4710 BOOKSTORE

COP 4710 FINAL PROJECT

Project Members:

- Anthony Leonel Carvalho (Leader)
- Alejandro Valverde
- Diego Munoz

OVERVIEW

This report outlines the design and core functionalities of our web-based book search website, **4710 Bookstore**. The website allows users to search for books by various criteria including title, genre, and rating. It displays concise information about each book, such as its image and title, and allows users to add books to their watchlist for future reference.

TECHNOLOGY STACK

- **Backend Framework:** Flask
- **Database:** SQLAlchemy with MySQL
- **Frontend:** HTML, CSS, and Flask Templates
- **Authentication:** User registration and login

KEY FEATURES AND FUNCTIONALITIES

1. **Database Management:** The backend is built using Flask, a lightweight Python web framework, which facilitates communication between the frontend interface and the backend logic. SQLAlchemy is employed for database interactions, allowing for efficient and seamless management of book-related data. The system includes functionalities to:
   - **Upload and Display Books:** Books can be added to the database via JSON file uploads. This feature ensures that bulk book data can be integrated into the system quickly.
   - **Search Books:** Users can search the database for books using various criteria. This search functionality helps users find specific books or explore available titles.
   - **Watchlist Functionality:** Users have the ability to add books to their personal watchlist. This feature allows users to track books of interest and easily access them later.
2. **User and Admin Features:**
   - **User Interaction:** Registered users can view book details, search for books, and manage their watchlist. This enhances the browsing experience and personalizes user engagement.
   - **Admin Panel:** Admins have access to an administrative interface where they can perform several critical tasks:
     - **View All Users and Books:** Admins can view lists of all users and books in the system. This includes details such as the books users have added to their watchlists.
     - **Manage Users and Books:** Admins can delete users and books as needed. They can also upload new books to the database using JSON files.

        - **Monitor User Activity:** Admins can track user interactions, including which books are being watched by users, providing insights into user preferences and system usage.

APPLICATION STRUCTURE AND WORKFLOW

- **Frontend Integration:** The user interface is designed to be intuitive and responsive, providing a seamless experience across different devices. It includes pages for book search, user watchlists, and administrative functions.
- **Backend Operations:** Flask routes handle various requests such as displaying the admin panel, adding books, deleting books, and managing user data. Each route is associated with specific functionalities and ensures that operations are executed securely and efficiently.
- **Database Interactions:** SQLAlchemy is used to manage database transactions, ensuring data consistency and integrity. Operations such as querying for books, adding new records, and deleting entries are handled through SQLAlchemy models and sessions.

TECHNICAL SPECIFICATIONS

- **Framework and Tools:**
    - **Flask:** For developing the backend and handling web requests.
    - **SQLAlchemy:** For ORM-based database interactions.
    - **JSON:** For bulk uploading book data.
- **User Interface:** Designed for ease of use, with features like search bars, watchlist management, and administrative controls.
- **Security Measures:** Authentication and authorization are implemented to protect user data and ensure that only authorized users (admins) can access certain functionalities.

