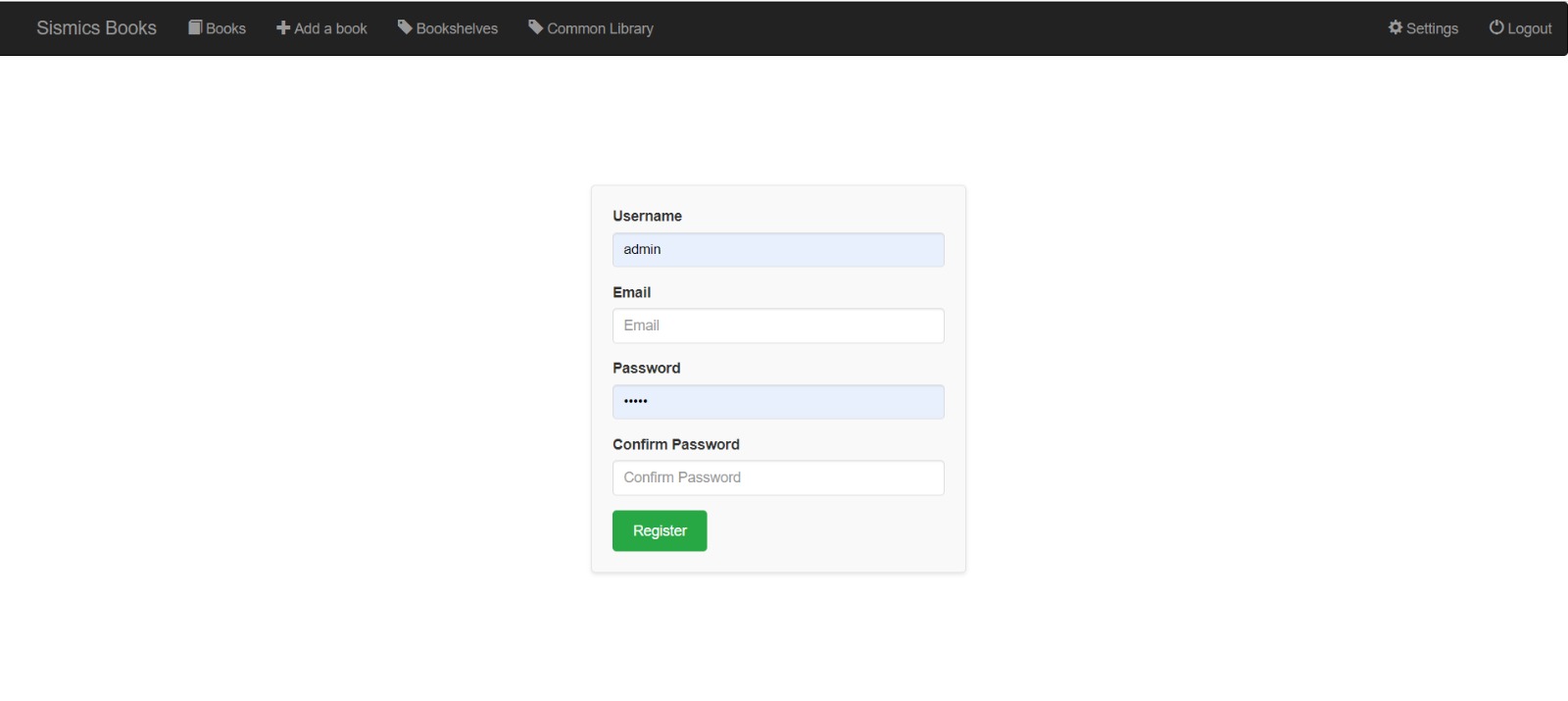
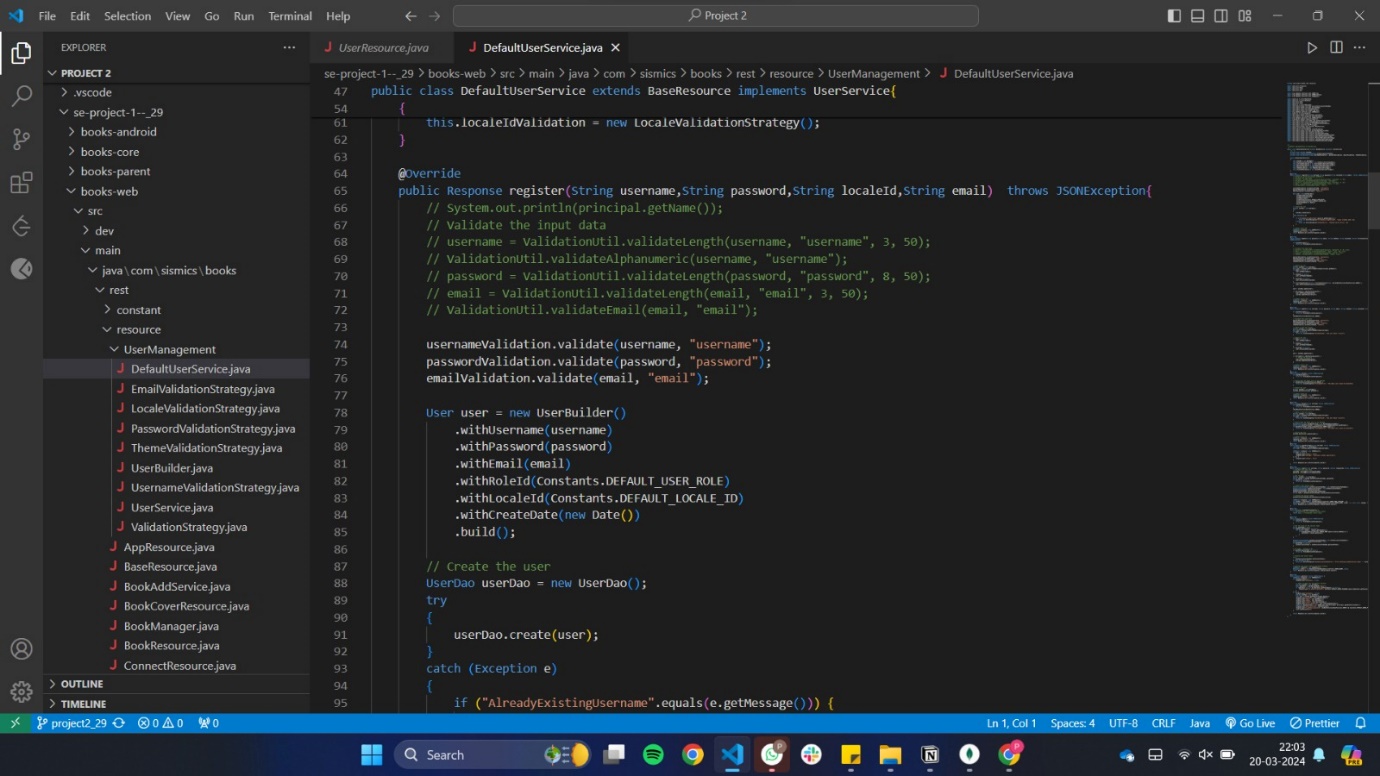
**Software Engineering**

