password')

    return render_template('login.html')
```

```
@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))
```

```
@app.route('/admin/reservations')
def admin_reservations():
    return render_template('adminStudentsHistoryBooks.html')
```

```python
@app.route('/api/reservations')
def get_reservations():
    reservations = Reservation.query.all()
    reservations_list = [{
        'id': res.id,
        'book_title': res.book.title,
        'book_author': res.book.author,
        'book_image': f'/static/uploads/{res.book.image_path}' if
res.book.image_path else '',
        'student_name': res.student.name,
        'student_id': res.student.id,
        'start_time': res.start_time.strftime('%Y-%m-%d %H:%M:%S'),
        'end_time': res.end_time.strftime('%Y-%m-%d %H:%M:%S')
    } for res in reservations]
    return jsonify({'reservations': reservations_list})
```

```python
@app.route('/admin/feedbacks')
def admin_feedbacks():
    return render_template('submittedFeedBacks.html')
```

```python
@app.route('/api/feedbacks')
def get_feedbacks():
    feedbacks = Feedback.query.all()
    feedbacks_list = [{
        'id': fb.id,
        'username': fb.student.name,
        'userId': fb.student.id,
        'rating': fb.rating,
        'feedback': fb.feedback_text,
        'image': f'https://ui-
avatars.com/api/?name={fb.student.name.replace(" ",
"+")}&background={"4A90E2" if fb.student.gender == "Male" else
"E91E63"}&color=fff&size=200',
        'created_at': fb.created_at.strftime('%Y-%m-%d %H:%M:%S')
    } for fb in feedbacks]
    return jsonify({'feedbacks': feedbacks_list})
```

```python
@app.route('/api/submit-feedback', methods=['POST'])
def submit_feedback():
    if 'user_id' not in session:
        return jsonify({'error': 'Not logged in'}), 401

    data = request.get_json()
    new_feedback = Feedback(
        student_id=session['user_id'],
        rating=data.get('rating'),
```

```
        feedback_text=data.get('feedback')
    )
    db.session.add(new_feedback)
    db.session.commit()
    return jsonify({'message': 'Feedback submitted successfully'}), 201
```

```
@app.route('/api/reserve-book', methods=['POST'])
def reserve_book():
    if 'user_id' not in session:
        return jsonify({'error': 'Not logged in'}), 401

    data = request.get_json()
```

```
    start_time_str = data.get('start_time')
    end_time_str = data.get('end_time')


    try:
        start_time = datetime.strptime(start_time_str, '%Y-%m-
%d %H:%M:%S')
    except ValueError:
        start_time = datetime.strptime(start_time_str, '%Y-%m-%dT%H:%M')

    try:
        end_time = datetime.strptime(end_time_str, '%Y-%m-%d %H:%M:%S')
    except ValueError:
        end_time = datetime.strptime(end_time_str, '%Y-%m-%dT%H:%M')

    new_reservation = Reservation(
        book_id=data.get('book_id'),
        student_id=session['user_id'],
        start_time=start_time,
        end_time=end_time
    )
    db.session.add(new_reservation)
    db.session.commit()
    return jsonify({'message': 'Book reserved successfully'}), 201
```

Explanation :*This file defines all the Flask route handlers for the library management system, implementing a comprehensive web application with separate interfaces for admins and regular users. The routes handle user authentication through a login system that stores session data and redirects admins to an admin panel while directing regular users to the book browsing page. Admin functionality includes CRUD operations for managing books (with image upload capabilities), managing student accounts, viewing all book reservations, and reviewing*

user feedback submissions. Regular user functionality includes browsing available books, viewing book details, making book reservations with start and end times, submitting feedback with ratings, and managing their account settings. The application uses both traditional HTML template rendering for page views and RESTful API endpoints (prefixed with /api/) that return JSON data for dynamic content loading, with session-based authentication protecting sensitive operations and ensuring proper authorization for admin-only features.

3. _init_.py :

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__, template_folder='template')
app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///library.db'
app.config['SECRET_KEY']= '6586dd966221826fe6b1659c'
db =SQLAlchemy(app)
app.app_context().push()

from library import routes
```

Explanation : *This initialization file bootstraps the Flask library management application by creating and configuring the core application components. It instantiates a Flask app with a custom template folder named 'template', configures SQLAlchemy to use an SQLite database file (library.db) for data persistence, and sets a secret key necessary for secure session management and cookie encryption. The file creates a SQLAlchemy database instance (db) bound to the Flask app, then pushes the application context to make it available throughout the application lifecycle, which is essential for database operations outside of request handlers. Finally, it imports the routes module at the end to register all the URL endpoints with the application, following Flask's application factory pattern where the app instance must be created before importing route definitions to avoid circular import issues.*

4. Run.py :

```
from library import app, db
from library.models import Student

with app.app_context():
    db.create_all()


    test_admin = Student.query.filter_by(id=1).first()
    if not test_admin:
        test_admin = Student(
            id=1,
```