CONCLUSION

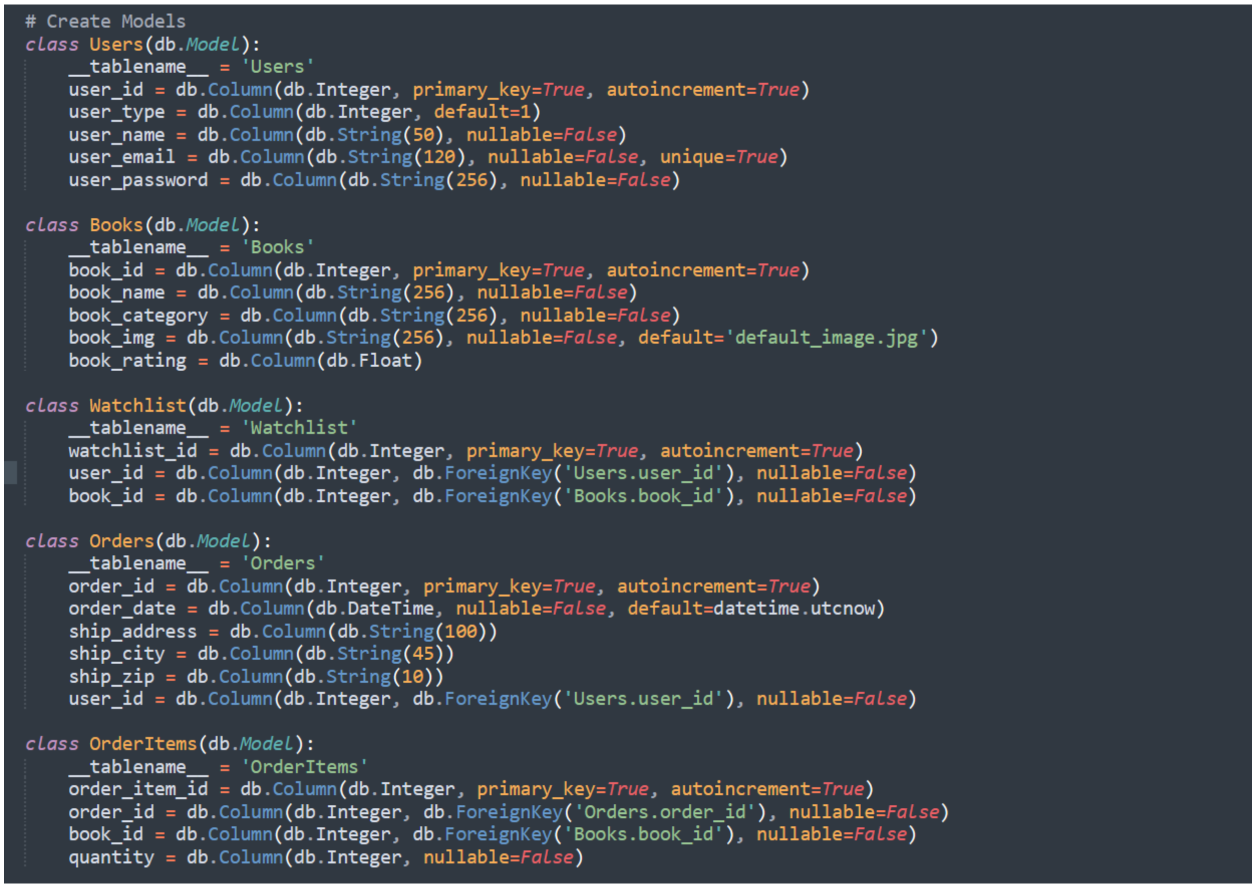
The Online Bookstore System provides a robust platform for book management, with features tailored to both end-users and administrators. It leverages Flask and SQLAlchemy to offer a dynamic and scalable solution for book-related operations, enhancing both user experience and administrative oversight.

NEXT PAGE DOCUMENTATION.

DOCUMENTATION:

DATABASE MODELS

Description: The database models define the structure of the database tables used to store information about users, books, watchlists, orders, and order items.



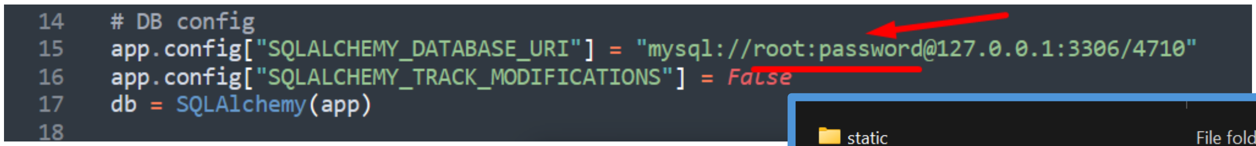
```python
# Create Models
class Users(db.Model):
    __tablename__ = 'Users'
    user_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    user_type = db.Column(db.Integer, default=1)
    user_name = db.Column(db.String(50), nullable=False)
    user_email = db.Column(db.String(120), nullable=False, unique=True)
    user_password = db.Column(db.String(256), nullable=False)

class Books(db.Model):
    __tablename__ = 'Books'
    book_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    book_name = db.Column(db.String(256), nullable=False)
    book_category = db.Column(db.String(256), nullable=False)
    book_img = db.Column(db.String(256), nullable=False, default='default_image.jpg')
    book_rating = db.Column(db.Float)

class Watchlist(db.Model):
    __tablename__ = 'Watchlist'
    watchlist_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    user_id = db.Column(db.Integer, db.ForeignKey('Users.user_id'), nullable=False)
    book_id = db.Column(db.Integer, db.ForeignKey('Books.book_id'), nullable=False)

class Orders(db.Model):
    __tablename__ = 'Orders'
    order_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    order_date = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    ship_address = db.Column(db.String(100))
    ship_city = db.Column(db.String(45))
    ship_zip = db.Column(db.String(10))
    user_id = db.Column(db.Integer, db.ForeignKey('Users.user_id'), nullable=False)

class OrderItems(db.Model):
    __tablename__ = 'OrderItems'
    order_item_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    order_id = db.Column(db.Integer, db.ForeignKey('Orders.order_id'), nullable=False)
    book_id = db.Column(db.Integer, db.ForeignKey('Books.book_id'), nullable=False)
    quantity = db.Column(db.Integer, nullable=False)
```