**Project 2 Report**

Feature 1: Better user management

The objective of this feature was to enhance the user experience and convenience by enabling self user registration directly from the login page. We achieved this by constructing a register page for the user where user needs to provide his/her name, email id and password for registration purpose after that checking the constraints for the respective fields that is Username, Email and Password Validation the user is redirected to the login page so that he/she can log in.

**Front-End Implementation:**

****Above is the Screenshot for the implementation of the User Registration Feature in the front-end this page is launched when we hit link [**http://localhost:8080/books-web/#/register**](http://localhost:8080/books-web/#/register).

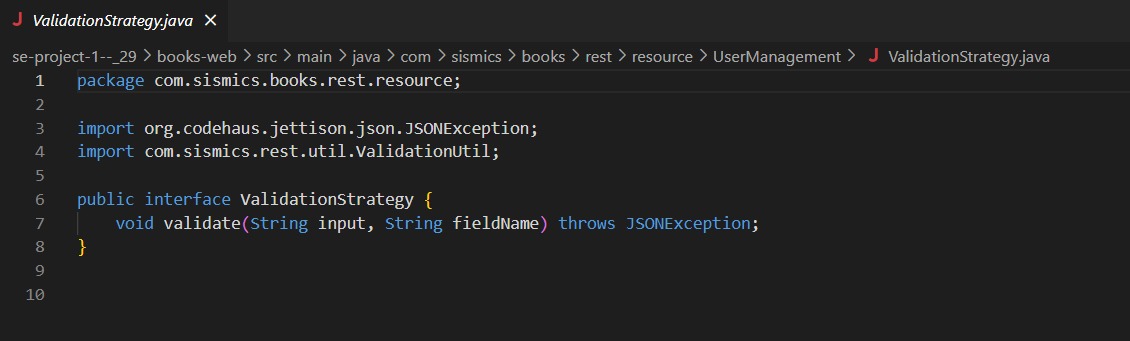
**Back-End Implementation:**

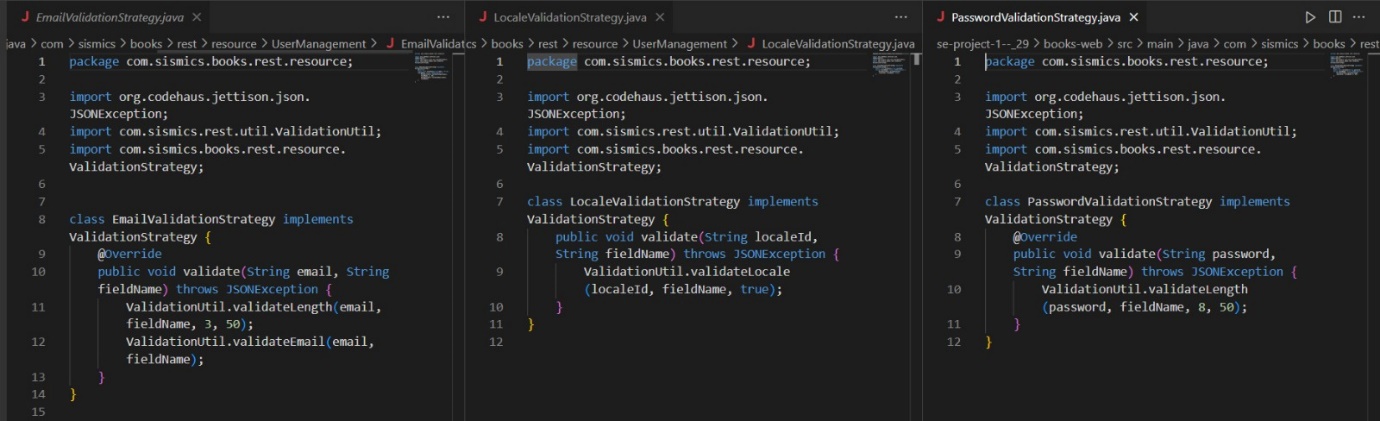
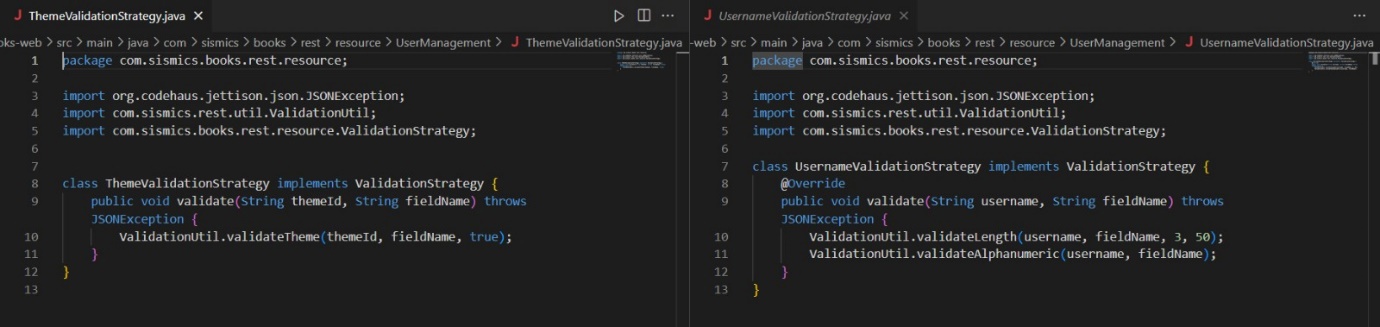
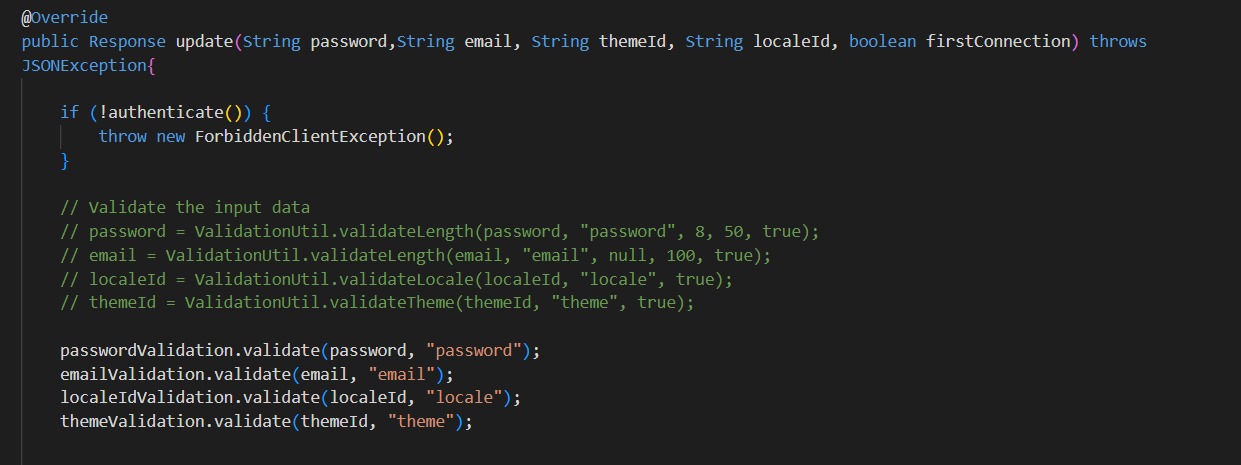
As we can see we have added an HTTP Request call for register in the **UserResource** class that is implemented in **DefaultUserService** class (which extends the **UserService** Interface). In the development of the **DefaultUserService** class for the user management system, we employed the Builder and Strategy design patterns to enhance code quality and maintainability. Here is the description of how we achieved it in this Feature Implementation.