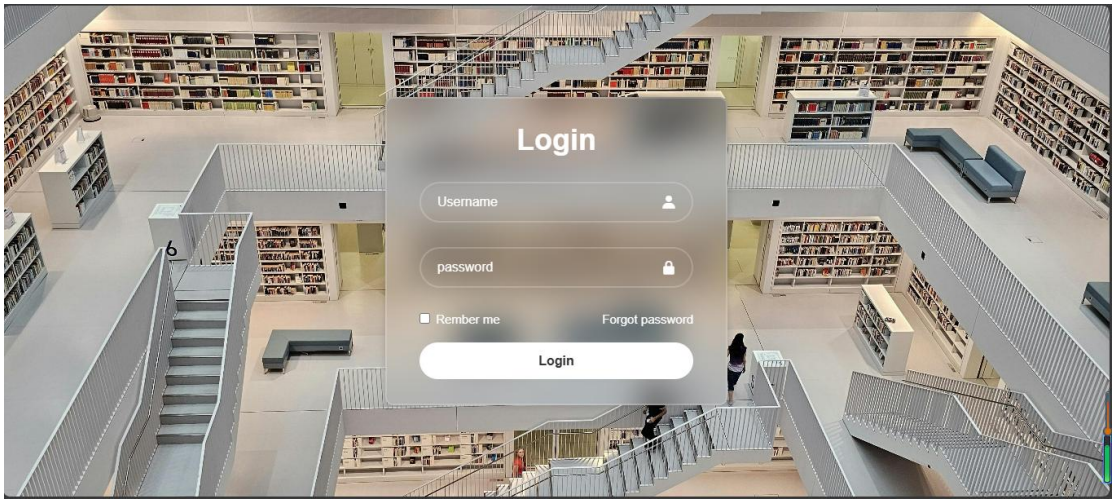
```
            name='Admin User',
            passport='ADMIN001',
            classs='N/A',
            type='Admin',
            gender='Male',
            password='admin123'
        )
        db.session.add(test_admin)
        db.session.commit()
        print("Test admin created: ID='1', Password='admin123'")


    students = Student.query.all()
    print("\n=== Current Users in Database ===")
    for student in students:
        print(f"ID: {student.id}, Name: {student.name}, Type:
{student.type}, Password: {student.password}")
    print("================================\n")
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Explanation: *This Flask application entry point initializes the database and starts the development server. It creates all necessary database tables using SQLAlchemy's