SETTING UP DATABASE

SET-UP INFO / RUN

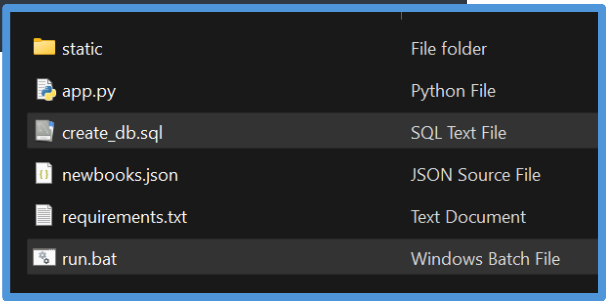
SETTING UP DATABASE

To setup-database, create a database named 4710 using the sql schema given on the zipped package (create_db.sql), and update the database root and password to your config on line 15 of the code.



```python
14   # DB config
15   app.config["SQLALCHEMY_DATABASE_URI"] = "mysql://root:password@127.0.0.1:3306/4710"
16   app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
17   db = SQLAlchemy(app)
18
```

Running the App: To run, simply run run.bat
The run.bat, will install all needed libraries and run the app.

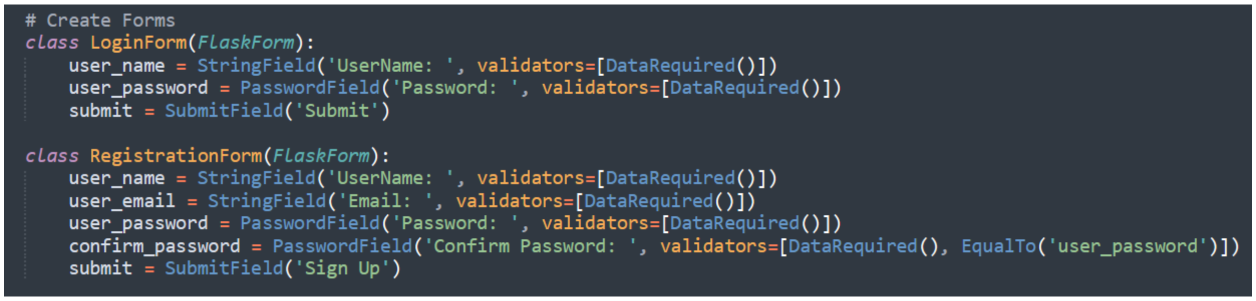


static	File folder
app.py	Python File
create_db.sql	SQL Text File
newbooks.json	JSON Source File
requirements.txt	Text Document
run.bat	Windows Batch File

FORMS:

SETTING UP FLASK FORMS ON APP.PY

Description: The forms handle user input for login and registration. They ensure that data entered by users meets certain criteria before processing it.