* **Builder Pattern**:
  + We created a **UserBuilder** class that provides a fluent interface for setting various properties of a **User** object and a build() method to return the constructed object.
  + This pattern was utilized in methods like register and update to create **User** objects with different combinations of properties based on the input data.
* **Strategy Pattern**:
  + We defined a **ValidationStrategy** interface and implemented it in classes like **UsernameValidationStrategy**, **PasswordValidationStrategy**, and others, each encapsulating the validation logic for a specific field.
  + In the register and update methods, we used objects of these strategy classes to validate the input data for the corresponding fields.

We identified this Design Patterns and thoroughly analysed the need for implementation of these Design Pattern in the code, here is what we thought before implementing these design patterns:

* Builder Pattern: It was employed to simplify the construction of User objects, making the code more readable and maintainable. It also provided flexibility in object construction, allowing for easy adaptation to different scenarios without the need for multiple constructors or complex parameter handling.
* Strategy Pattern: It was applied to organize the validation logic, making the code more modular and easier to extend. It allowed for the easy addition or modification of validation rules for different fields, promoting the open/closed principle and enhancing the system's adaptability to future requirements.

The implementation for these design pattern is as follows:

Below is the usage of these pattern in the update method of **DeafultUserService** class.

By integrating these design patterns, we achieved a more structured and maintainable codebase, facilitating easier future enhancements and modifications to the books management system.

Feature 2: Common Library

We were asked to implement many different functionalities under this feature here is how we did it:

* **Understanding the Requirements:**

We started by carefully reviewing the requirements provided for the common library subsystem. These requirements outlined the need for functionalities such as viewing all available books, adding new books, rating books, filtering books by genre and author, user registration, and checking user registration status.