create_all() method, then checks if a default admin user exists in the database—if not, it creates one with ID '1', username 'Admin User', and password 'admin123' for initial system access. The script also includes debugging functionality that prints all registered users in the database with their credentials when the application starts, making it easier to verify the database state during development. Finally, if executed directly, it launches the Flask development server with debug mode enabled for hot-reloading and detailed error messages.*

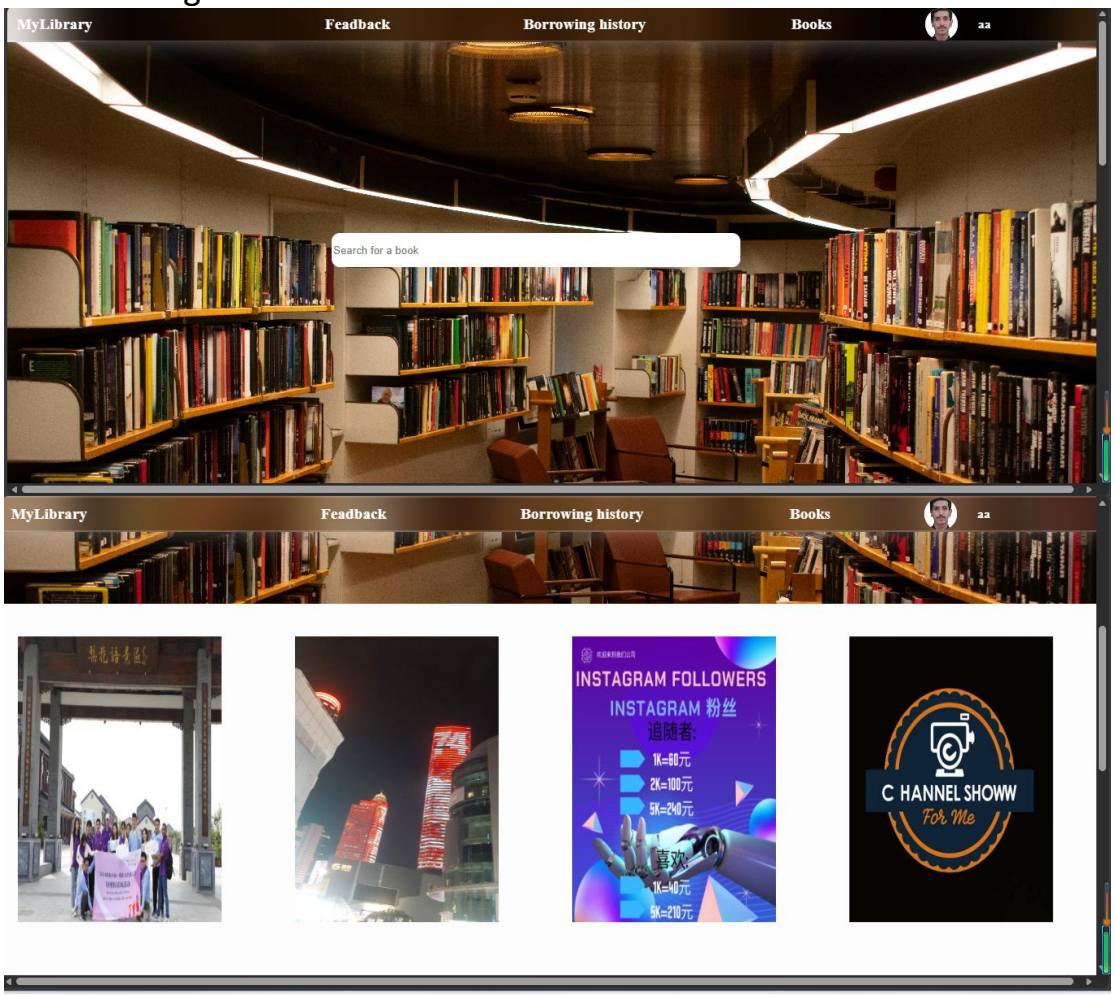
VII. Basic Operation of the Program :

First we have Login page where the user can enter his Id and password

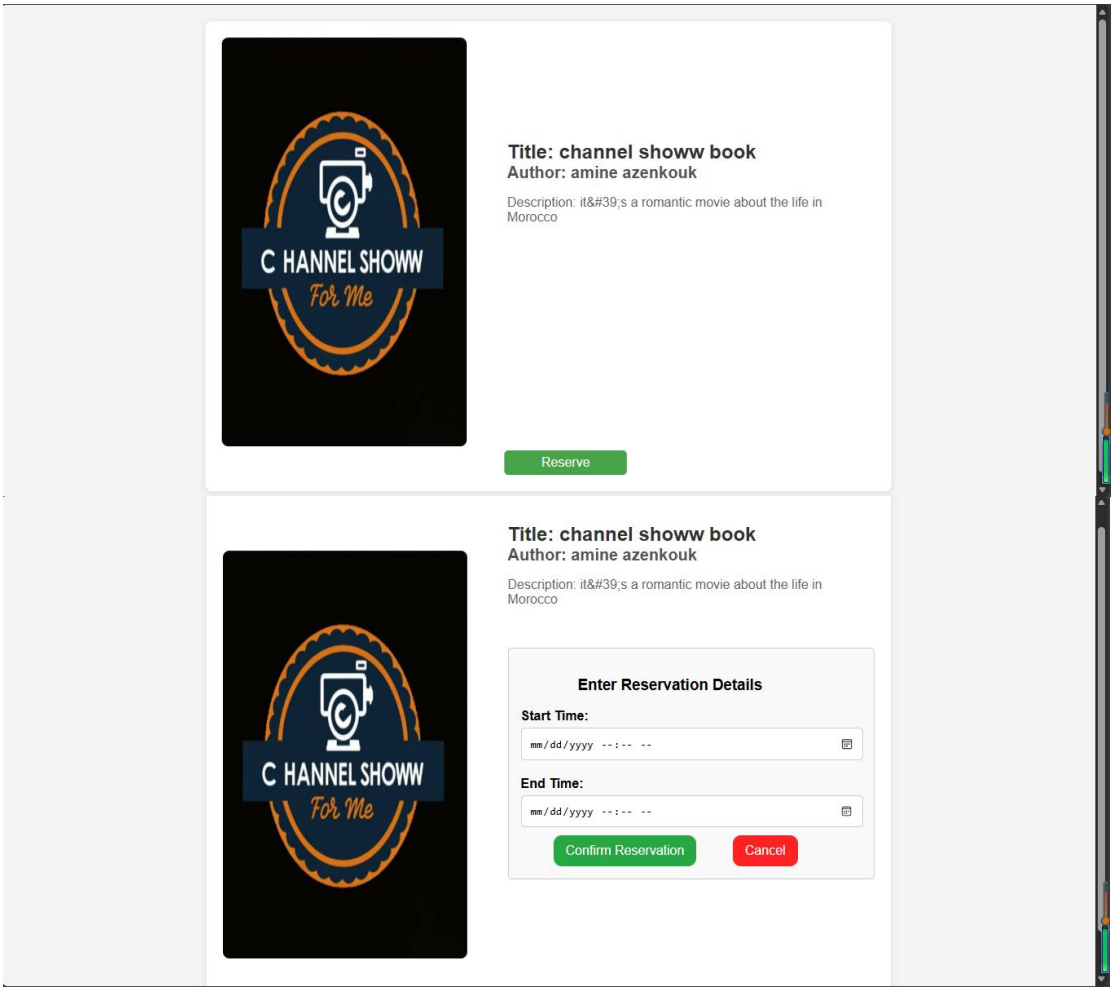


After he click the login button the system will check if he is a student or a teacher to direct him to the library main page otherwise if he is a librarian then he will direct him to the admin page .

