Bookstore Management System

CIS 552: Database Design (Online – Summer '24)

Project Summary Report

Group 12

(4) Sindhuja Baikadi (02128756)
(49) Vaishnavi Paineni (02125170)

Introduction

The Book Store Management System is a comprehensive solution designed to streamline and automate the operations of an online bookstore. This project leverages MongoDB, a NoSQL database, to efficiently handle the diverse and dynamic nature of book data, including information on authors, genres, and customer reviews. The flexibility of MongoDB's schema allows for seamless management of various data types and structures, ensuring that the database can evolve alongside the expanding inventory of books and user-generated content.

Planned Database-driven Application Requirements/Use-case(s)

The primary purpose of this system is to streamline inventory management, enhance the customer shopping experience, and provide robust administrative tools for managing orders and user accounts. Users can browse and search for books, place orders, and manage their personal profiles within the system. This system provides a search functionality to browse books by genre, authors or publishers. The system also supports user authentication, allowing customers to create accounts, place orders, and write reviews.

Use Cases:

- **Inventory Management:** Administrators can add, and delete book records, ensuring accurate tracking of stock levels and availability.
- **User Management:** The system allows users to create and manage their profiles, including updating personal information and viewing order history.
- **Order Management:** Customers can place orders for books, while administrators can manage orders.

Entities and Attributes:

| Entity	Attributes
Book	BookID, Title, Author, Price, Genre, Published_Year, Publisher
User	UserID, Username, Email, UserType (e.g., Customer, Administrator)
Order	OrderID , UserID, OrderDate, BookID, Price, OrderStatus
Review	ReviewID, BookID, UserID, Rating, Comment, ReviewDate

The **Book** table stores details about each book available in the store.

The **User** table stores information about users, including customers and administrators.

The **Order** table records each order placed by customers.

The **Review** table captures user reviews and ratings for books.

ER Diagram:

User	user_id	username	email	user_type

places   submits

Order	order_id	user_id	order_date	book_id	price	order_status		Review	review_id	book_id	user_id	rating	comment	review_date

contains   linked to

Book	book_id	title	author	price	genre	publisher_year	publisher
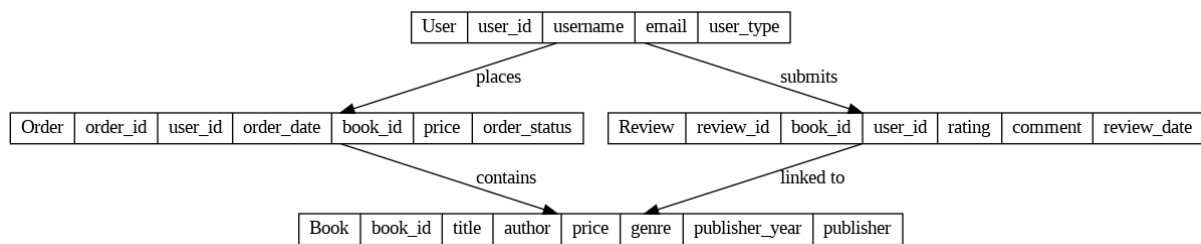
The Entity-Relationship (ER) diagram above illustrates the relationships between key entities in the Bookstore Management System. It includes four primary entities: **User**, **Order**, **Book**, and **Review**.

- **User**: This entity represents the users of the system and includes attributes such as user_id, username, email, and user_type. The user_type attribute helps distinguish between different roles (e.g., customer, admin).

- **Order**: The Order entity captures details related to purchases made by users. It has attributes like order_id, user_id, order_date, book_id, price, and order_status. The user_id establishes a relationship with the User entity, indicating which user placed the order. The book_id links to the Book entity, specifying which book was ordered.

- **Book**: This entity contains information about the books available in the bookstore. Attributes include book_id, title, author, price, genre, publisher_year, and publisher. The book_id serves as a unique identifier for each book.

- **Review**: The Review entity represents user-submitted feedback on books. It contains attributes such as review_id, book_id, user_id, rating, comment, and review_date. The user_id links to the User entity, showing which user submitted the review, while the book_id connects to the Book entity, identifying the book being reviewed.

Relationships:

- **Places**: A User can place multiple Orders, establishing a one-to-many relationship between the User and Order entities.

- **Contains**: An Order contains one or more books, represented by the book_id in the Order entity that connects to the Book entity.

- **Submits**: A User can submit multiple Reviews, creating a one-to-many relationship between User and Review.

- **Linked to**: A Review is linked to a specific Book, indicating that a Book can have multiple reviews associated with it.

This ER diagram effectively models the relationships between the users, books, orders, and reviews within the Bookstore Management System, ensuring a clear understanding of how data flows and interacts within the system.

Technologies Used:

The Bookstore Management System was developed using a combination of modern technologies, ensuring an efficient, scalable, and user-friendly application. The following technologies were utilized:

1. **Streamlit**:
   Streamlit served as the primary framework for the web-based user interface. It enabled the development of an interactive, real-time application with minimal code, allowing for seamless integration of various functionalities. Streamlit's simplicity and flexibility made it an ideal choice for building the bookstore's front-end, offering an intuitive experience for users and administrators alike.

2. **Python**:
   Python was the core programming language used for the backend logic. It handled essential operations such as user authentication, account management, book and order management, and the processing of reviews. Python's extensive libraries and readability facilitated rapid development and integration of complex functionalities.

3. **MongoDB**:
   MongoDB, a NoSQL database, was used to store all data related to users, books, orders, and reviews. Its flexible schema design allowed for dynamic data models, making it easier to accommodate the evolving requirements of the bookstore system. MongoDB's document-oriented approach provided efficient data retrieval and management, contributing to the system's overall performance.

4. **Pymongo**:
   Pymongo was the Python library used to connect and interact with MongoDB. It provided the necessary tools for querying, inserting, updating, and deleting data in the MongoDB database. Pymongo's integration with Python enabled seamless communication between the application's backend and the database, ensuring data integrity and efficient data handling.

Additional Tools and Libraries:

- **Streamlit Authentication**: A custom authentication system was built using Python to manage user access, ensuring that only registered users could log in and perform specific actions based on their user type (e.g., customer, admin).

- **HTML/CSS**: While Streamlit handles much of the UI, custom HTML and CSS were used where necessary to style certain elements and enhance the visual appeal of the application.

Overall Architecture:

The combination of these technologies created a robust architecture for the Bookstore Management System. Streamlit's front-end capabilities, coupled with Python's backend logic and MongoDB's flexible data storage, provided a scalable and efficient solution for managing a wide range of bookstore operations, from user management to order processing and review handling.
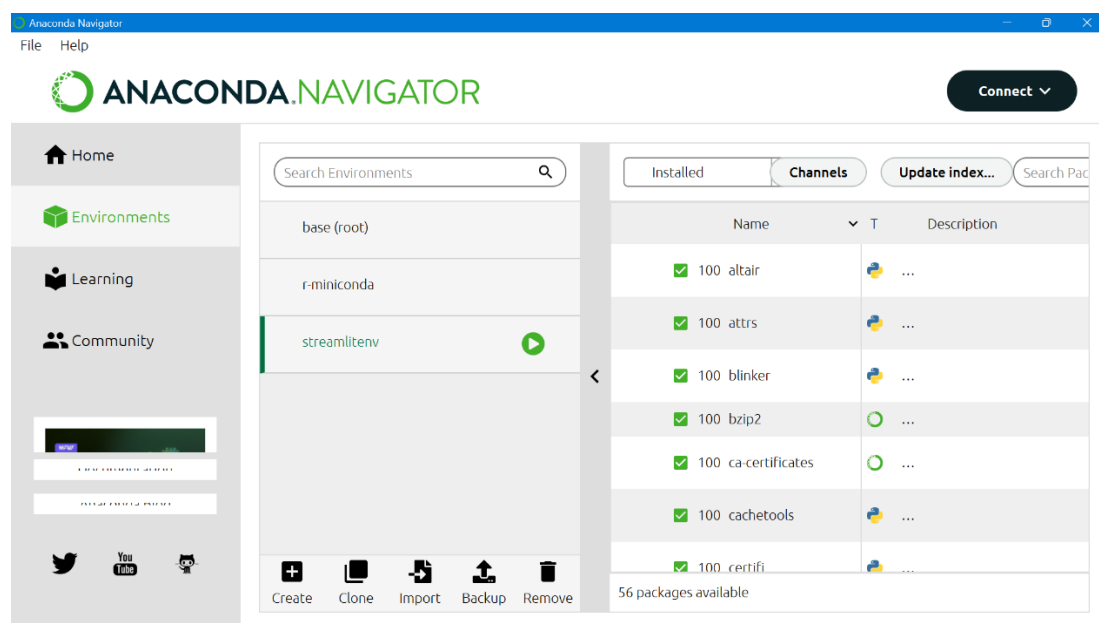
Environment Setup:

To develop and deploy the Bookstore Management System, a series of steps were followed to set up the development environment, install necessary libraries, and configure the tools used for coding and running the application. Below is a detailed description of the process:

1. Creating the Streamlit Environment

The first step in setting up the environment was to create a dedicated environment named streamlitenv using **Anaconda Navigator**. This isolated environment ensured that all necessary packages and dependencies required for the project were managed effectively, without interfering with other projects or system-wide installations.

- **Anaconda Navigator** was chosen for its user-friendly interface and powerful environment management capabilities, which made it easy to create and manage the streamlitenv environment.

- Within Anaconda Navigator, a new environment named streamlitenv was created, with Python as the core interpreter. This provided a clean slate to install the libraries required for the project.

2. Installing Required Libraries

Once the environment was set up, the next step was to install the essential libraries needed to develop the application. This was done using the Command Prompt (CMD):

- **Pymongo**: The pymongo library was installed to facilitate interaction between the Python backend and the MongoDB database. Pymongo provides the necessary tools for connecting to MongoDB, performing CRUD operations, and handling data within the application.
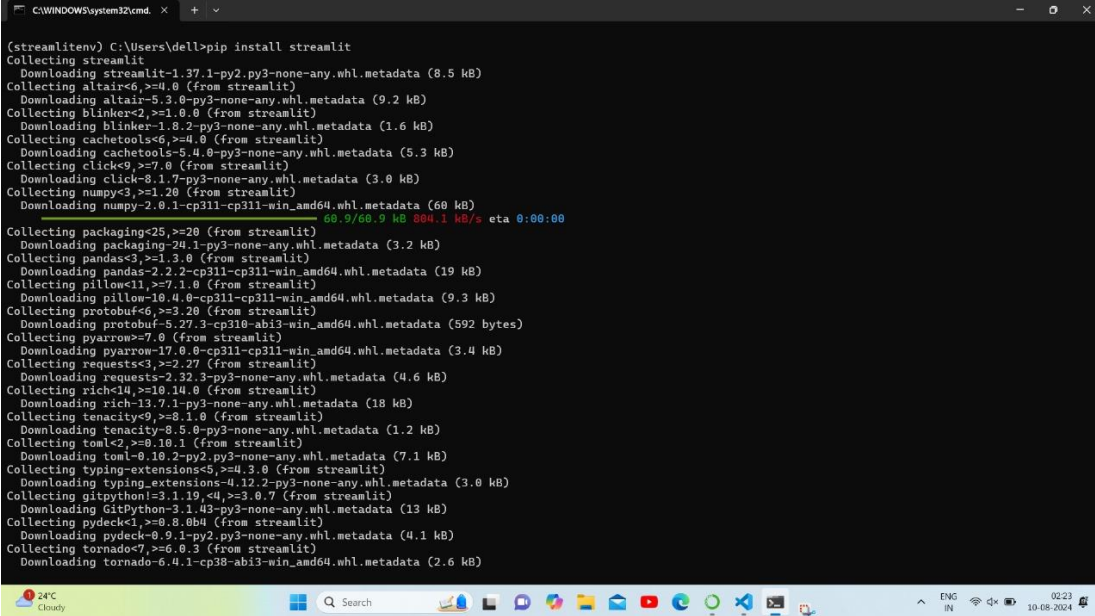
- **Streamlit**: The streamlit library was installed to build the web-based user interface. Streamlit allowed for the rapid development of interactive web applications with minimal coding effort, making it ideal for the project's requirements.

The installations were executed using the following commands in the CMD:

pip install pymongo

pip install streamlit

These commands ensured that the pymongo and streamlit libraries were installed within the streamlitenv environment, making them available for use in the project.

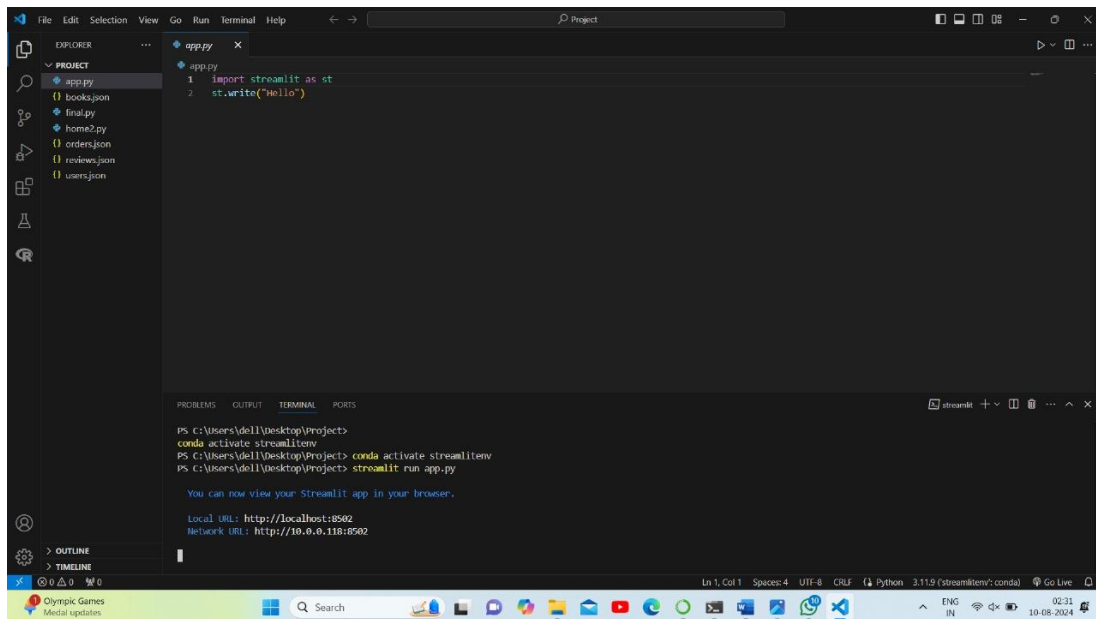## 3. Configuring Visual Studio Code

To streamline the development process, **Visual Studio Code (VS Code)** was configured as the primary Integrated Development Environment (IDE). Setting up VS Code involved the following steps:

- **Interpreter Configuration**: The interpreter for VS Code was set to the streamlitenv environment. This ensured that any code executed within the IDE used the Python interpreter and libraries installed in the streamlitenv environment, preventing any conflicts with other environments or global installations.