* **Designing the REST API:**

With the requirements in mind, we proceeded to design the REST API endpoints that would facilitate the desired functionalities. We created a Java class named CommonLibraryResource.java, which serves as the controller for handling HTTP requests related to the common library operations. Within this class, we implemented methods for each endpoint, ensuring that they adhere to RESTful principles and provide the necessary functionality.

* **Defining Entity Classes with JPA:**

In order to persist data related to books, ratings, and user registrations, we utilized the Java Persistence API (JPA) to define entity classes. These classes represent the core data structures of the application and map directly to database tables. For example, we created entity classes like LibraryBook.java, Rating.java, and Registered.java, each containing attributes that mirror the corresponding database columns.

* **Implementing Data Access Objects (DAOs):**

To encapsulate database interactions and provide a layer of abstraction over the data access logic, we implemented Data Access Objects (DAOs). Following the requirements, we created DAO classes such as LibraryBookDAO.java, RatingDAO.java, and RegisteredDAO.java. These classes contain methods for performing CRUD operations on the corresponding entities, ensuring that the database interactions are handled efficiently and consistently.

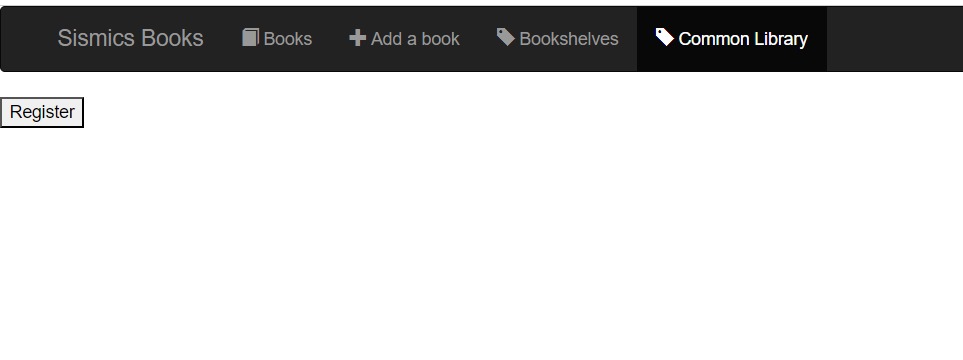
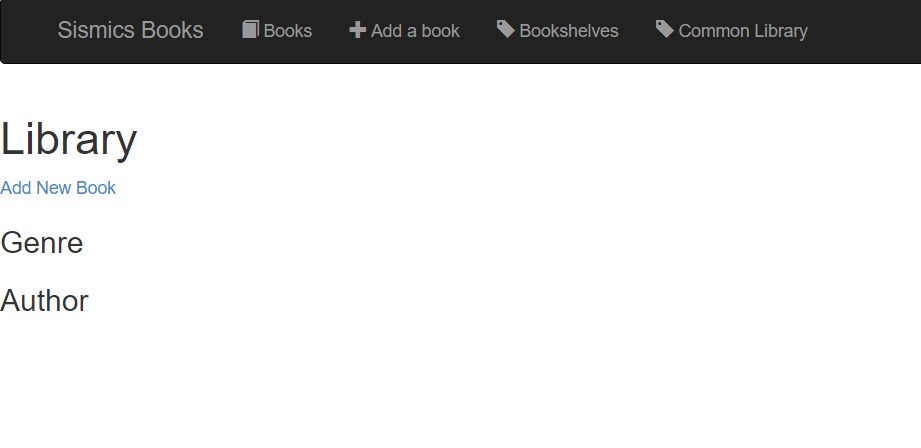
* **Leveraging the Factory Design Pattern:**