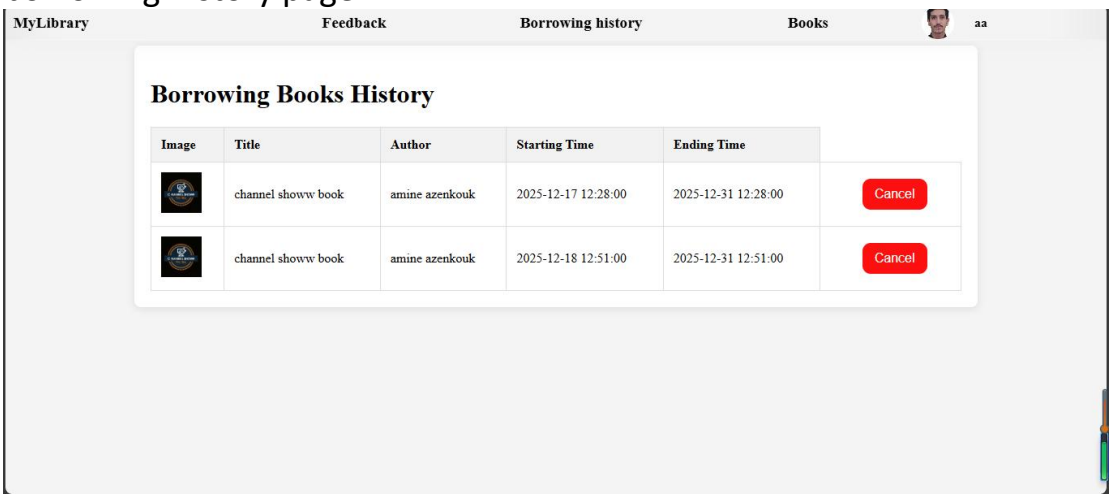
Now let's log in with a student details :



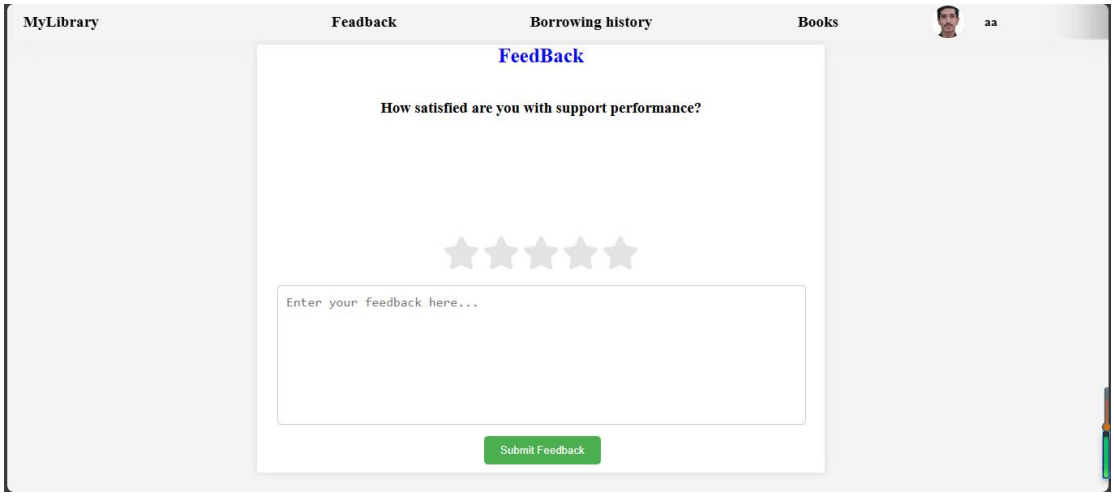
This is the main page where the student can search or choose a book to reserve . and when you click a book it will show you it's details and then it will show you how to book it :



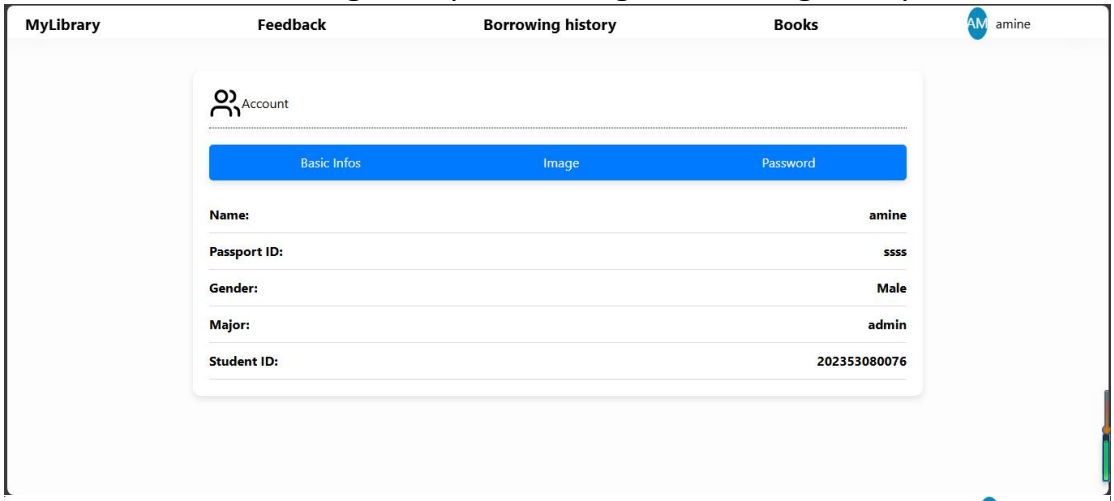
After you confirm the reservation you can find the details in the borrowing history page :

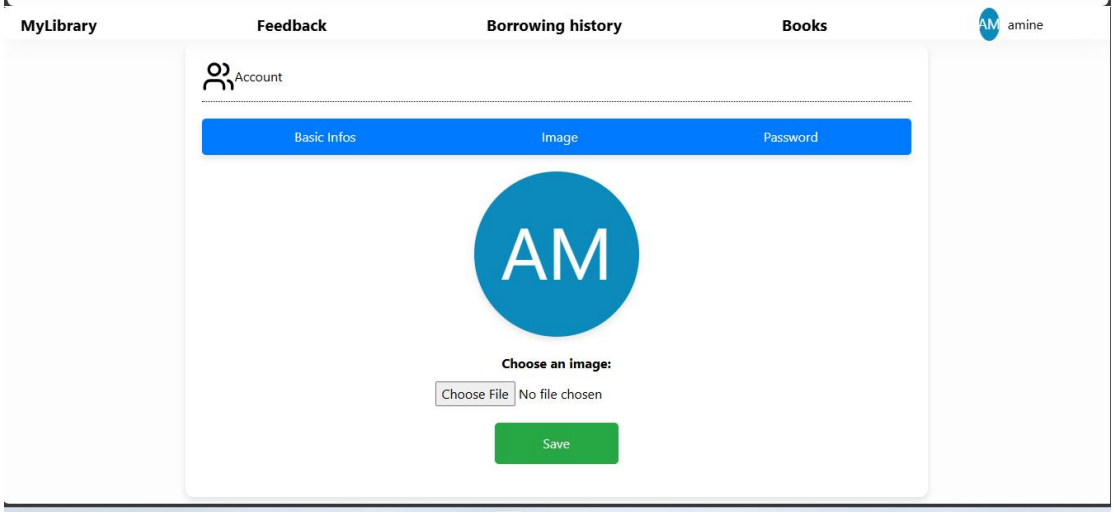


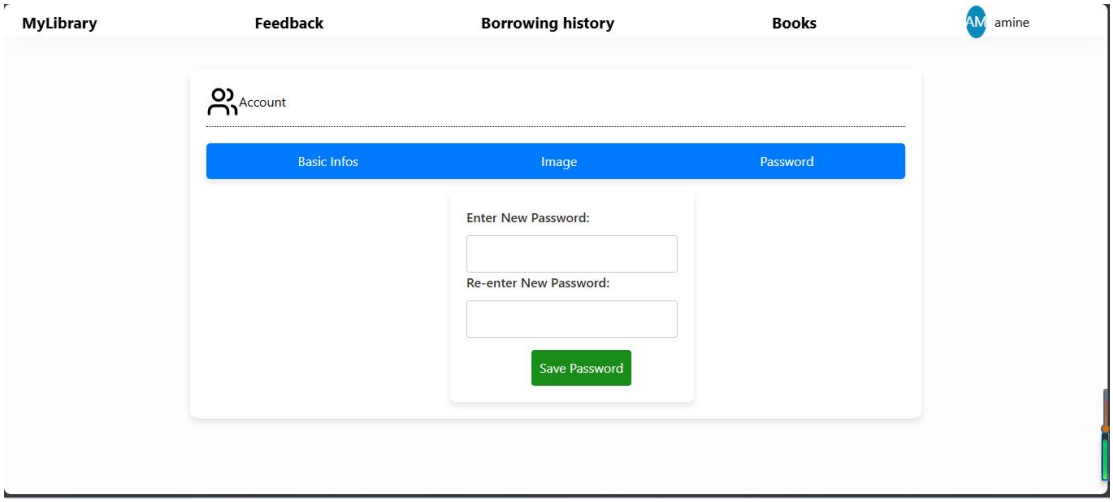
Also there is a feedback page where the user can send his feedback to the responsible and give him a rate :



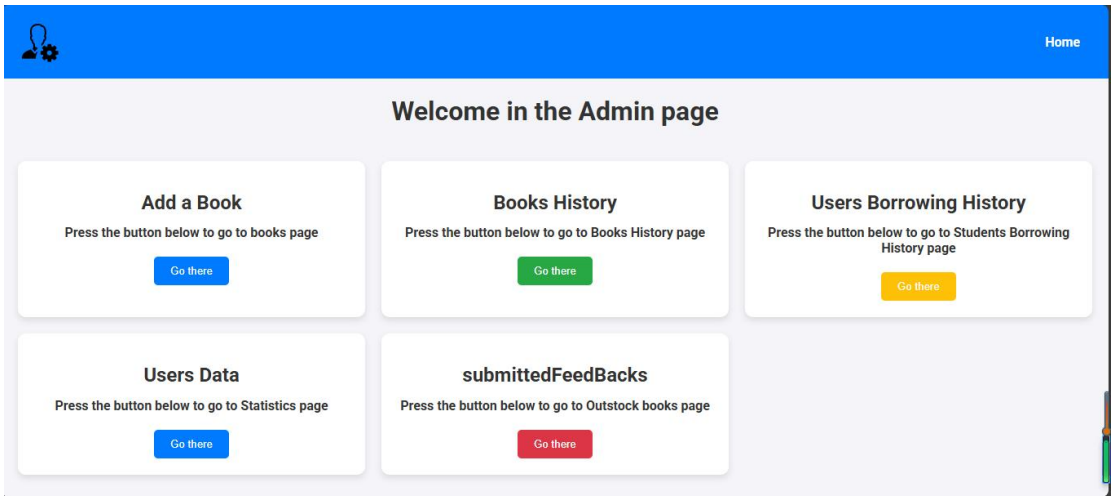
There is also a Settings page where the user can check his personnel informations and change the profile image and change the password :