To set the interpreter in VS Code:

1. Open the command palette (Ctrl+Shift+P).

2. Type "Python: Select Interpreter" and choose the streamlitenv environment from the list.

This configuration allowed seamless development, debugging, and execution of the application within the correct environment.
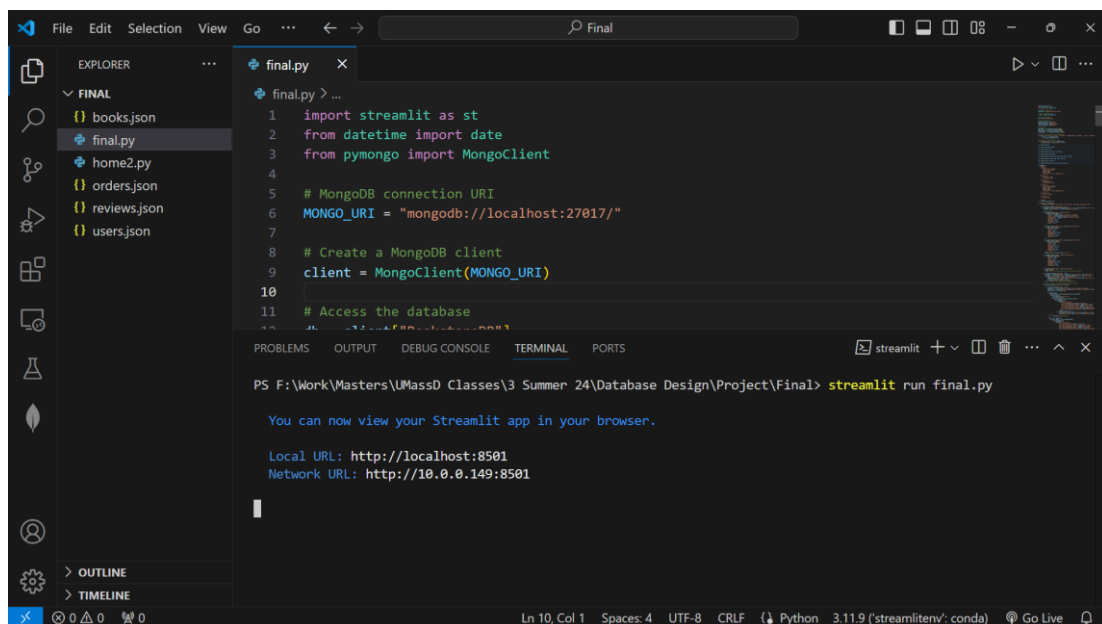
4. Running the Application

With the environment configured and the necessary libraries installed, the next step was to run the application. This was accomplished by using the integrated terminal in Visual Studio Code:

- The following command was used to launch the application:

streamlit run final.py

- This command initiated the Streamlit server, which processed the Python script (final.py) and rendered the web application.

5. Frontend Launch

Upon running the application, Streamlit automatically opened the default web browser, displaying the Bookstore Management System's user interface. The application provided an interactive and visually appealing interface for both customers and administrators, with features such as user management, book management, order processing, and review handling.

- **Customer Interface**: Customers could browse books, place orders, and submit reviews through the user-friendly interface.

- **Admin Interface**: Administrators had access to additional functionalities, including managing books and orders.

This setup process ensured that the development environment was well-organized and that the application could be easily tested and deployed. The use of Anaconda for environment management, along with VS Code for development and Streamlit for deployment, provided a robust and efficient workflow for building and running the Bookstore Management System.
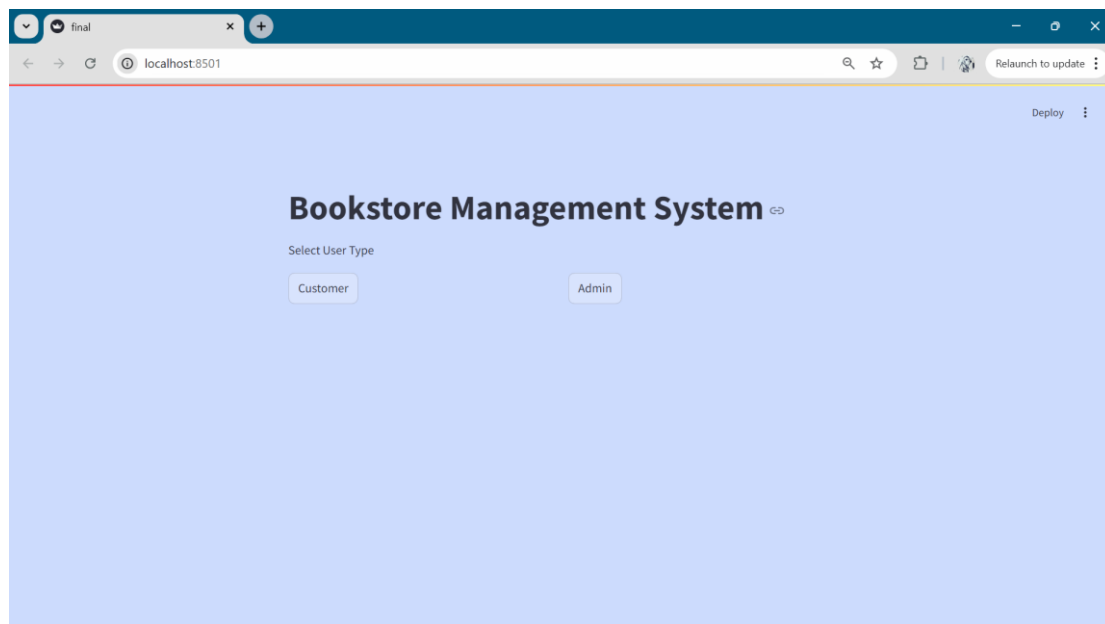

Database Setup and Configuration:

The Bookstore Management System relies on a robust backend database to store and manage data related to users, books, orders, and reviews. The following steps were undertaken to set up and configure the MongoDB database:

1. Connecting to MongoDB:

The first step in setting up the database was to establish a connection with MongoDB. MongoDB was chosen for its flexibility, scalability, and document-oriented data model, which aligned well with the project's requirements. The connection was established by connecting to the localhost and then using the pymongo library in Python, allowing seamless interaction between the application and the database.
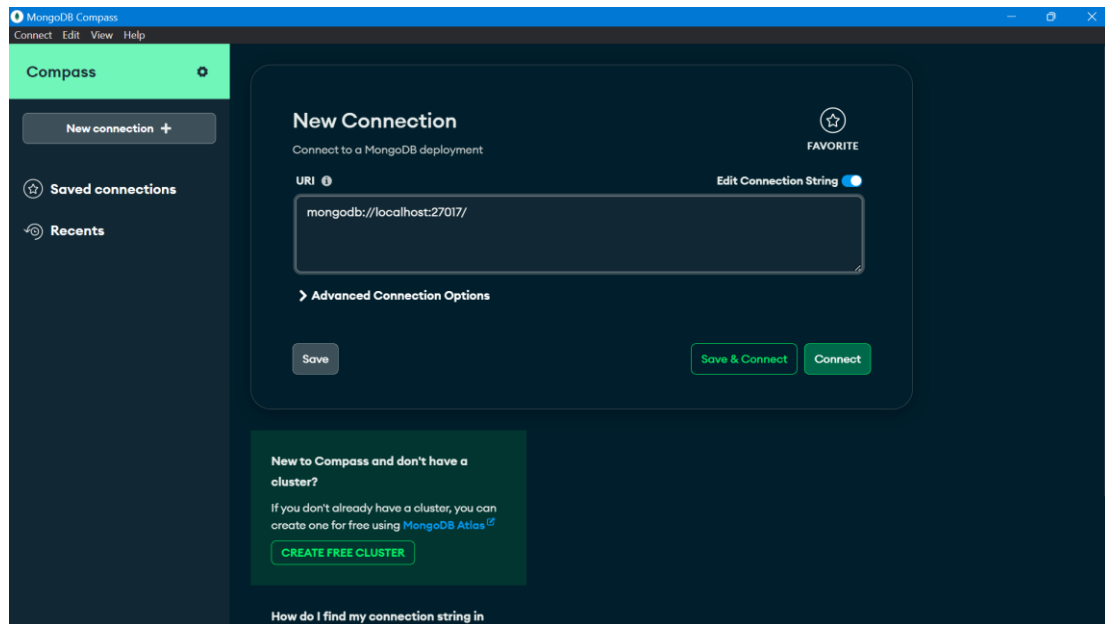
2. Creating the Database:

Once connected to MongoDB, a new database named BookstoreDB was created. This database serves as the central repository for all data managed by the Bookstore Management System.

- The creation of the database was straightforward; MongoDB automatically creates the database when data is inserted into it for the first time. This feature allowed for a seamless transition from connection to data management.
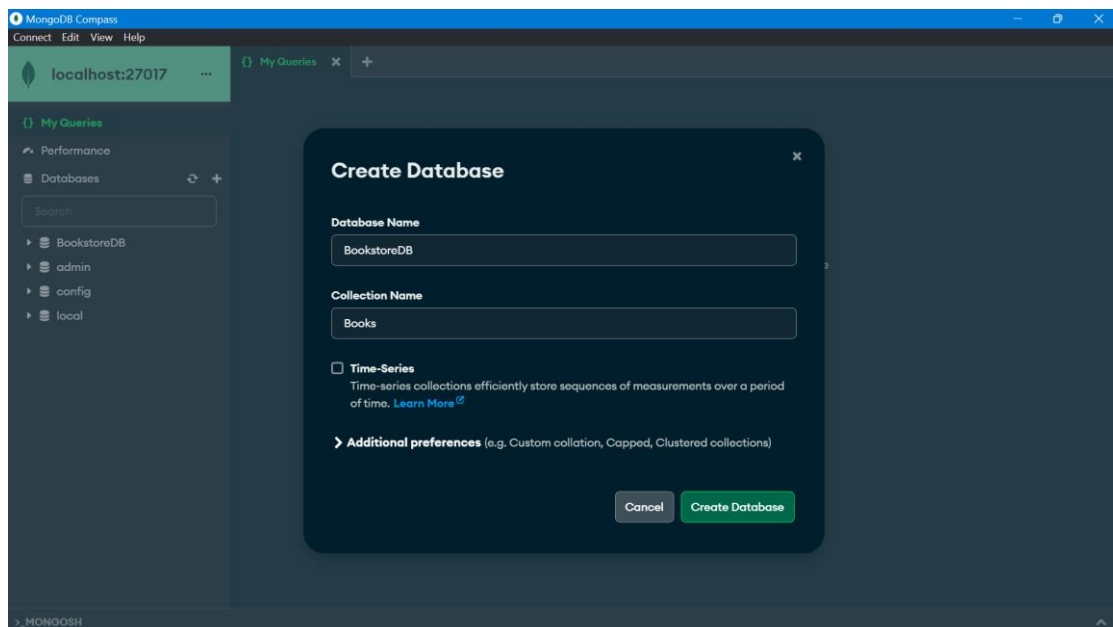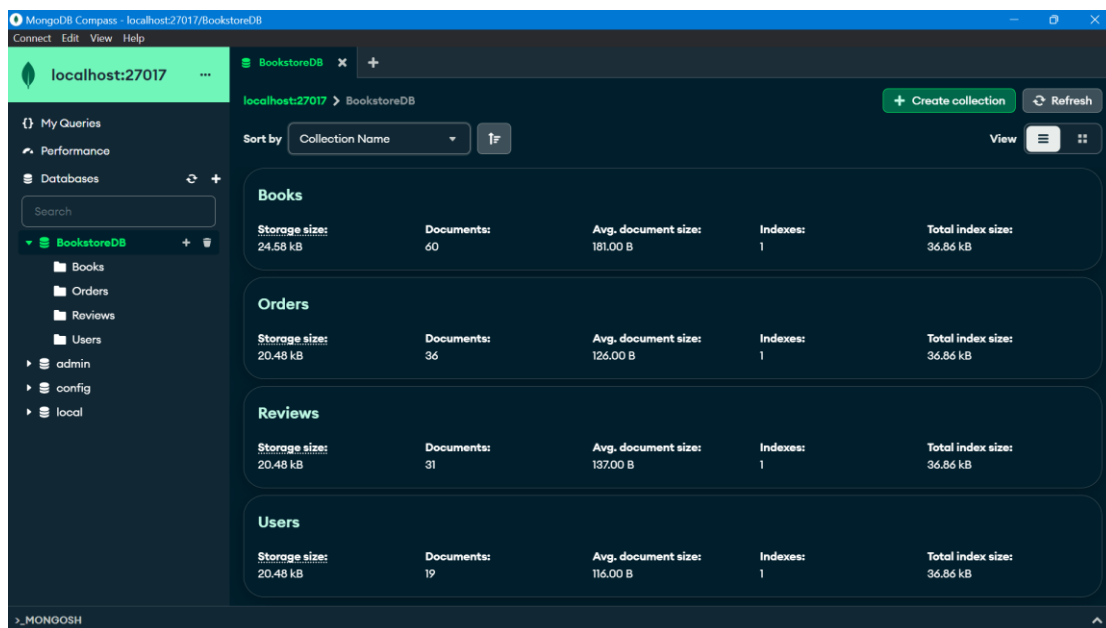
3. Creating Collections:

Within the BookstoreDB database, four distinct collections were created to organize and manage different types of data. Each collection serves a specific purpose and stores related documents:

- **Books Collection:** This collection stores information about the books available in the bookstore. Each document within the books collection includes details such as the title, author, price, genre, publication year, and publisher.

- **Users Collection:** The users collection holds data on the users of the system, including both customers and administrators. Each document in this collection contains user-specific details such as username, email, password, and user type.

- **Orders Collection:** The orders collection captures all the purchase transactions made by users. Each document includes information such as the order date, user ID, book ID, order status, and the price paid.

- **Reviews Collection:** The reviews collection manages user-submitted feedback on books. Each review document contains a review ID, book ID, user ID, rating, comments, and the date the review was submitted.

The creation of these collections provided a well-structured schema for storing and retrieving data, ensuring that the application could efficiently handle operations related to book management, order processing, user authentication, and review submissions.


4. Populating Collections with Data:

After the collections were created, data was added to each collection to populate the database. This involved inserting documents that represented books, users, orders, and reviews.

- **Books Collection:** Initial data entries included a variety of books with relevant metadata, ensuring that the bookstore had a diverse catalog for users to browse and purchase.

- **Users Collection:** User accounts were added, including sample customers and administrator accounts, to facilitate testing and development of user-specific functionalities.

- **Orders Collection:** Sample orders were created to simulate the purchasing process, allowing for thorough testing of order management features.

- **Reviews Collection:** Reviews were added to provide content for the review management system, ensuring that the functionality for submitting, viewing, and managing reviews could be fully tested.

By organizing data into these four collections, the database was structured in a way that supports efficient querying, updating, and management of all the entities involved in the Bookstore Management System. This setup ensured that the application could handle real-world operations smoothly and provided a solid foundation for future enhancements.
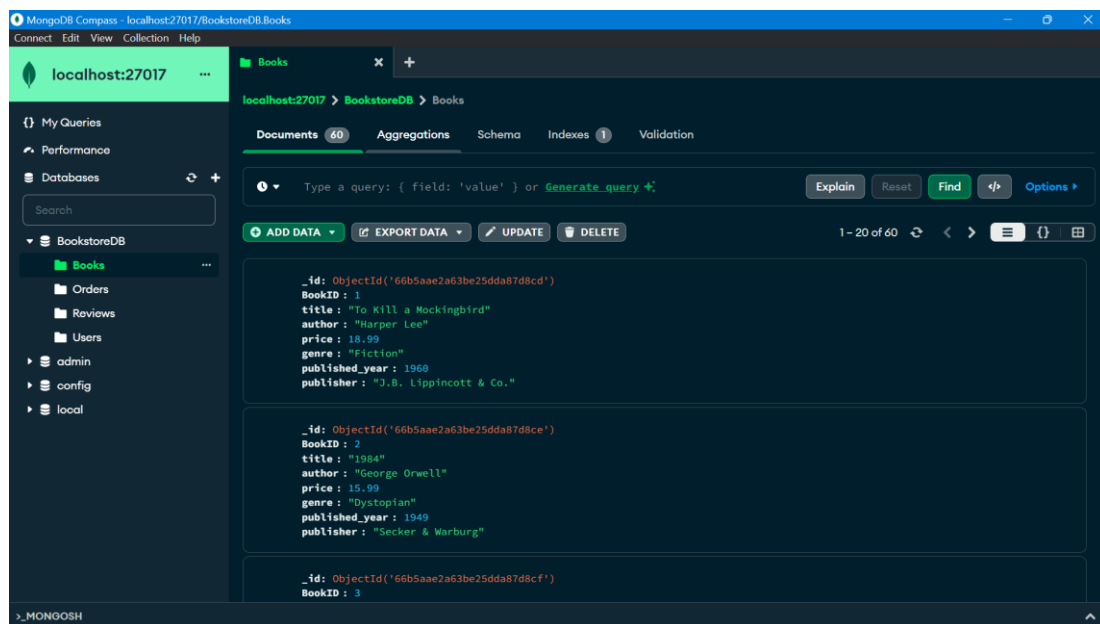
Implementation:

The implementation of the Bookstore Management System involved translating the project's design and requirements into a fully functional application. This section outlines the process of building the system, focusing on the integration of the frontend, backend, and database components. Key functionalities, such as user authentication, book and order management, and review handling, were developed using a combination of Python, Streamlit, and MongoDB. The implementation phase also included rigorous testing to ensure that the system operates smoothly, meets user expectations, and provides a seamless experience for both customers and administrators. Through careful planning and execution, the project was brought to life, resulting in a comprehensive and user-friendly online bookstore.

Connecting the Database to the Backend:

```python
final.py > ...
1    import streamlit as st
2    from datetime import date
3    from pymongo import MongoClient
4
5    # MongoDB connection URI
6    MONGO_URI = "mongodb://localhost:27017/"
7
8    # Create a MongoDB client
9    client = MongoClient(MONGO_URI)
10
11   # Access the database
12   db = client["BookstoreDB"]
13
14   # Access collections
15   books_collection = db["Books"]
16   orders_collection = db["Orders"]
17   reviews_collection = db["Reviews"]
18   users_collection = db["Users"]
19
20   # Fetch data
21   books_data = list(books_collection.find())
22   users_data = list(users_collection.find())
23   orders_data = list(orders_collection.find())
24   reviews_data = list(reviews_collection.find())
```

In this project, MongoDB is connected to the backend using the pymongo library, which facilitates interaction between the application and the database. The connection is established by first importing the necessary modules (pymongo and MongoClient) and specifying the MongoDB connection URI, which in this case is "mongodb://localhost:27017/". A MongoClient object is then created using this URI, allowing the application to connect to the MongoDB server. The specific database, BookstoreDB, is accessed through the client, and individual collections (Books, Orders, Reviews, and Users) are referenced within the database. The collections are then queried to retrieve and store their data in Python lists (books_data, users_data, orders_data, and reviews_data) for use in the application. This setup ensures seamless communication between the backend and the database, allowing for efficient data retrieval and manipulation.

User Authentication and Login Functionality:

```python
def login_user(user_id):
    """ Simulate a login function with different authentication for admins and custo
    if authenticate_user(user_id, st.session_state['user_type']):
        st.session_state['user_id'] = user_id
        st.session_state['authenticated'] = True
        st.session_state['search'] = st.session_state['user_type'] == "Customer"
        st.session_state['order'] = st.session_state['user_type'] == "Customer"
        st.session_state['manage_books'] = st.session_state['user_type'] == "Admin"
        st.session_state['viewing_orders'] = False
        st.session_state['edit_mode'] = False
        st.session_state['add_mode'] = False
        st.session_state['delete_mode'] = False
    else:
        st.session_state['authenticated'] = False
        st.error("Invalid User ID")
```

The login functionality in the Bookstore Management System is designed to authenticate users based on their role as either an admin or a customer. The login_user function is responsible for handling the login process, where it first calls the authenticate_user function to validate the user's credentials against the Users collection in MongoDB. The authenticate_user function checks if the provided user_id and user_type match an entry in the database. If the user is authenticated, various session states are set according to the user's role, enabling access to specific features like book management for admins or order and search functionalities for customers. If authentication fails, an error message is displayed. This approach ensures that users have role-based access to the system's features, enhancing both security and user experience.

Account Creation Functionality:

```python
def create_account(user_type):
    """ Function to create a new customer or admin account without using a form """
    st.session_state['creating_account'] = True
    # Input fields
    name = st.text_input("Name", key="name_input")
    email = st.text_input("Email", key="email_input")
    new_id = st.text_input("ID", key="id_input")

    # Submit button
    if st.button("Create Account", key="create_account_button"):
        if name and email and new_id:
            handle_account_creation(user_type, name, email, new_id)
        else:
            st.error("All fields are required.")
```

The Bookstore Management System includes a streamlined account creation process that allows users to create either customer or admin accounts directly through the interface. The create_account function prompts users to input their name, email, and a unique ID, ensuring all fields are filled before proceeding. Once the "Create Account" button is clicked, the handle_account_creation function validates the ID to ensure it is numeric and checks for its uniqueness in the Users collection within MongoDB. If the ID is valid and not already in use, the add_account function creates a new user document with the provided details and inserts it into the database. Upon successful account creation, the system automatically logs the user in. This functionality simplifies the user onboarding process while maintaining data integrity by validating inputs and preventing duplicate user IDs.

Profile Update Functionality:

```python
if st.session_state.get('edit_mode', False):
    new_name = st.text_input("Name", value=user_profile['Username'], key="name_input")
    new_email = st.text_input("Email", value=user_profile['Email'], key="email_input")
    if st.button("Save Changes", key="save_changes_button"):
        update_profile(int(st.session_state['user_id']),
                       st.session_state['user_type'], new_name, new_email)
        st.session_state['edit_mode'] = False
        st.success("Profile updated successfully!")
    return  # Stop further rendering after editing profile
```

```python
def update_profile(user_id, user_type, name, email):
    """ Update customer or admin profile information in MongoDB """
    users_collection.update_one(
        {"UserID": user_id, "UserType": user_type},
        {"$set": {"Username": name, "Email": email}}
    )
    st.session_state['user_name'] = name
```

The profile update functionality in the Bookstore Management System allows users to modify their account details, such as their name and email, directly within the application. The update_profile function updates the user's information in the Users collection of MongoDB by finding the relevant user document based on their user_id and user_type, and then applying the new values using the $set operator. The edit_mode session state enables the profile editing interface, pre-populating the input fields with the current user information. When the "Save Changes" button is clicked, the updated values are passed to the update_profile function, which applies the changes to the database and updates the session state accordingly. This feature provides users with a seamless and secure way to keep their account information up to date.

Manage Orders Functionality:

```python
def manage_orders(user_id):
    """Function to manage orders based on user type"""
    st.header("Orders")
    # Create two columns for the grid layout
    cols = st.columns(2)

    if st.session_state['user_type'] == "Admin":
        orders = list(orders_collection.find())

    elif st.session_state['user_type'] == "Customer":
        user_id = int(user_id)
        orders = list(orders_collection.find({"UserID": user_id}))
        if not orders:
            st.write("No orders found.")  # Message if no orders are present
```

The manage_orders function in the application displays order information based on the user's role. Admin users can view all orders in the system, while customers are restricted to viewing only their own orders. The function first checks the user type and fetches the relevant orders from the MongoDB database. It then presents these orders in a grid layout with two columns, using HTML for custom styling. Each order's details, including Order ID, Book ID, Price, Order Date, and Status, are displayed in a styled card format.

Review Management Functionality:

```python
def display_reviews(user_id):
    """ Display reviews for books """
    st.header("Book Reviews")
    user_id = int(user_id)
    book_ids = [book['BookID'] for book in books_data]
    reviews = list(reviews_collection.find({"BookID": {"$in": book_ids}}))

    if not reviews:
        st.write("No reviews available.")

    for review in reviews:
        book = books_collection.find_one({"BookID": review['BookID']})
        st.markdown(f"""
            <div class="order-card">
                <h4>Book: {book['title']}</h4>
                <p><strong>Rating:</strong> {review.get('Rating', 'N/A')}</p>
                <p><strong>Comment:</strong> {review.get('Comment', 'N/A')}</p>
                <p><strong>Date:</strong> {review.get('ReviewDate', 'N/A')}</p>
            </div>
        """, unsafe_allow_html=True)
```

The display_reviews function displays book reviews from the database, showing each review's details, such as rating, comment, and date, along with the book's title. It retrieves reviews for all books and presents them in a styled card format using HTML. If no reviews are available, it displays a relevant message. The add_review function allows customers to submit new reviews by selecting a book, rating it, and providing a comment. Upon submission, it adds the review to the database and confirms successful addition.

Book Ordering Functionality:

```python
def order_book(user_id, book_id, price):
    """ Function to order a book """
    new_order_id = orders_collection.count_documents({}) + 1
    id = int(user_id)
    today = date.today()
    new_order = {
        "OrderID": new_order_id,
        "UserID": id,
        "OrderDate": today.strftime('%Y-%m-%d'),
        "BookID": book_id,
        "Price": price,
        "OrderStatus": 'Processing'
    }
    orders_collection.insert_one(new_order)  # Insert new order into MongoDB
    st.success("Book ordered successfully!")
```

The order_book function enables customers to place orders for books. It generates a unique order ID, captures the current date, and creates a new order entry in the database with details such as the user ID, book ID, price, and an initial status of 'Processing'. After inserting the order into the MongoDB collection, it confirms the successful placement of the order with a

notification. This functionality ensures that customers can seamlessly order books and track their status through the application.

Book Search Functionality:

```python
def search_books(search_term, search_by):
    """ Search for books based on search term and search by criteria """
    query = {search_by: {"$regex": search_term, "$options": "i"}}
    return list(books_collection.find(query))
```

The search_books functionality allows customers to search for books based on specific criteria such as genre, author, or publisher. Users can input a search term and select the desired search category. Matching results are displayed in a three-column layout, showing the book's title, author, price, genre, published year, and publisher. Customers can also directly order a book from the search results by clicking the "Order Book" button, which triggers the order_book function to place the order. This feature enhances the user experience by enabling easy access to specific books and streamlining the ordering process.

Book Addition Functionality:

```python
def add_book():
    """Function to add books to the database"""
    st.subheader("Add Books")
    book_title = st.text_input("Title", key="book_title_input")
    book_author = st.text_input("Author", key="book_author_input")
    book_price = st.number_input("Price", value=0.0, step=0.01, key="book_price_input")
    book_genre = st.text_input("Genre", key="book_genre_input")
    book_year = st.number_input("Published Year", value=1950, key="book_year_input")
    book_publisher = st.text_input("Publisher", key="book_publisher_input")
    if st.button("Add Book", key="add_book_button"):
        # Check for empty fields
        if not all([book_title, book_author, book_price, book_genre,
                    book_year, book_publisher]):
            st.error("All fields must be filled!")
        # Check if the book already exists
        elif books_collection.find_one({"title": book_title}):
            st.error("A book with this title already exists!")
        else:
            new_book_id = books_collection.count_documents({}) + 1
            new_book = {
                "BookID": int(new_book_id),
                "title": book_title,
                "author": book_author,
                "price": book_price,
                "genre": book_genre,
                "published_year": int(book_year),
                "publisher": book_publisher
            }
            books_collection.insert_one(new_book)
            st.success(f"Book '{book_title}' added successfully!")
```

The add_book function allows admins to add new books to the database. It collects details such as the book's title, author, price, genre, published year, and publisher through input fields. Upon clicking the "Add Book" button, the function first checks for any empty fields and ensures that the book title is unique. If all validations pass, it generates a new book ID,

inserts the book details into the MongoDB collection, and provides a success message. This functionality is essential for admins to manage and expand the book inventory effectively.

Book Deletion Functionality:

```python
def delete_books():
    """Function to delete books from the database"""
    for book in books_data:
        cols = st.columns([1, 2, 1])
        with cols[1]:
            st.write(f"Title: {book['title']} | Author: {book['author']}")
        with cols[2]:
            if st.button(f"Delete Book {book['BookID']}", key=str(book['BookID'])):
                # Delete book from MongoDB
                books_collection.delete_one({"BookID": book['BookID']})
                st.success("Book deleted successfully!")
                st.session_state["deleted"] = True
```

The delete_books function enables admins to remove books from the database. It displays a list of books with their titles and authors, and provides a "Delete Book" button next to each entry. When an admin clicks the button, the corresponding book is deleted from the MongoDB collection. A success message confirms the deletion, and a session state flag is set to indicate a successful operation. This functionality is crucial for maintaining and updating the book inventory by removing outdated or incorrect entries.

MongoDB Embedded Queries in Python:

In the Bookstore Management System, MongoDB is used to store and manage data related to books, users, orders, and reviews. The system leverages embedded queries within Python code to interact with MongoDB and perform various operations. These operations include searching for books, adding and deleting records, managing orders, and handling user authentication. The following sections detail some of the key MongoDB queries embedded in the Python code.

1. User Authentication Query:

To authenticate users, the system checks the Users collection in MongoDB to verify that the UserID and UserType match the credentials provided during login. This is done using the find_one method:

Example:

user = users_collection.find_one({"UserID": user_id, "UserType": user_type})

This query retrieves a single document from the Users collection where the UserID and UserType match the provided values. If a matching user is found, authentication is successful.

2. Search Books by Criteria:

Users can search for books based on specific criteria like genre, author, or publisher. The search query is dynamically constructed based on the selected criteria using the find method:

Example:

query = {search_by: {"$regex": search_term, "$options": "i"}}

search_results = list(books_collection.find(query))

Here, the search_by variable (e.g., "genre", "author") determines the field to be searched, and the $regex operator is used to perform a case-insensitive search using the provided search_term.

3. Add New Book to Collection:

Admins can add new books to the Books collection. Before insertion, the code checks if a book with the same title already exists to prevent duplicates:

Example:

if books_collection.find_one({"title": book_title}):

   st.error("A book with this title already exists!")

If no duplicate is found, a new book document is inserted using the insert_one method:

books_collection.insert_one(new_book)

4. Delete a Book:

Admins have the ability to delete books from the Books collection. The code identifies the book to be deleted by its BookID and uses the delete_one method to remove it:

Example:

books_collection.delete_one({"BookID": book['BookID']})

5. Manage Orders:

For managing orders, both admins and customers can view their orders. The system retrieves all orders for admins and only the orders placed by a specific customer for customers:

Example:

orders = list(orders_collection.find({"UserID": user_id}))

Admins do not filter by UserID, so they can see all orders.

6. Adding a Review:

Customers can add reviews for books they have purchased. The review is added to the Reviews collection using the insert_one method:

Example:

reviews_collection.insert_one(new_review)

The new_review dictionary contains fields like ReviewID, BookID, UserID, Rating, Comment, and ReviewDate.

7. Display Reviews:

To display reviews for books, the system retrieves all reviews related to the books a customer has purchased. This is done using the find method with an $in operator to match multiple book IDs:
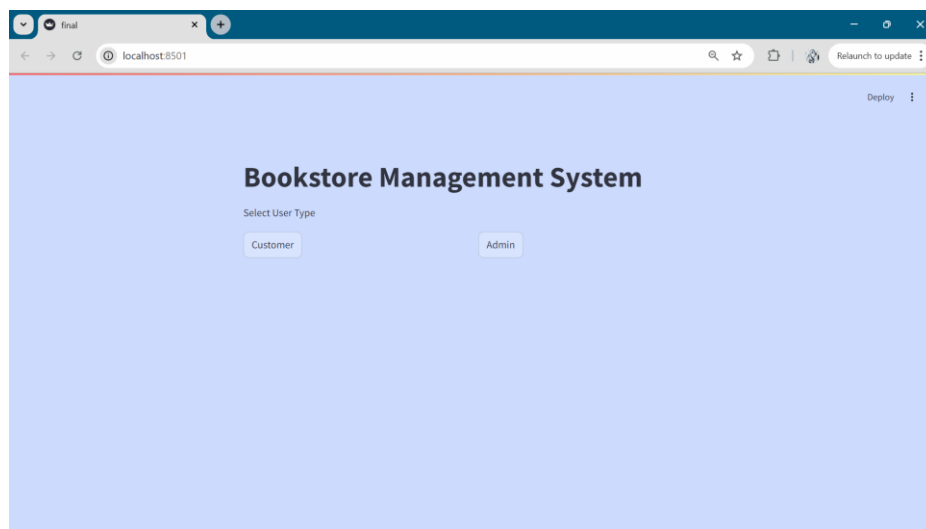
Example:

reviews = list(reviews_collection.find({"BookID": {"$in": book_ids}}))

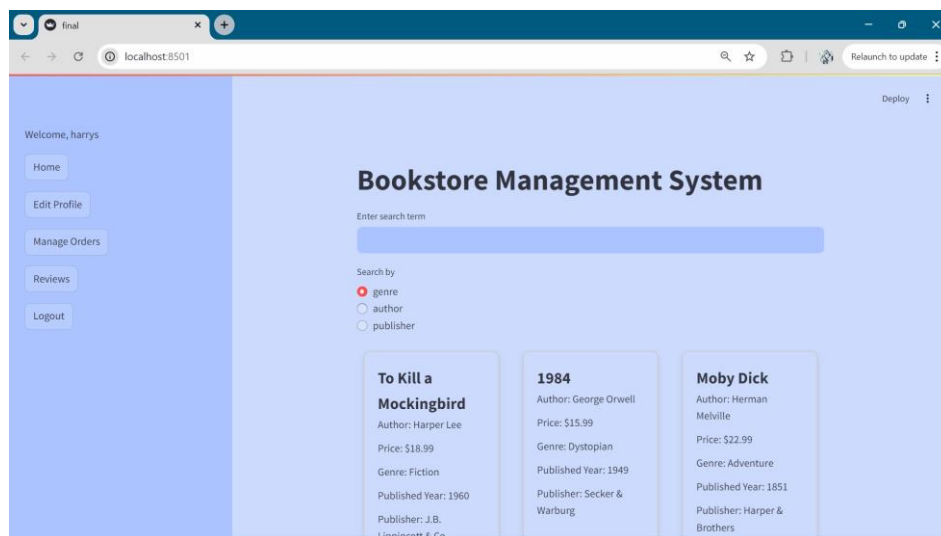This query returns all reviews for books whose IDs are in the book_ids list.
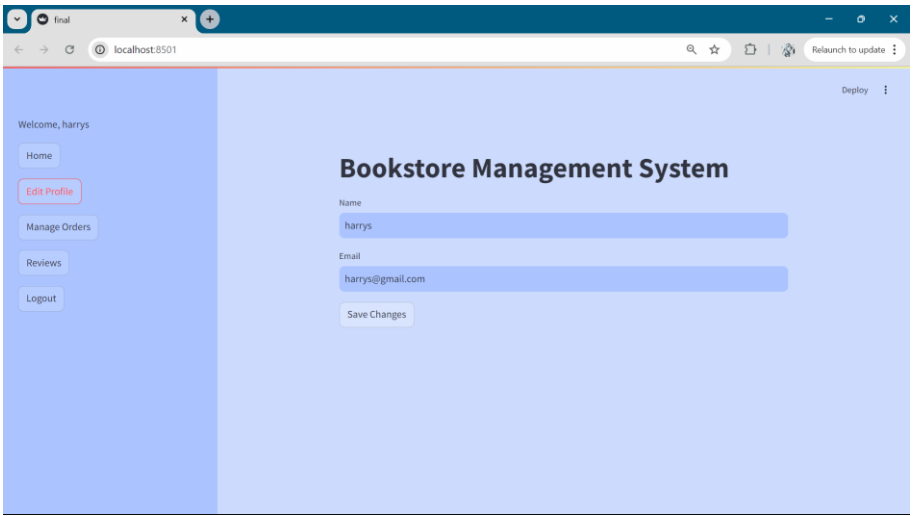
User Interface Overview:
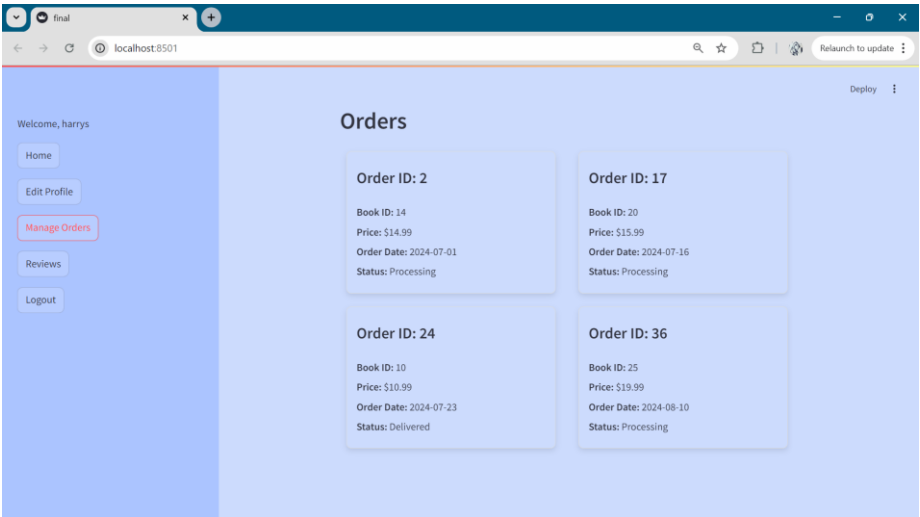
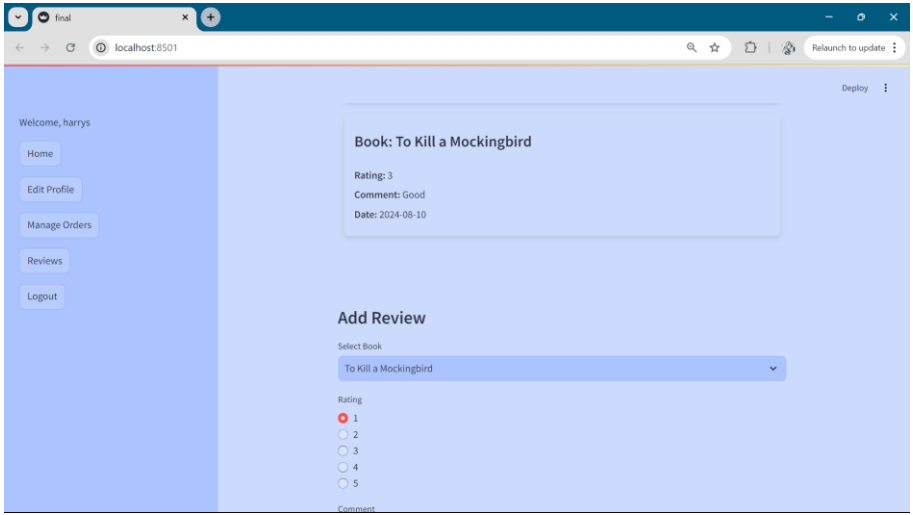User Type Page:

Customer UI Design:
Customer Dashboard:

Update Profile Page:

Manage Orders Page:

Reviews Page:

Administrator UI Design:

Admin Dashboard:
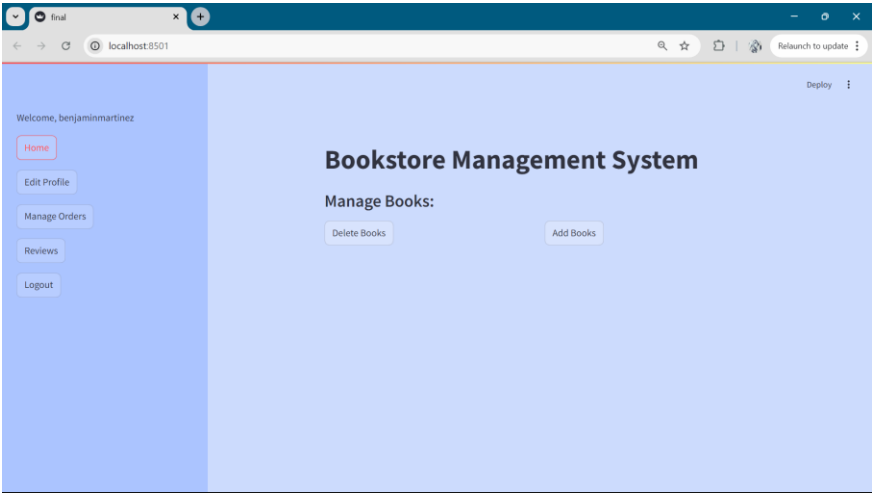
Delete Book Page:
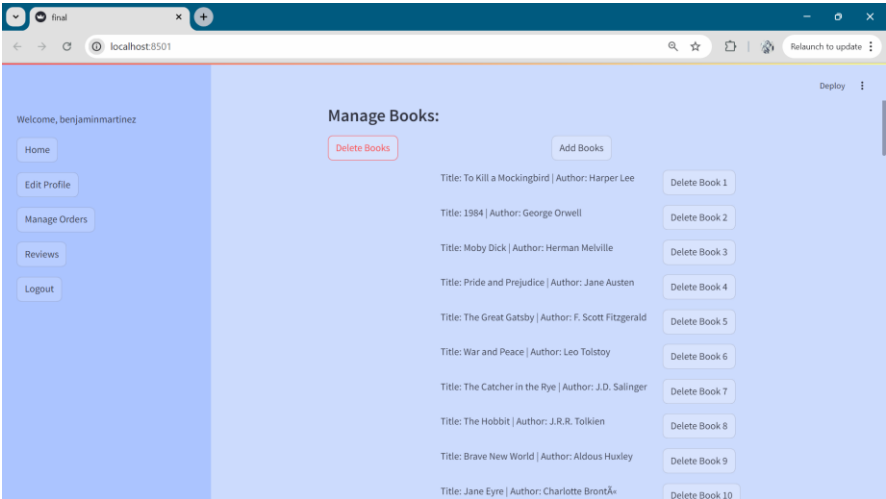
Add Book Page:

Update Profile Page:

Manage Orders Page:

Book Reviews Page:

Future Enhancements:

There are several potential future enhancements that could further improve the functionality and user experience of the Bookstore Management System. One key enhancement could be the implementation of a recommendation system that suggests books to users based on their purchase history and reviews. This would not only personalize the shopping experience but also increase user engagement by promoting relevant content. Additionally, integrating a payment gateway to enable secure online transactions would allow the system to handle the entire purchase process, from browsing to payment, making it more convenient for users.

Another area for enhancement is the expansion of the user role system. Currently, the system differentiates between admins and customers, but adding more granular roles such as content managers, reviewers, or inventory managers could allow for better distribution of responsibilities within larger teams. Also, the introduction of advanced analytics for admin users to track sales trends, customer behavior, and book popularity could provide valuable insights for decision-making and inventory management. These enhancements would significantly increase the system's versatility and scalability, catering to a broader range of user needs.

References:

- https://discuss.streamlit.io/t/how-to-launch-streamlit-app-from-google-colab-notebook/42399/2
- https://code.visualstudio.com/docs/azure/mongodb
- https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label
- https://www.mongodb.com/products/tools/vs-code?ajs_aid=fba1acb3-1fc0-484d-b419-c52048770918&utm_source=vscode&utm_medium=product
- https://docs.streamlit.io/get-started/installation/anaconda-distribution
- https://discuss.streamlit.io/t/how-to-launch-streamlit-app-from-google-colab-notebook/42399/2
- https://jsonlint.com/
- https://github.com/streamlit/streamlit/wiki/Installing-in-a-virtual-environment