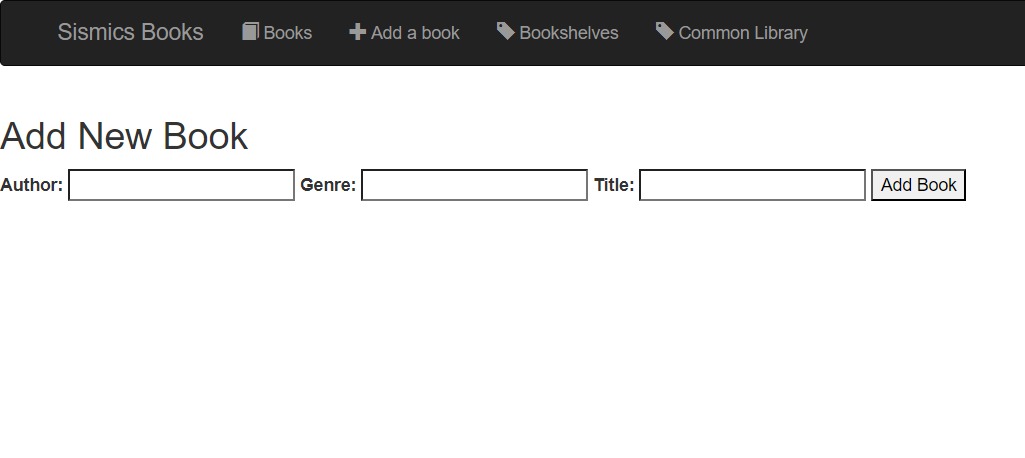
To streamline the creation of DAO instances and promote modularity within the application, we employed the Factory design pattern. We implemented a class named DAOFactory.java, which serves as a central factory for creating DAO objects. This approach allows us to manage DAO instantiation in a flexible and scalable manner, facilitating future changes or additions to the DAO layer.

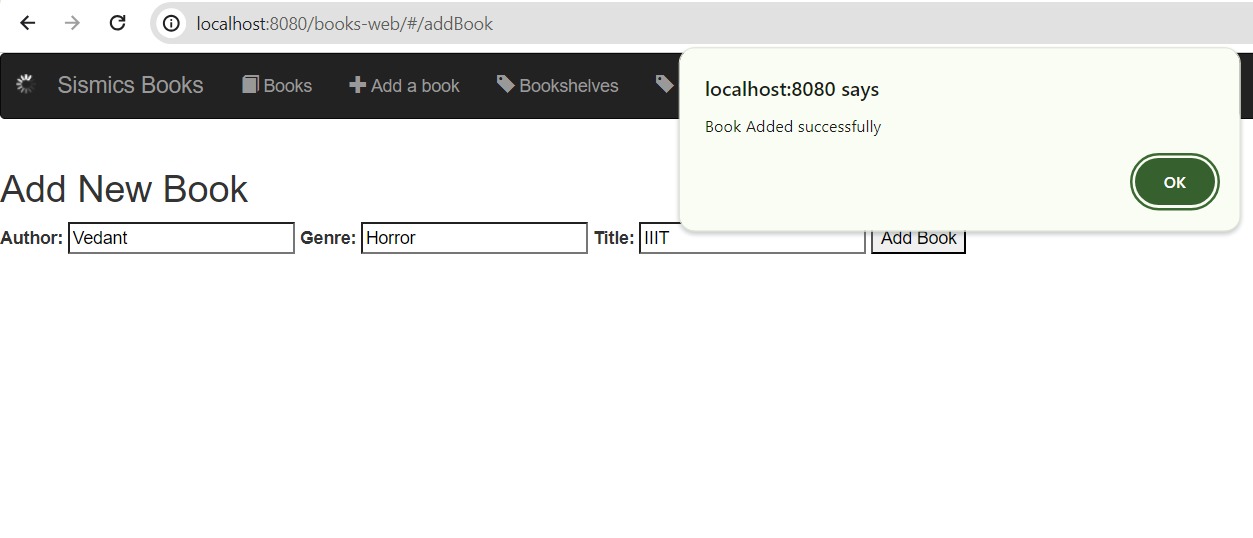
* **Ensuring Modularity and Maintainability:**