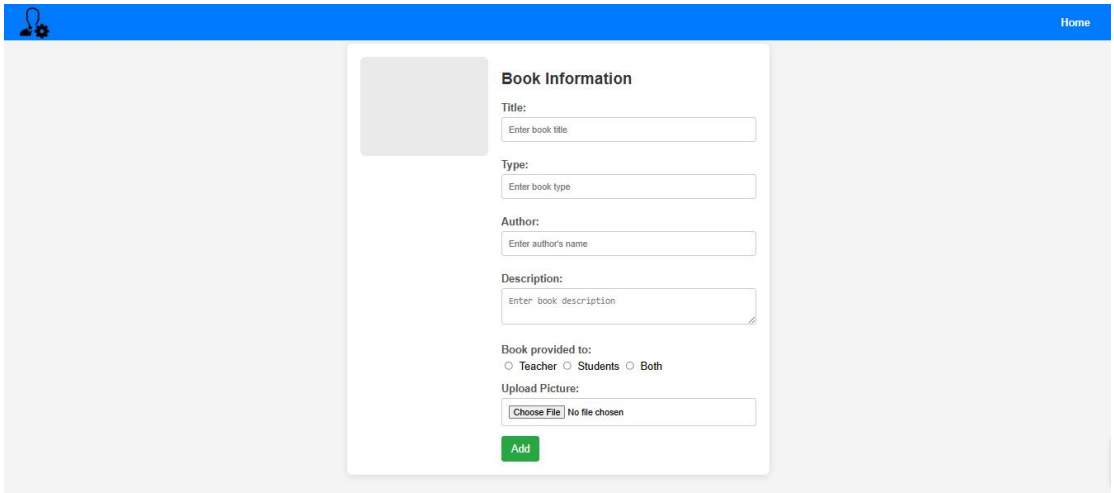




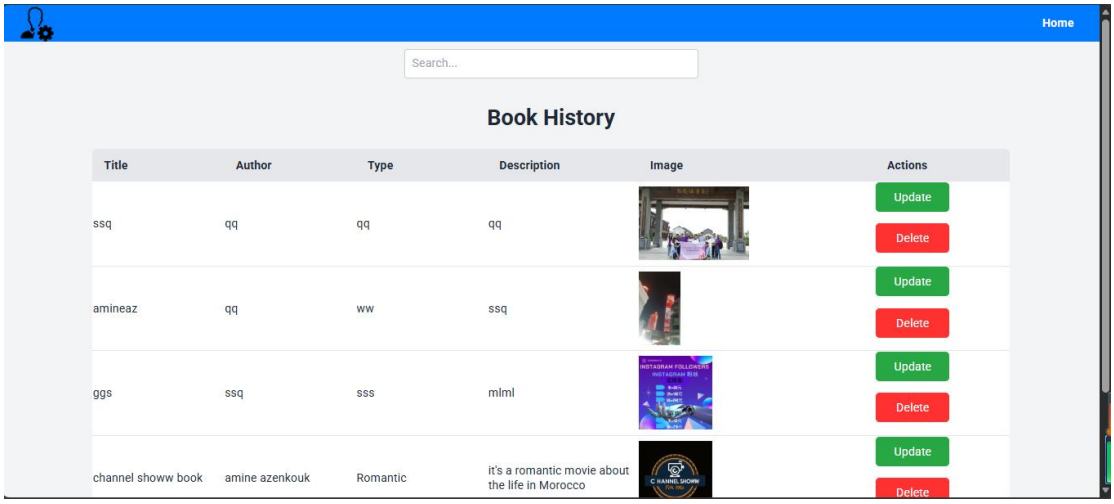
All the above images is for the student and teacher interfaces . now let's move to the admin interfaces :



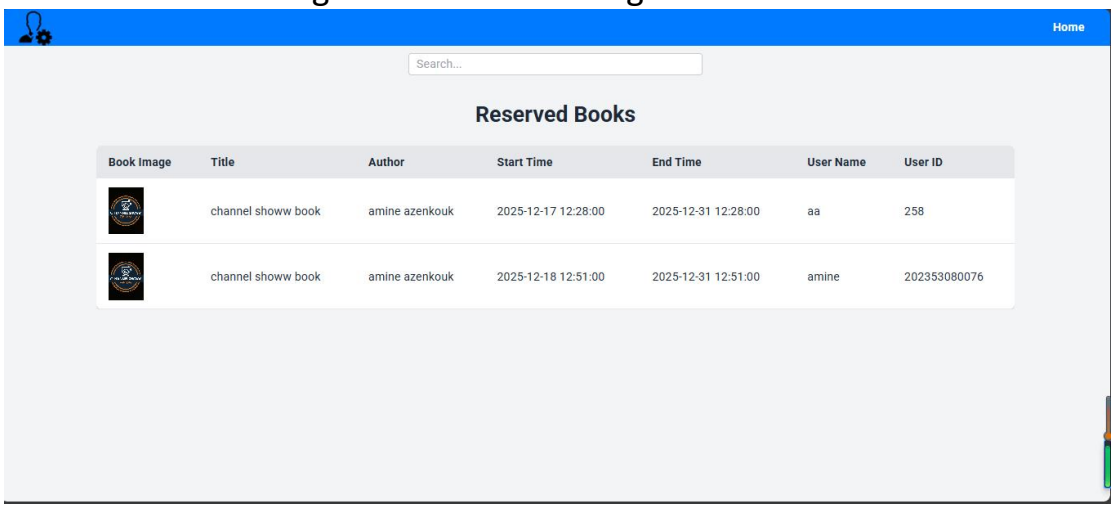
Here the admin has a lot of functions like add a book :



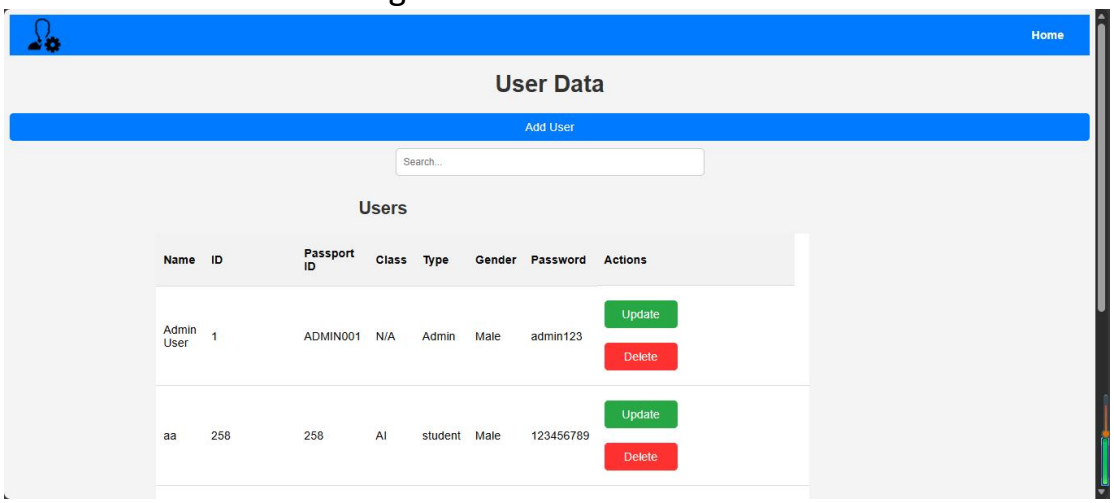
After he added a book it will be displayed to the user interface and to the books history in the admin part :

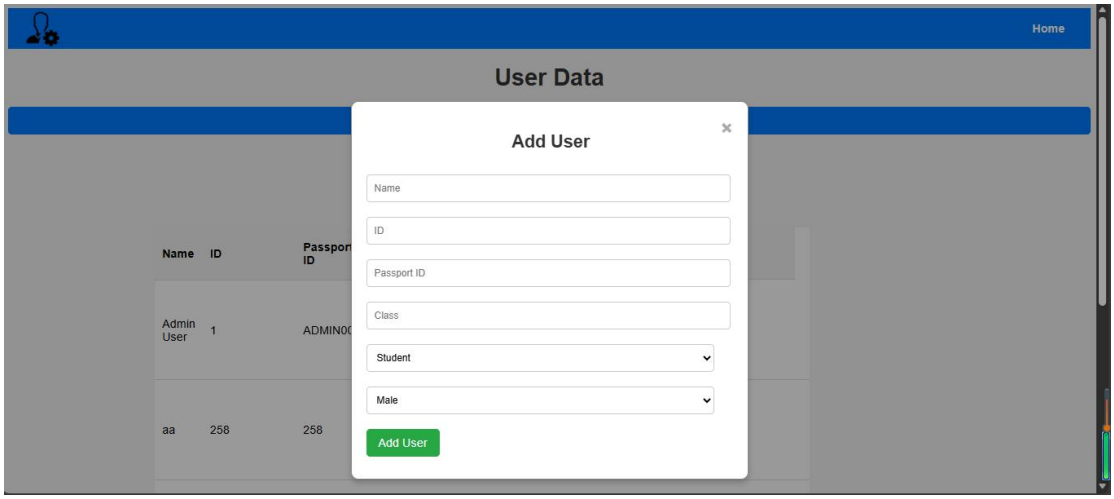


The admin also can see the borrowing books details like who reserved the book and starting time and the ending time :

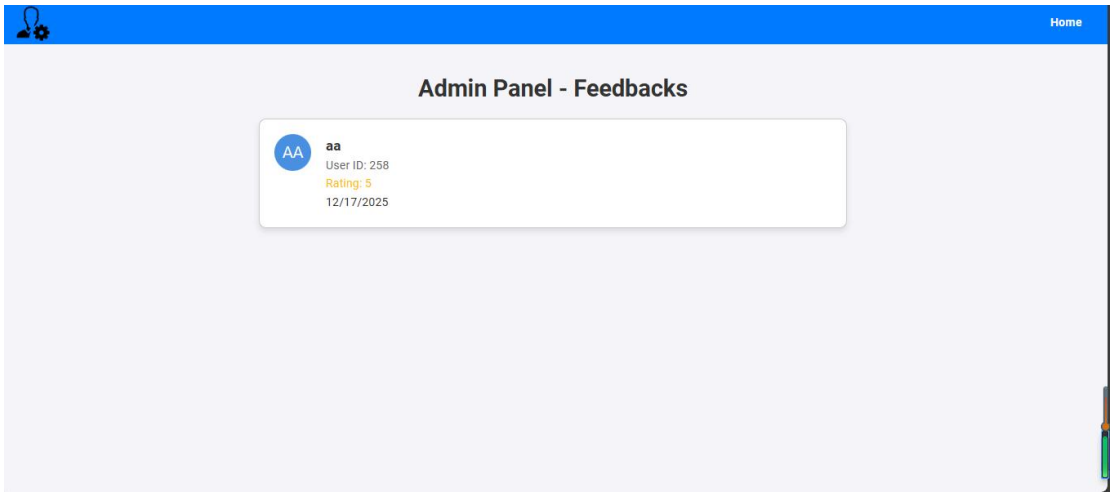


The admin also can manage the user data :





He can also see the feedbacks submitted from the students and teachers :



VIII. Design Insights :

Throughout the development of this library management system, I gained valuable insights into full-stack web development and database design. One of the most important lessons was understanding the importance of proper session management and user authentication - initially, I struggled with keeping users logged in across different pages, but implementing Flask's session system taught me how web applications maintain state.

-Working with SQLAlchemy's ORM was eye-opening. Instead of writing raw SQL queries, I learned how to think in terms of Python objects and relationships. Creating the foreign key relationships between Students, Books, and Reservations helped me understand database normalization and how to structure data efficiently. The many-to-many relationship through the Reservation table was particularly challenging but rewarding once I got it working.

One major challenge was handling file uploads for book covers and user profile images. I learned that storing images requires careful consideration of file paths, security validation, and proper folder structure. The decision to store images in a static/uploads directory and save only the filename in the database proved to be a clean solution.

The separation of concerns between admin and regular user interfaces taught me about role-based access control. Initially, I had everything in one interface, but splitting the routes and creating different dashboards made the application much more maintainable and secure. Implementing checks to ensure only admins could access certain endpoints was crucial for security.

JavaScript integration for dynamic content loading was another learning curve. Using fetch API calls to load data without page refreshes made the application feel more modern and responsive. The localStorage implementation for profile images showed me how client-side storage can enhance user experience, though I encountered bugs with inconsistent key naming that taught me the importance of standardization.

Overall, this project reinforced that good planning and consistent coding conventions prevent many headaches later in development.

References :

1. *Library management subsystem1 it is a paradigm file where I have all my diagrams .*
2. *Flask Web Development: Building Web Applications with Python : This book covers Flask fundamentals, session management, authentication, file uploads, and REST API development*
3. *Designing RESTful Web APIs with Flask: Best Practices for API Development : Discusses RESTful endpoint design, JSON responses, and stateless authentication*