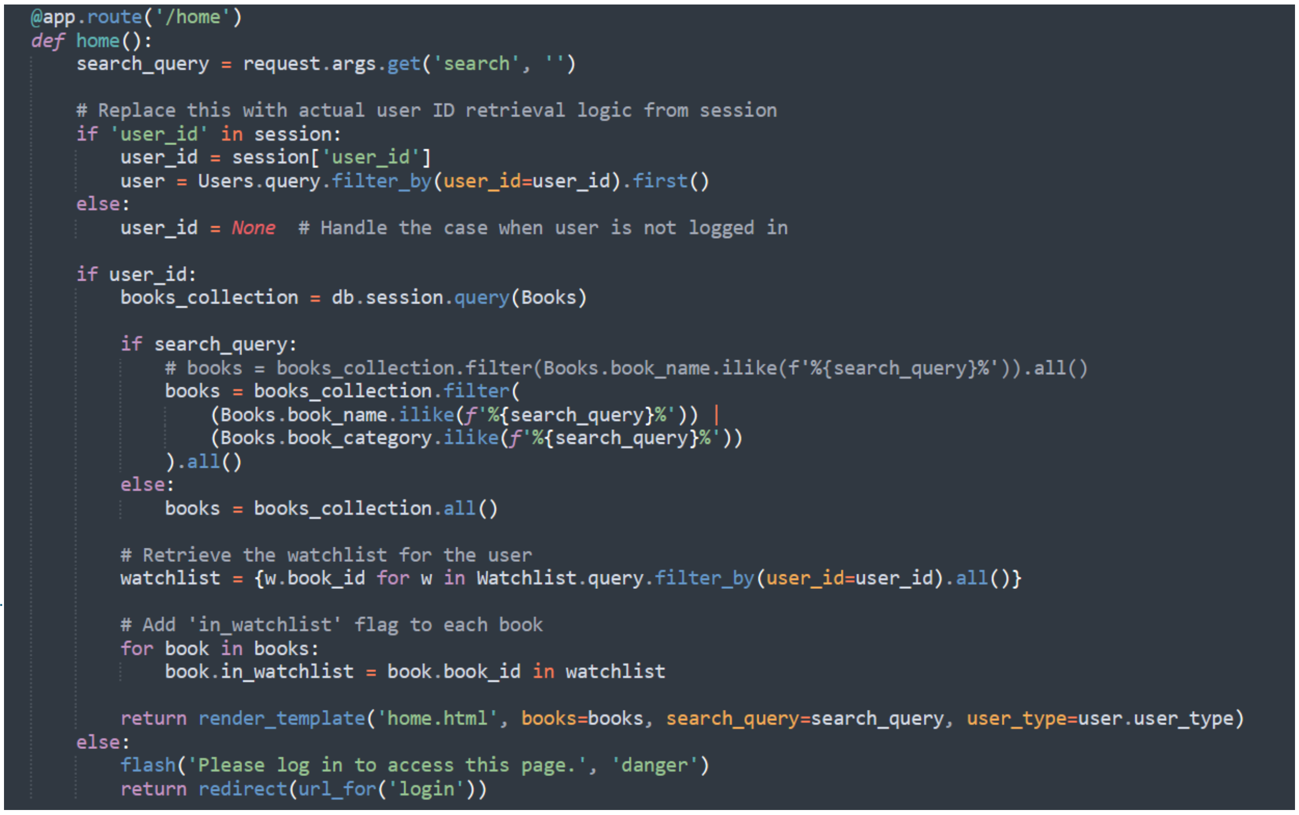
```python
# Create Forms
class LoginForm(FlaskForm):
    user_name = StringField('UserName: ', validators=[DataRequired()])
    user_password = PasswordField('Password: ', validators=[DataRequired()])
    submit = SubmitField('Submit')

class RegistrationForm(FlaskForm):
    user_name = StringField('UserName: ', validators=[DataRequired()])
    user_email = StringField('Email: ', validators=[DataRequired()])
    user_password = PasswordField('Password: ', validators=[DataRequired()])
    confirm_password = PasswordField('Confirm Password: ', validators=[DataRequired(), EqualTo('user_password')])
    submit = SubmitField('Sign Up')
```

ROUTES

HOMEPAGE:

HOME / SEARCH BOOKS

Description: This route renders the home page, allowing users to search for books. It displays books based on the search query and indicates whether they are in the user's watchlist.

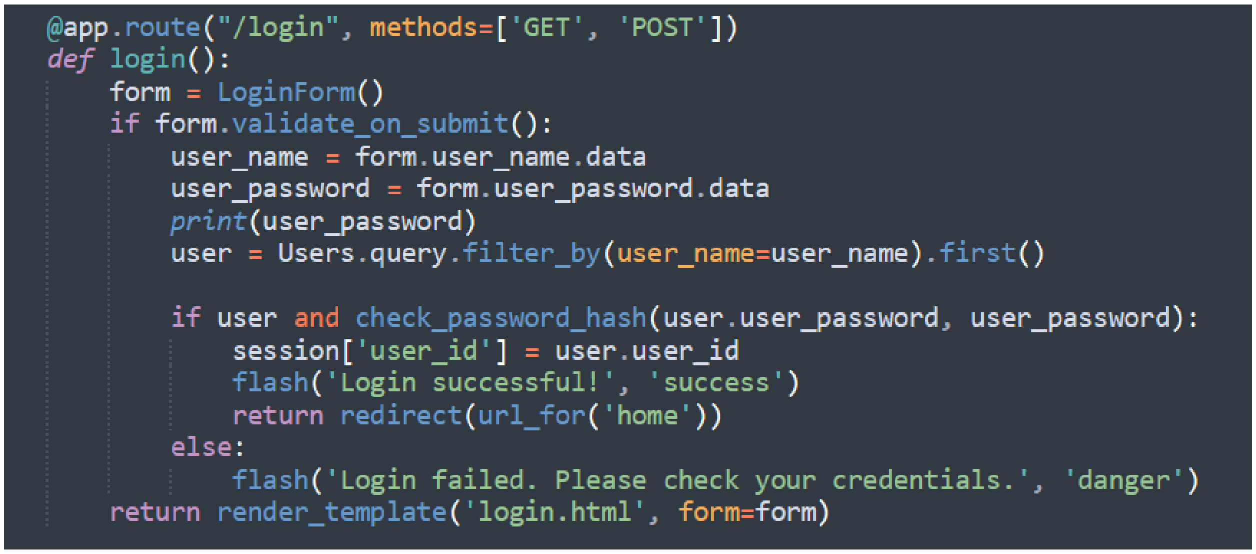


```python
@app.route('/home')
def home():
    search_query = request.args.get('search', '')

    # Replace this with actual user ID retrieval logic from session
    if 'user_id' in session:
        user_id = session['user_id']
        user = Users.query.filter_by(user_id=user_id).first()
    else:
        user_id = None  # Handle the case when user is not logged in

    if user_id:
        books_collection = db.session.query(Books)

        if search_query:
            # books = books_collection.filter(Books.book_name.ilike(f'%{search_query}%')).all()
            books = books_collection.filter(
                (Books.book_name.ilike(f'%{search_query}%')) |
                (Books.book_category.ilike(f'%{search_query}%'))
            ).all()
        else:
            books = books_collection.all()

        # Retrieve the watchlist for the user
        watchlist = {w.book_id for w in Watchlist.query.filter_by(user_id=user_id).all()}

        # Add 'in_watchlist' flag to each book
        for book in books:
            book.in_watchlist = book.book_id in watchlist

        return render_template('home.html', books=books, search_query=search_query, user_type=user.user_type)
    else:
        flash('Please log in to access this page.', 'danger')
        return redirect(url_for('login'))
```

LOGIN

LOGIN / REGISTER

Description: This route handles user login. It validates user credentials and manages the login session.

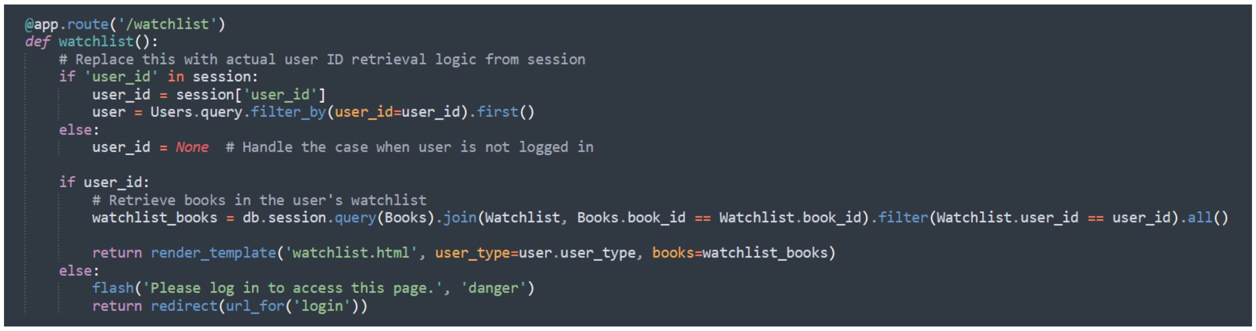


```python
@app.route("/login", methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user_name = form.user_name.data
        user_password = form.user_password.data
        print(user_password)
        user = Users.query.filter_by(user_name=user_name).first()

        if user and check_password_hash(user.user_password, user_password):
            session['user_id'] = user.user_id
            flash('Login successful!', 'success')
            return redirect(url_for('home'))
        else:
            flash('Login failed. Please check your credentials.', 'danger')
    return render_template('login.html', form=form)
```

WATCHLIST:

WATCHLIST PAGE

Description: This route shows the books that are in the user's watchlist.



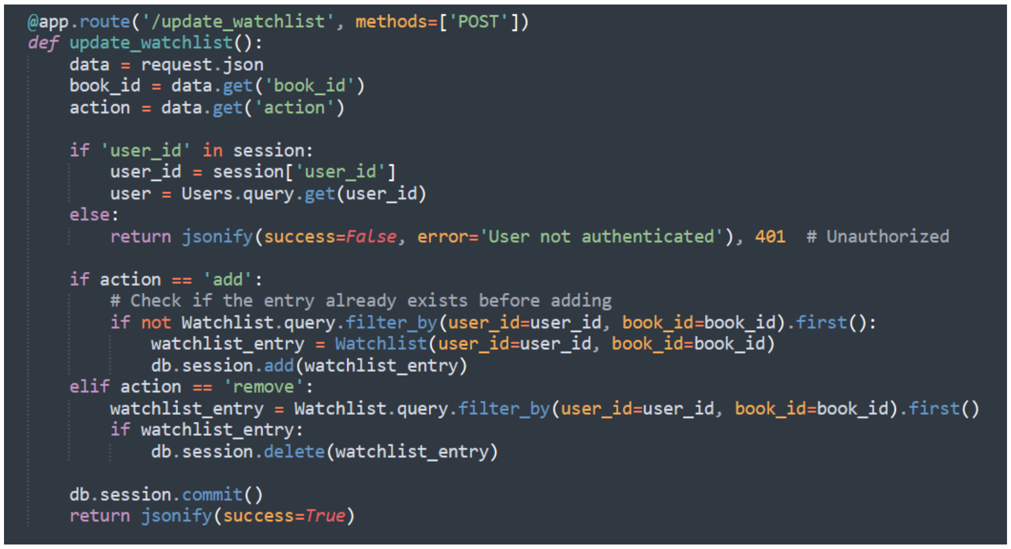
```python
@app.route('/watchlist')
def watchlist():
    # Replace this with actual user ID retrieval logic from session
    if 'user_id' in session:
        user_id = session['user_id']
        user = Users.query.filter_by(user_id=user_id).first()
    else:
        user_id = None  # Handle the case when user is not logged in

    if user_id:
        # Retrieve books in the user's watchlist
        watchlist_books = db.session.query(Books).join(Watchlist, Books.book_id == Watchlist.book_id).filter(Watchlist.user_id == user_id).all()

        return render_template('watchlist.html', user_type=user.user_type, books=watchlist_books)
    else:
        flash('Please log in to access this page.', 'danger')
        return redirect(url_for('login'))
```

WATCHLIST:

API (QUERY) - TO UPDATE WATCHLSIT

UPDATE WATCHLIST:

Description: This route handles adding and removing books from the user's watchlist based on the provided action (add or remove).

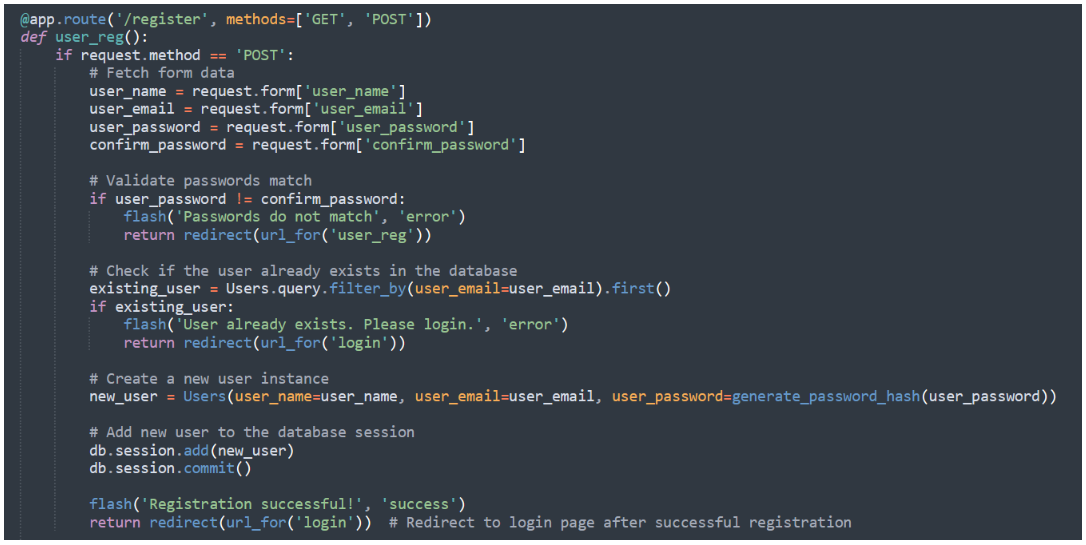


```python
@app.route('/update_watchlist', methods=['POST'])
def update_watchlist():
    data = request.json
    book_id = data.get('book_id')
    action = data.get('action')

    if 'user_id' in session:
        user_id = session['user_id']
        user = Users.query.get(user_id)
    else:
        return jsonify(success=False, error='User not authenticated'), 401  # Unauthorized

    if action == 'add':
        # Check if the entry already exists before adding
        if not Watchlist.query.filter_by(user_id=user_id, book_id=book_id).first():
            watchlist_entry = Watchlist(user_id=user_id, book_id=book_id)
            db.session.add(watchlist_entry)
    elif action == 'remove':
        watchlist_entry = Watchlist.query.filter_by(user_id=user_id, book_id=book_id).first()
        if watchlist_entry:
            db.session.delete(watchlist_entry)

    db.session.commit()
    return jsonify(success=True)
```

REGISTER:

REGISTERING API (SQLALQUEMY)

Description: This route manages user registration, including input validation and adding new users to the database.



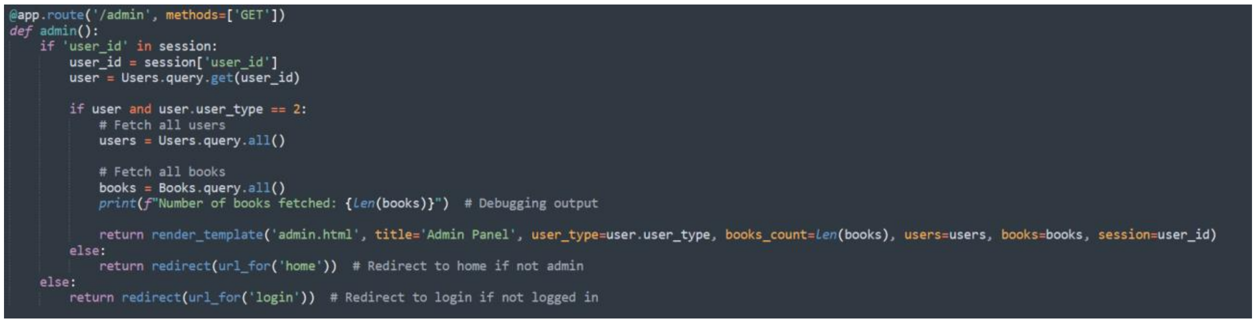
```python
@app.route('/register', methods=['GET', 'POST'])
def user_reg():
    if request.method == 'POST':
        # Fetch form data
        user_name = request.form['user_name']
        user_email = request.form['user_email']
        user_password = request.form['user_password']
        confirm_password = request.form['confirm_password']

        # Validate passwords match
        if user_password != confirm_password:
            flash('Passwords do not match', 'error')
            return redirect(url_for('user_reg'))

        # Check if the user already exists in the database
        existing_user = Users.query.filter_by(user_email=user_email).first()
        if existing_user:
            flash('User already exists. Please login.', 'error')
            return redirect(url_for('login'))

        # Create a new user instance
        new_user = Users(user_name=user_name, user_email=user_email, user_password=generate_password_hash(user_password))

        # Add new user to the database session
        db.session.add(new_user)
        db.session.commit()

        flash('Registration successful!', 'success')
        return redirect(url_for('login'))  # Redirect to login page after successful registration
```

ADMIN ROUTES

ADMIN ROUTE:

ADMIN PANEL PAGE

Description: This route allows administrators to access the admin panel. It checks if the logged-in user has admin privileges. If so, it fetches and displays a list of all users and books. Otherwise, it redirects to the home page or login page based on the user's authentication status.

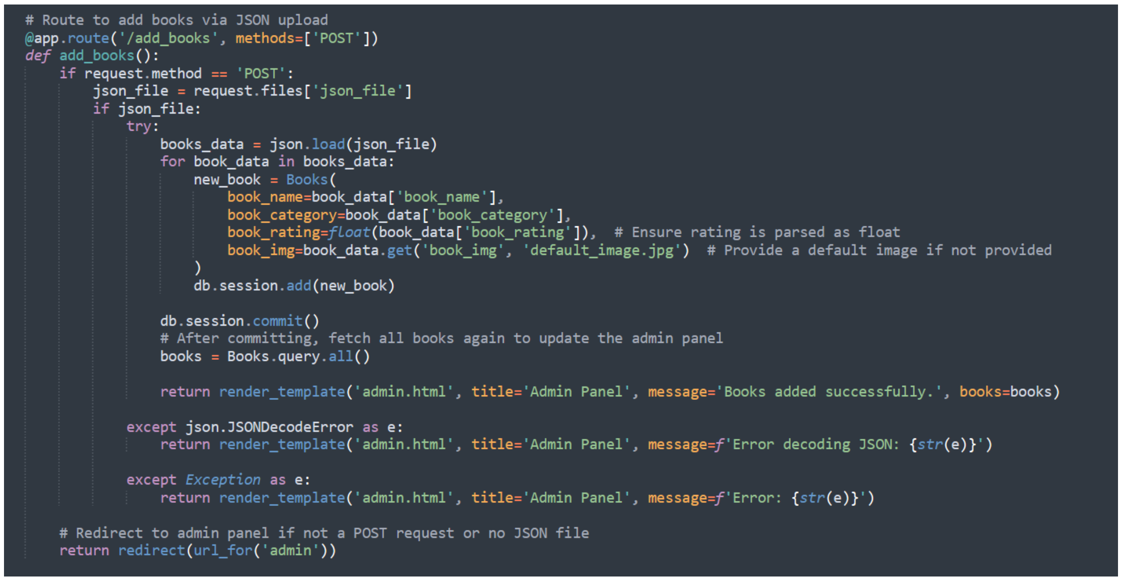


```
@app.route('/admin', methods=['GET'])
def admin():
    if 'user_id' in session:
        user_id = session['user_id']
        user = Users.query.get(user_id)

        if user and user.user_type == 2:
            # Fetch all users
            users = Users.query.all()

            # Fetch all books
            books = Books.query.all()
            print(f"Number of books fetched: {len(books)}")  # Debugging output

            return render_template('admin.html', title='Admin Panel', user_type=user.user_type, books_count=len(books), users=users, books=books, session=user_id)
        else:
            return redirect(url_for('home'))  # Redirect to home if not admin
    else:
        return redirect(url_for('login'))  # Redirect to login if not logged in
```

ADDING BOOKS:

ADDING BOOKS API (SQLALQUEMY)

Description: This route allows the admin to upload a JSON file containing book data. The file is parsed, and each book entry is added to the database. If successful, the admin panel is updated to reflect the new additions. It handles errors related to JSON parsing and other issues.

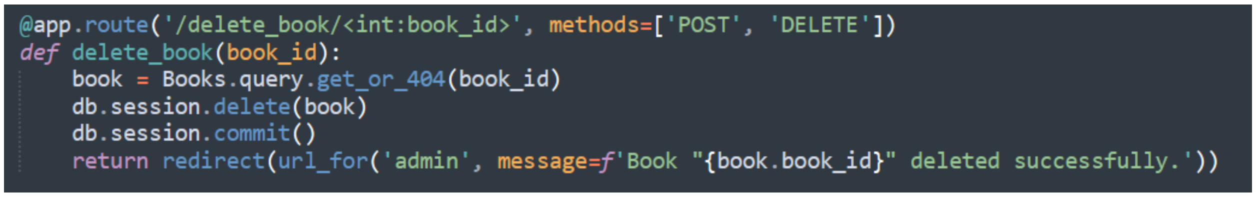


```
# Route to add books via JSON upload
@app.route('/add_books', methods=['POST'])
def add_books():
    if request.method == 'POST':
        json_file = request.files['json_file']
        if json_file:
            try:
                books_data = json.load(json_file)
                for book_data in books_data:
                    new_book = Books(
                        book_name=book_data['book_name'],
                        book_category=book_data['book_category'],
                        book_rating=float(book_data['book_rating']),  # Ensure rating is parsed as float
                        book_img=book_data.get('book_img', 'default_image.jpg')  # Provide a default image if not provided
                    )
                    db.session.add(new_book)

                db.session.commit()
                # After committing, fetch all books again to update the admin panel
                books = Books.query.all()

                return render_template('admin.html', title='Admin Panel', message='Books added successfully.', books=books)

            except json.JSONDecodeError as e:
                return render_template('admin.html', title='Admin Panel', message=f'Error decoding JSON: {str(e)}')

            except Exception as e:
                return render_template('admin.html', title='Admin Panel', message=f'Error: {str(e)}')

    # Redirect to admin panel if not a POST request or no JSON file
    return redirect(url_for('admin'))
```

DELETE BOOKS:

DELETE BOOK ROUTE API (SQLALQUEMY)

Description: This route allows the admin to delete a book from the database based on its ID. After deletion, it redirects back to the admin panel with a success message.

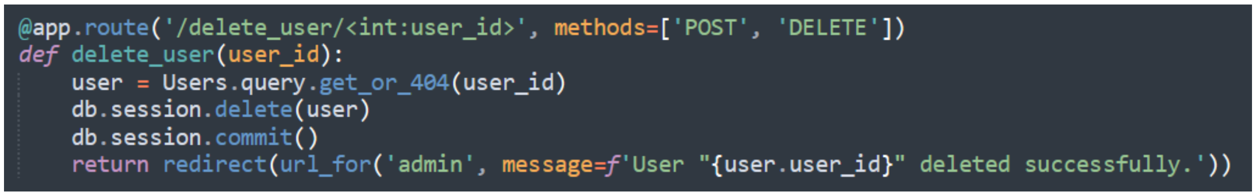


```
@app.route('/delete_book/<int:book_id>', methods=['POST', 'DELETE'])
def delete_book(book_id):
    book = Books.query.get_or_404(book_id)
    db.session.delete(book)
    db.session.commit()
    return redirect(url_for('admin', message=f'Book "{book.book_id}" deleted successfully.'))
```

DELETE USER:

DELETE USER ROUTE API (SQLALQUEMY)

Description: This route allows the admin to delete a user from the database based on their ID. After deletion, it redirects back to the admin panel with a success message



```
@app.route('/delete_user/<int:user_id>', methods=['POST', 'DELETE'])
def delete_user(user_id):
    user = Users.query.get_or_404(user_id)
    db.session.delete(user)
    db.session.commit()
    return redirect(url_for('admin', message=f'User "{user.user_id}" deleted successfully.'))
```

THANKS FOR REVIWEING OUR PROJECT.

Please feel free to reach out and leave your comments.