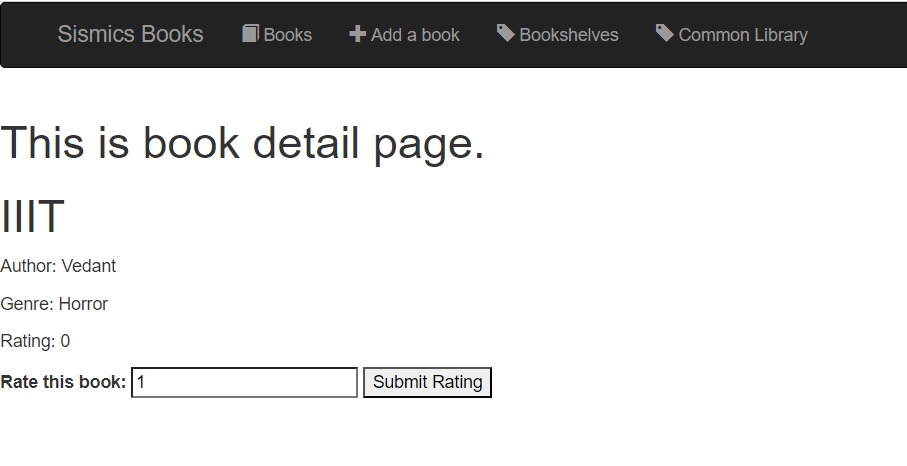
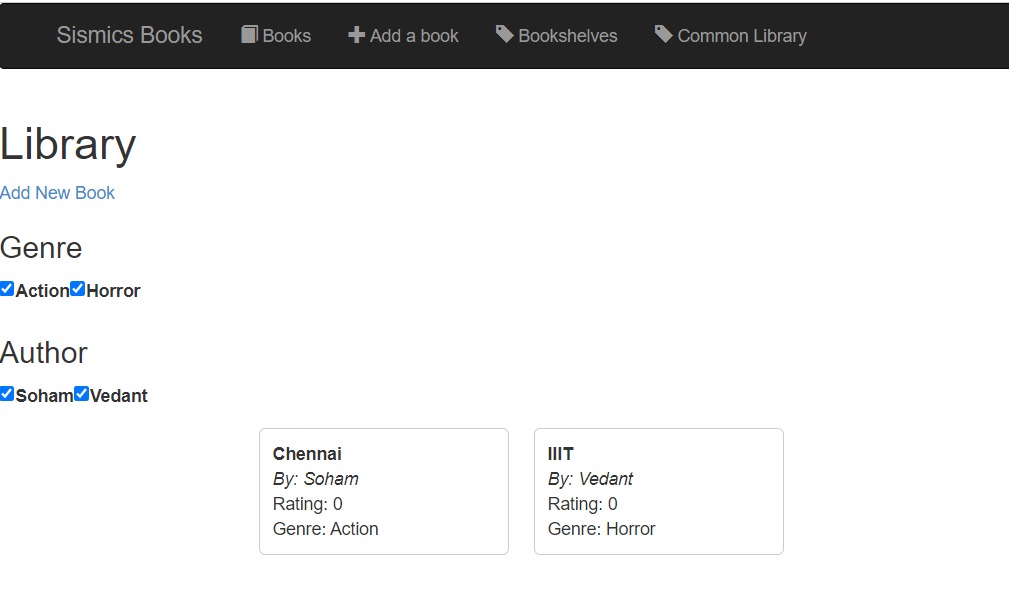
Throughout the implementation process, we focused on ensuring modularity and maintainability to support the long-term evolution of the application. We adhered to best practices in code organization, separation of concerns, and design patterns, making the system easier to understand, extend, and maintain over time. Additionally, we incorporated error handling and logging mechanisms to enhance the reliability and robustness of the application.

**Front-End Implementation:**

The above is the page for the Common-Library Feature page. Before the user is able to access the Common Library for that purpose he/she needs to register for the Common Library.

The Front-End Implementation of Adding a Book in Library after adding the required details is as follows:

The functionality to add give rating to an already added book is as follows:

The functionality to get the Common Library books based on the Genre and Author is as follows (It shows all the available Genres and Authors that are available across all the added books):

**Back-End Implementation:**

Following is the Common Library Book class that we have created to fulfil the requirements for this feature implementation:

The following is the class that allow the implementation of different functionalities for the Common Library class:

Following is the Implementation for the Factory Design Pattern that we have recognized and implemented for the specified feature:

