

**UNIVERSITY OF GHANA**

**SCHOOL OF ENGINEERING SCIENCES**

**COLLEGE OF BASIC AND APPLIED SCIENCES**

**DEPARTMENT OF COMPUTER ENGINEERING**

**FIRST SEMESTER 2024/2025 ACADEMIC YEAR**

**Course Code & Title: CPEN 421 – Web & Mobile Software Architecture**

**Course Instructor: Mrs. Gifty Osei**

**Teaching Assistant: Ms. Akua Serwaa Nkrumah**

**Lab: 3**

**Name: Waqas Haaris Ali**

**Student ID: 10947015**

**LAB 3 REPORT - BUILDING A SCALABLE WEB APPLICATION USING SOLID PRINCIPLES: A CLIENT-SERVER APROACH WITH REACT & NODE JS**

# INTRODUCTION

The objective of this lab was to design and implement a **Book Collection System** that adheres to the principles of the **client-server architecture**. The project demonstrates concepts such as HTTP communication, REST APIs, and modular design.

The system provides the following features:

* Add a new genre/book to the collection.
* View the list of genres/books in the collection.
* View details about a specific genre/book.
* Edit genre/book details.
* Delete a genre/book from the collection.

**Technologies Used**:

* **Frontend**: React, Fetch API, React Router.
* **Backend**: Node.js, Express, Sequelize (ORM for Database)
* **Database**: PostgreSQL.

# DATABASE DESIGN

## Schema Design

The database contains two entities:

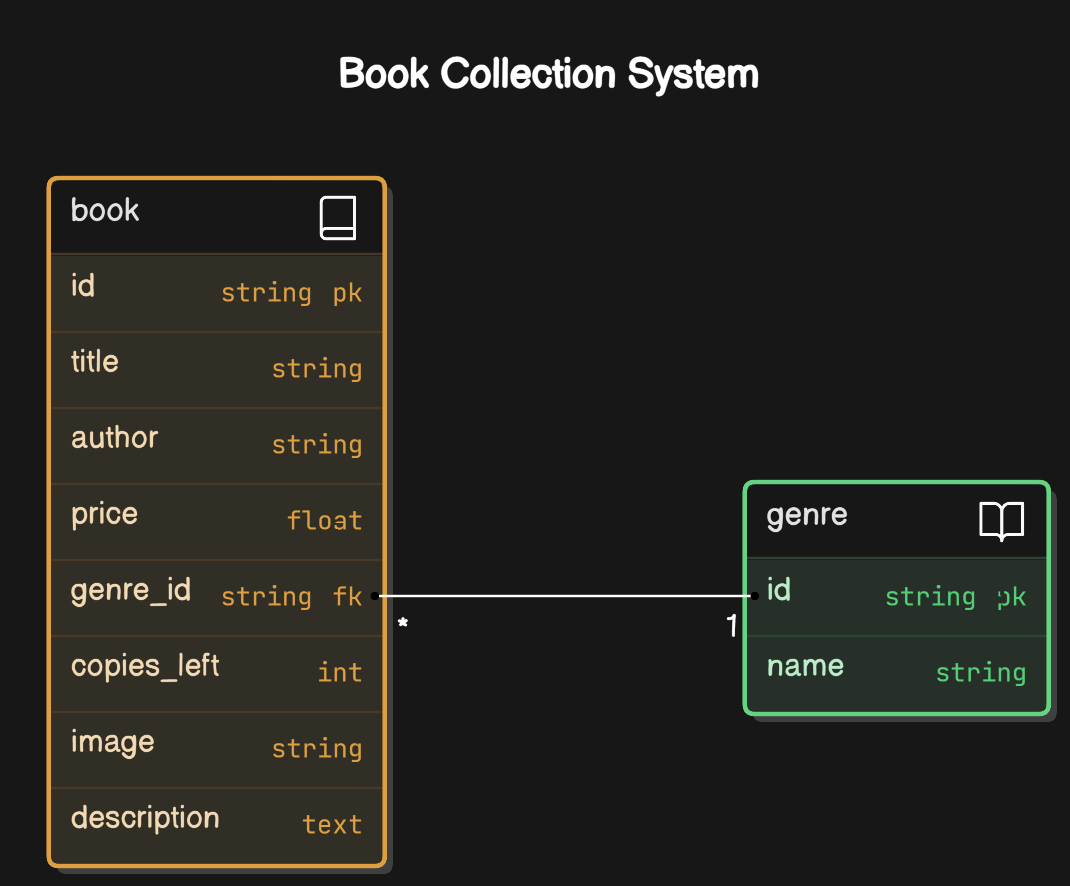
1. **Genre**:

* Attributes: id, name.
* Relationships: Has many Books

1. **Book**:

* Attributes: id, title, author, price, genre\_id, copies\_left, image.
* Relationships: Belongs to Genre.

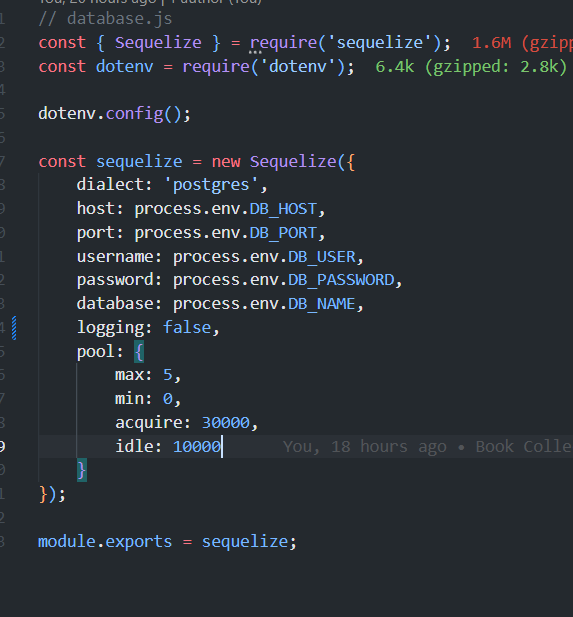
## ER Diagram



## Setup Process

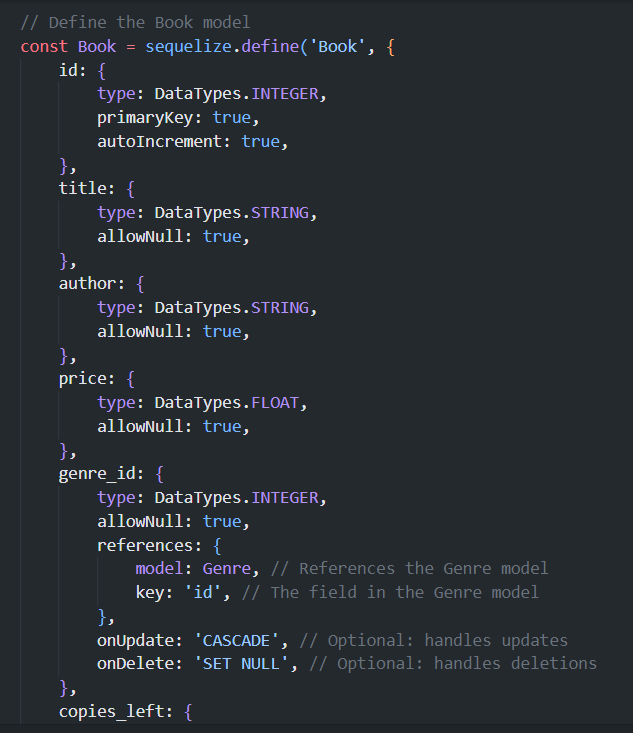
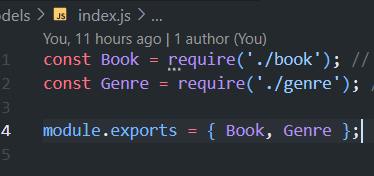
The database ‘books’ was created in PostgreSQL

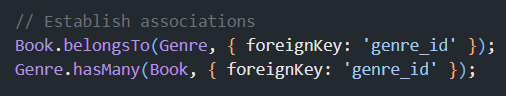
* Defined models using Sequelize ORM:
  + Here is the instance of Sequelize created within a database.js file:



* **Genre model in models/Genre.js.**



* **Book model in models/Book.js.**
* 
* **Exporting the Book and Genre model in index.js:**
* 
* **Established relationships:**

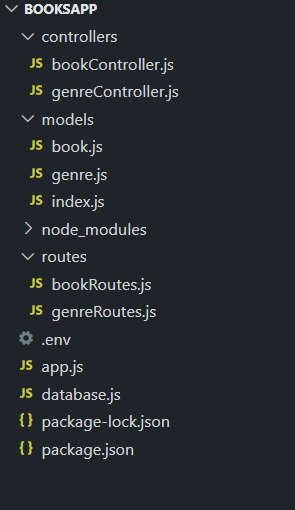


# SERVER IMPLEMENTATION

## Environment Setup

* Initialized a Node.js project.
* Installed required packages: express, sequelize, pg, pg-hstore, cors, and dotenv.

## Server File Structure

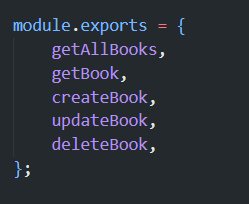


## Server Logic (controllers)

1. **genreController.js (controllers/genreController.js)**



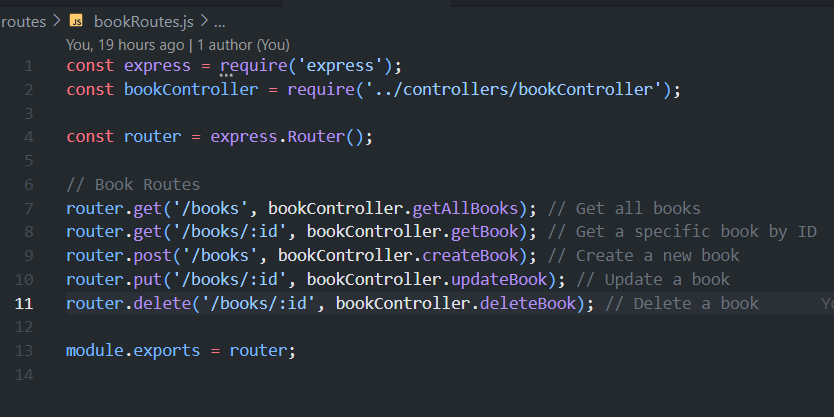
1. **bookController.js (controllers/bookController.js)**



## API Endpoints (routes)

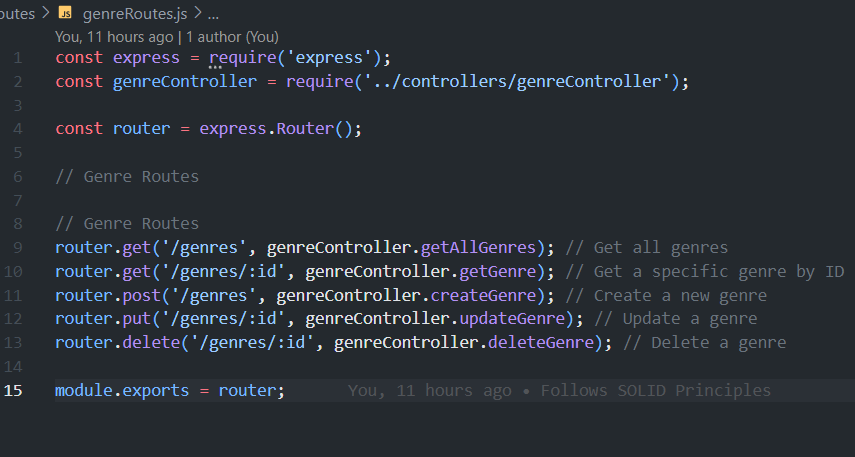
1. Books (defined in routes/bookRoutes.js)

* **GET /books**: Fetch all books.
* **GET /books/:id**: Fetch a specific book by ID.
* **POST /books**: Add a new book.
* **PUT /books/:id**: Update book details.
* **DELETE /books/:id**: Delete a book by ID.



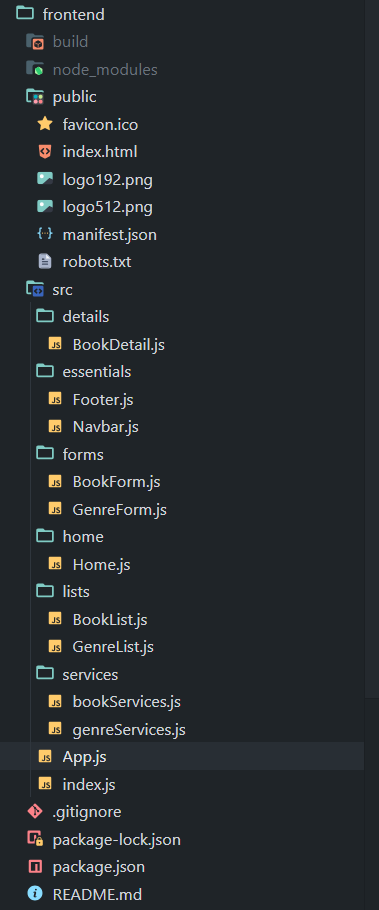
1. Genres (defined in routes/genreRoutes.js)

* **GET /genres**: Fetch all genres.
* **POST /genres**: Add a new genre.
* **PUT /genres/:id**: Update genre details.
* **DELETE /genres/:id**: Delete a genre by ID.



# CLIENT-SIDE (FRONTEND) IMPLEMENTATION

## Client Side Structure



## SERVICES

**bookServices.js 🡪** This contains all the API calls to POST, PUT, DELETE, GET books, and GET a book by its ID.

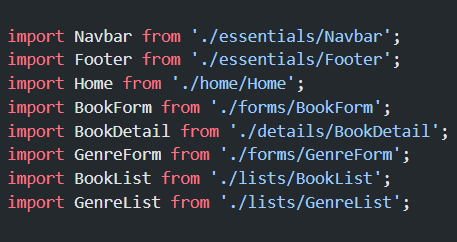


**genreServices.js 🡪** This contains This contains all the API calls to POST, PUT, DELETE, GET genre, and GET a genre by its ID.



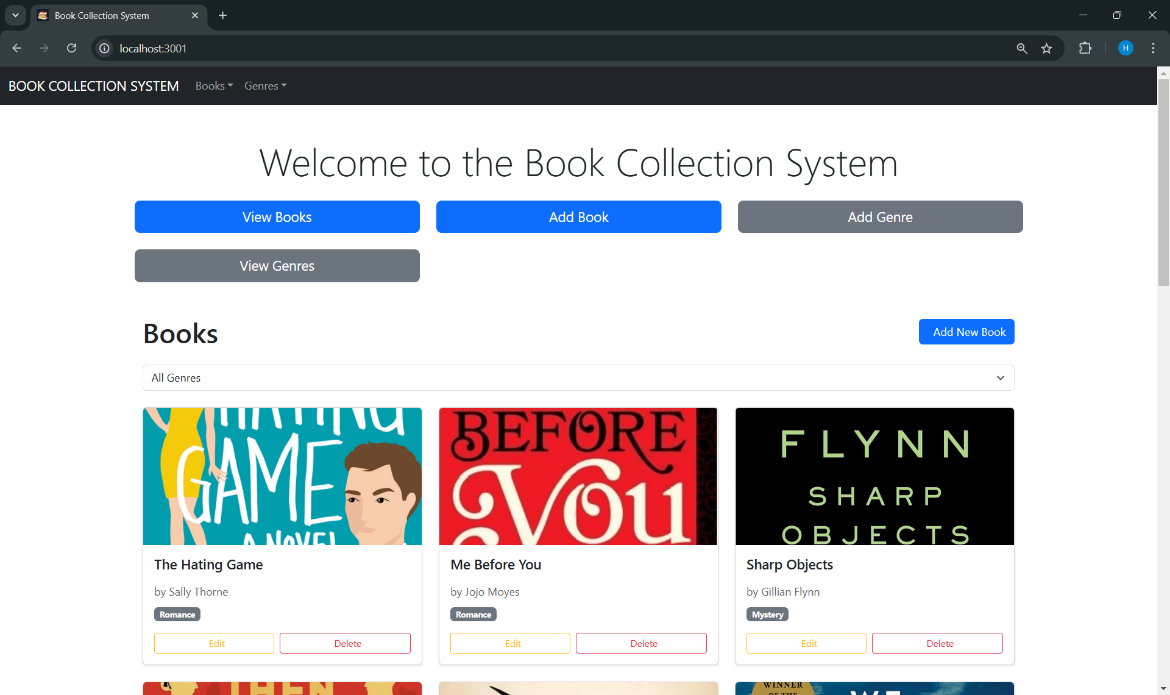
Each fetch function is a module that is exported to be used in the other components (eg. BookList.js imports the fetchBooks function to view all the books).

**App.js (All Client-Side Components)**

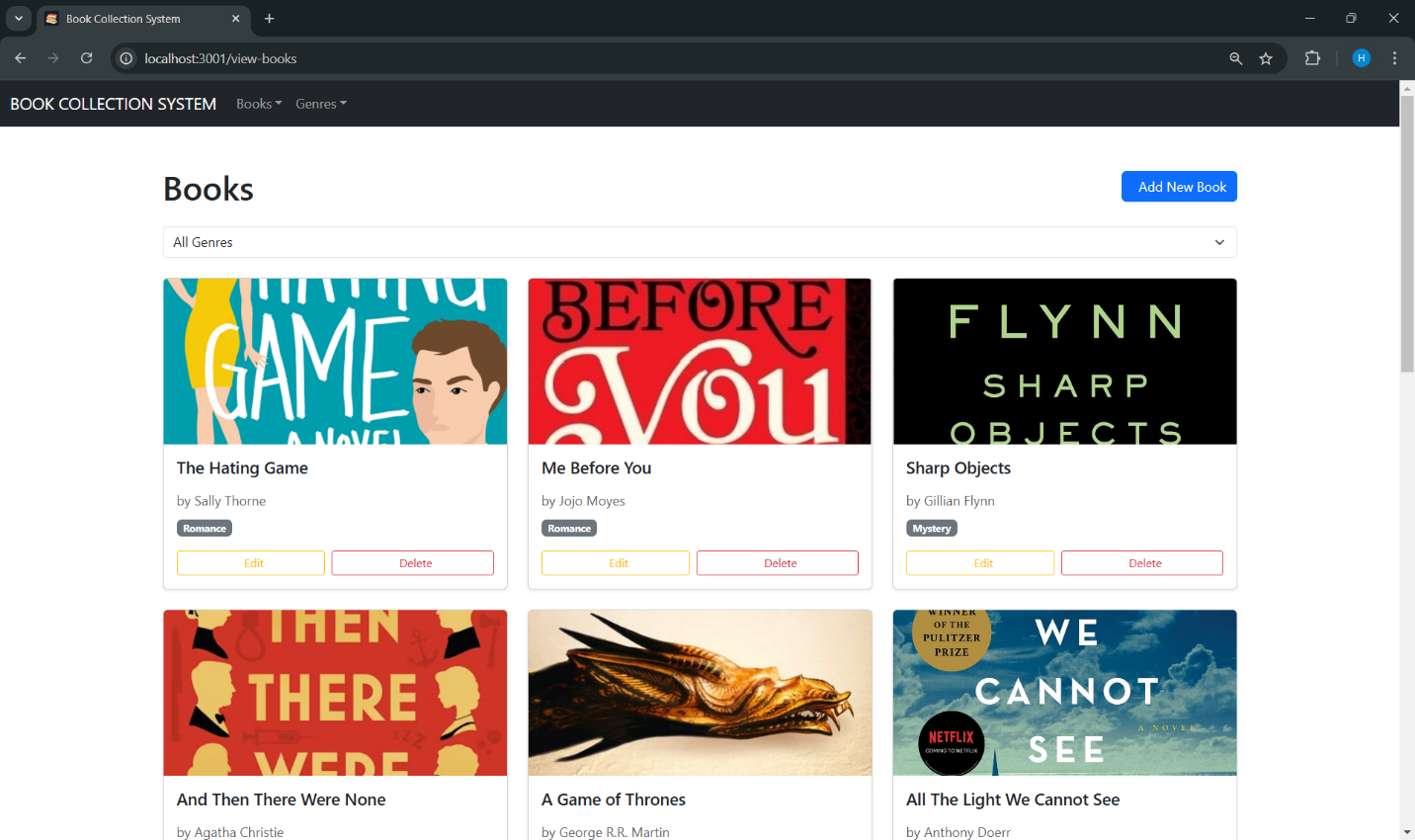


## Screenshots of Application

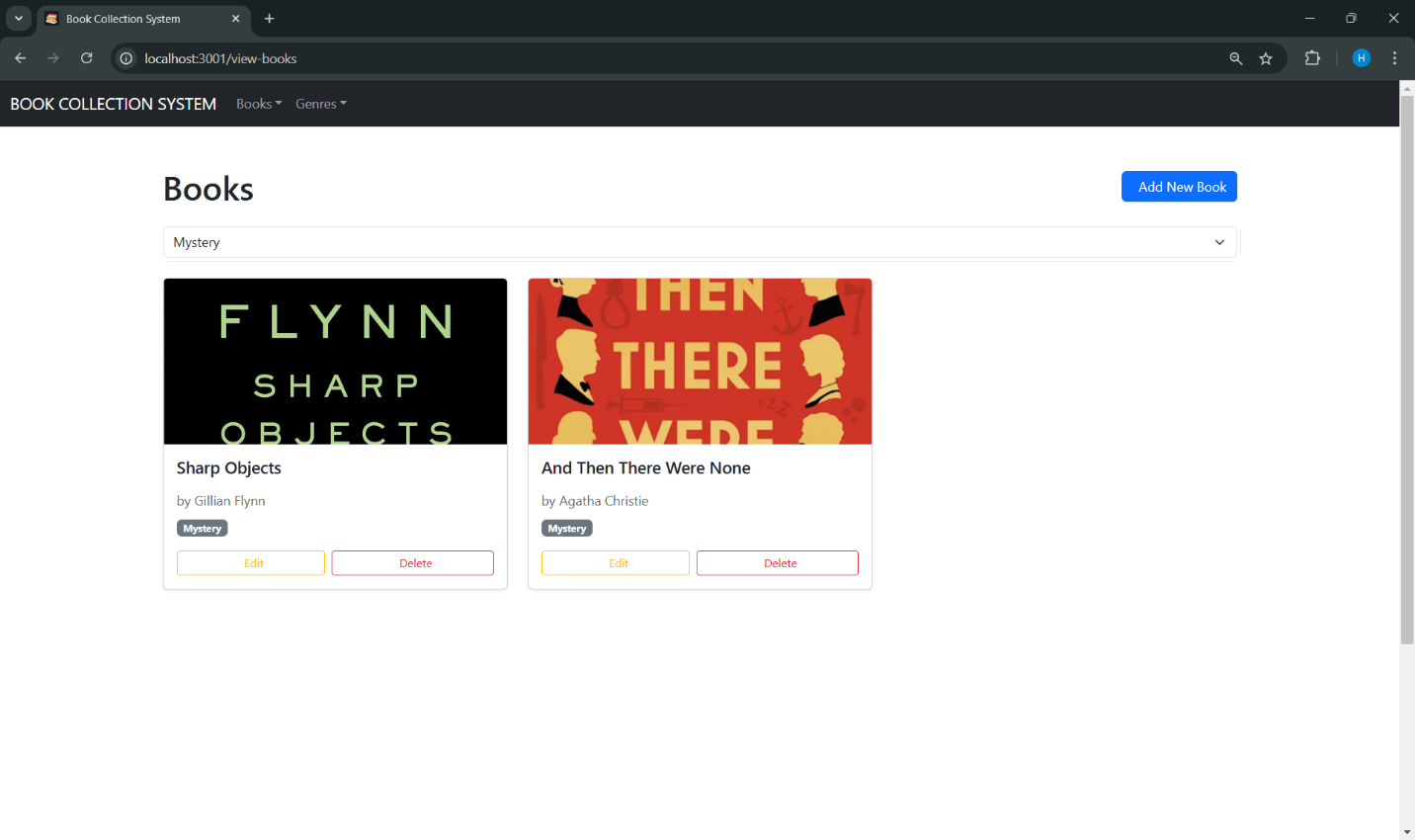
* **Home.js**



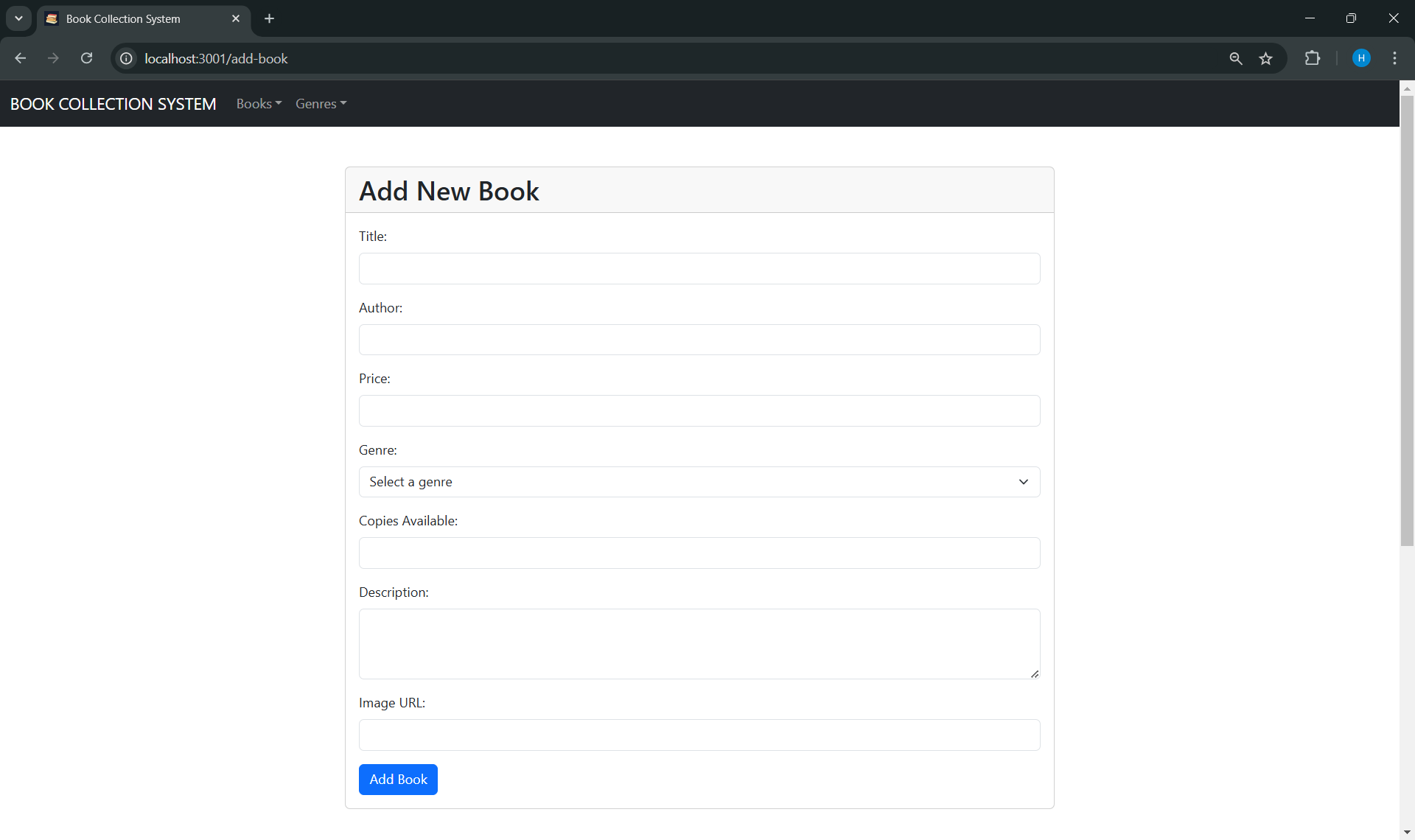
* **BookList.js**



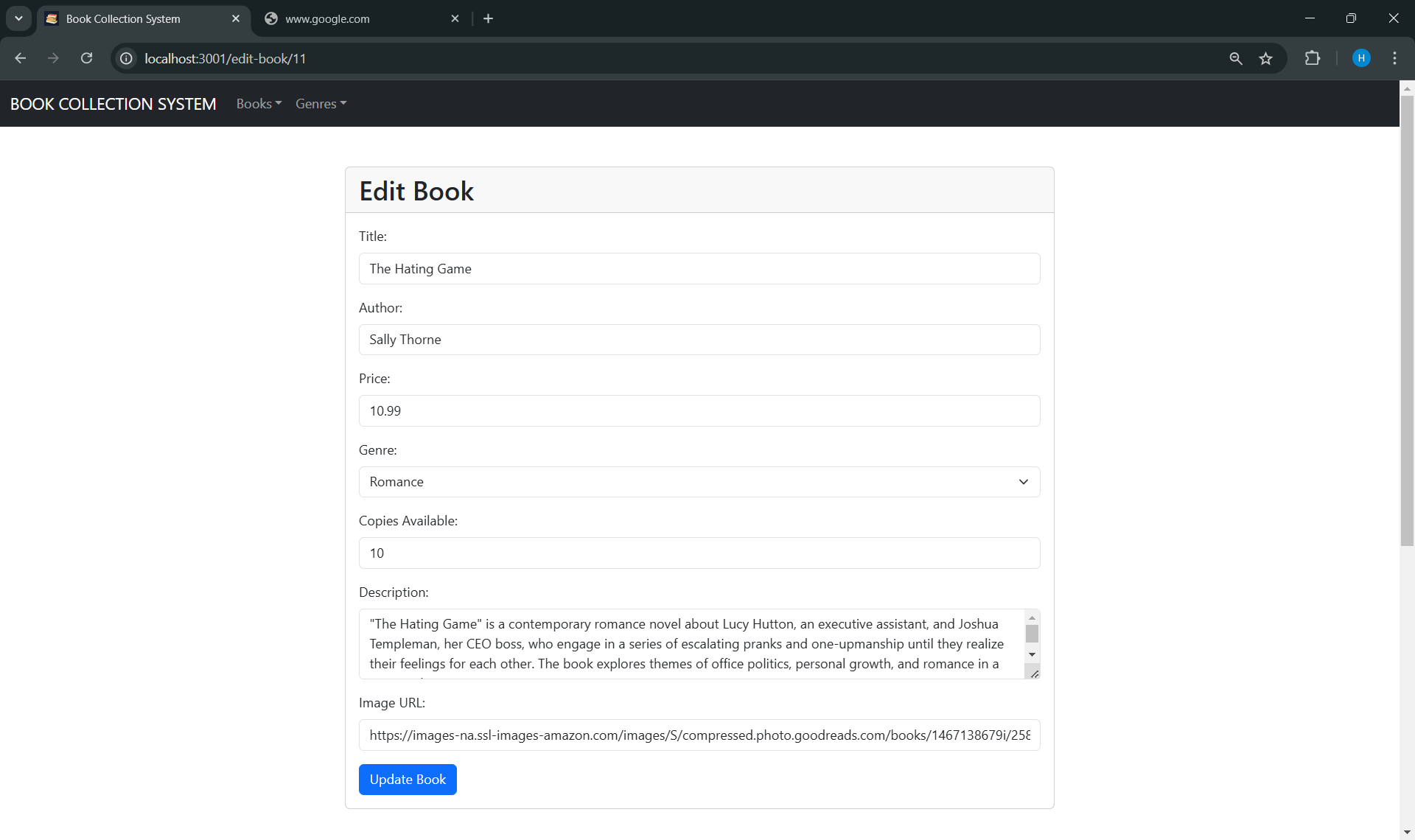
* **BookList.js (Filtering Mystery Books)**



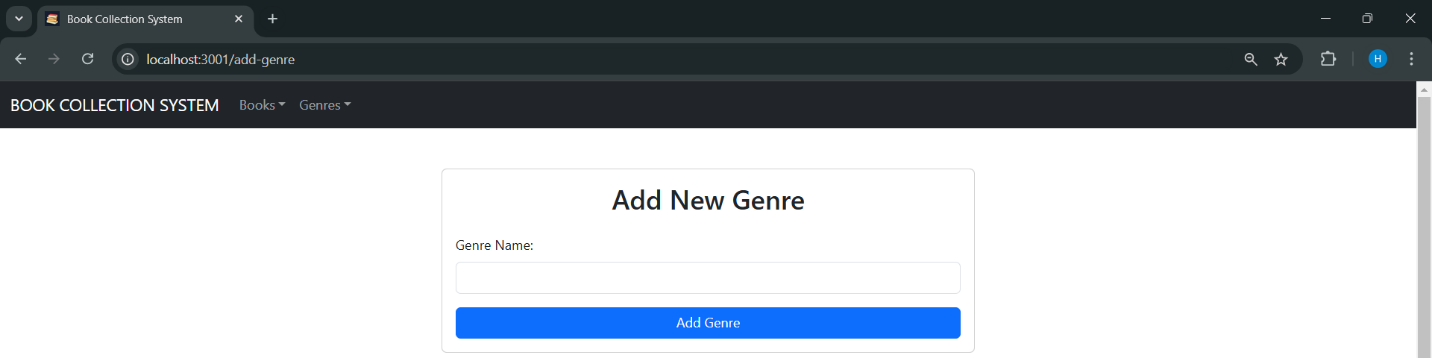
* **BookForm.js**



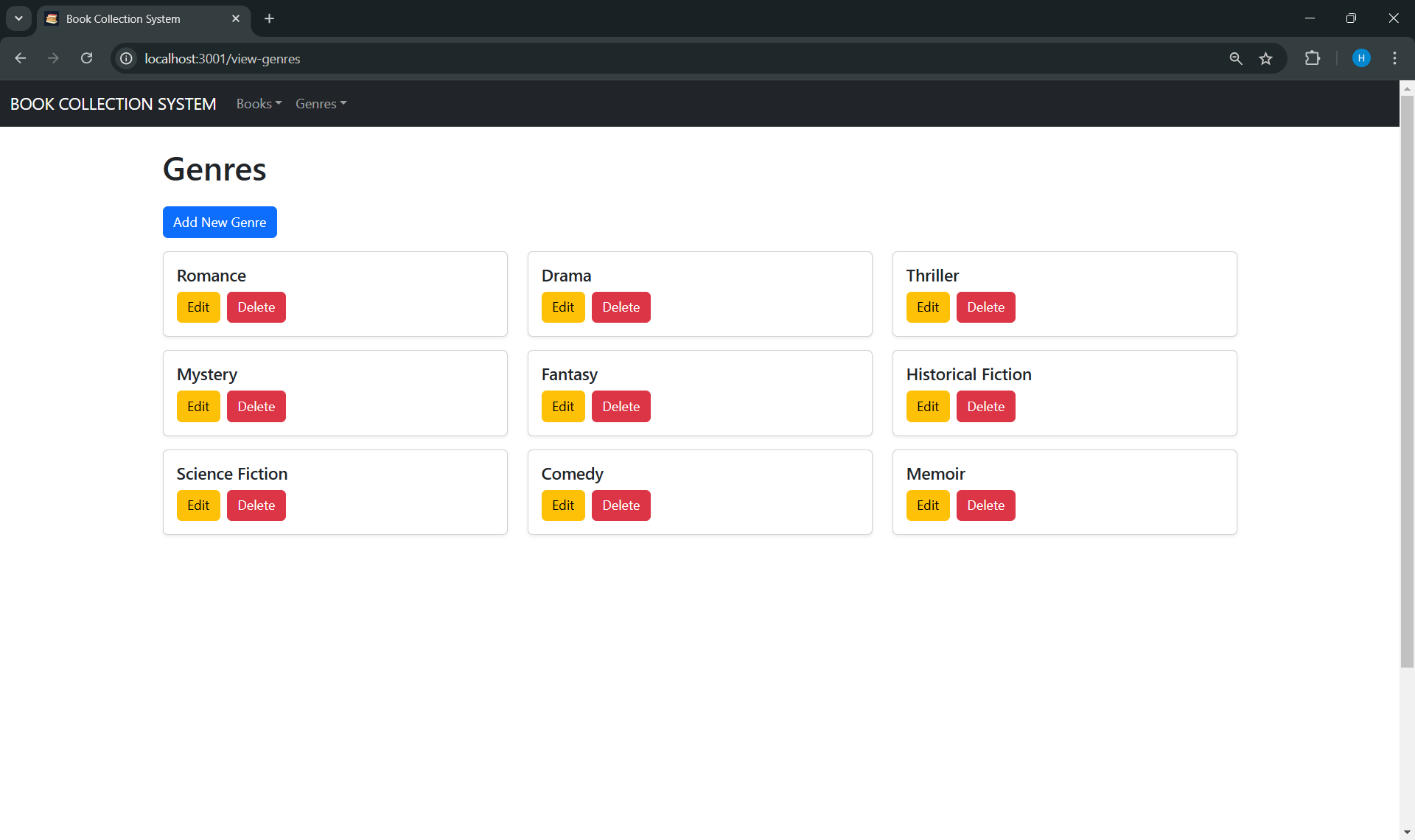
* **BookForm.js (for Editing a Book)**



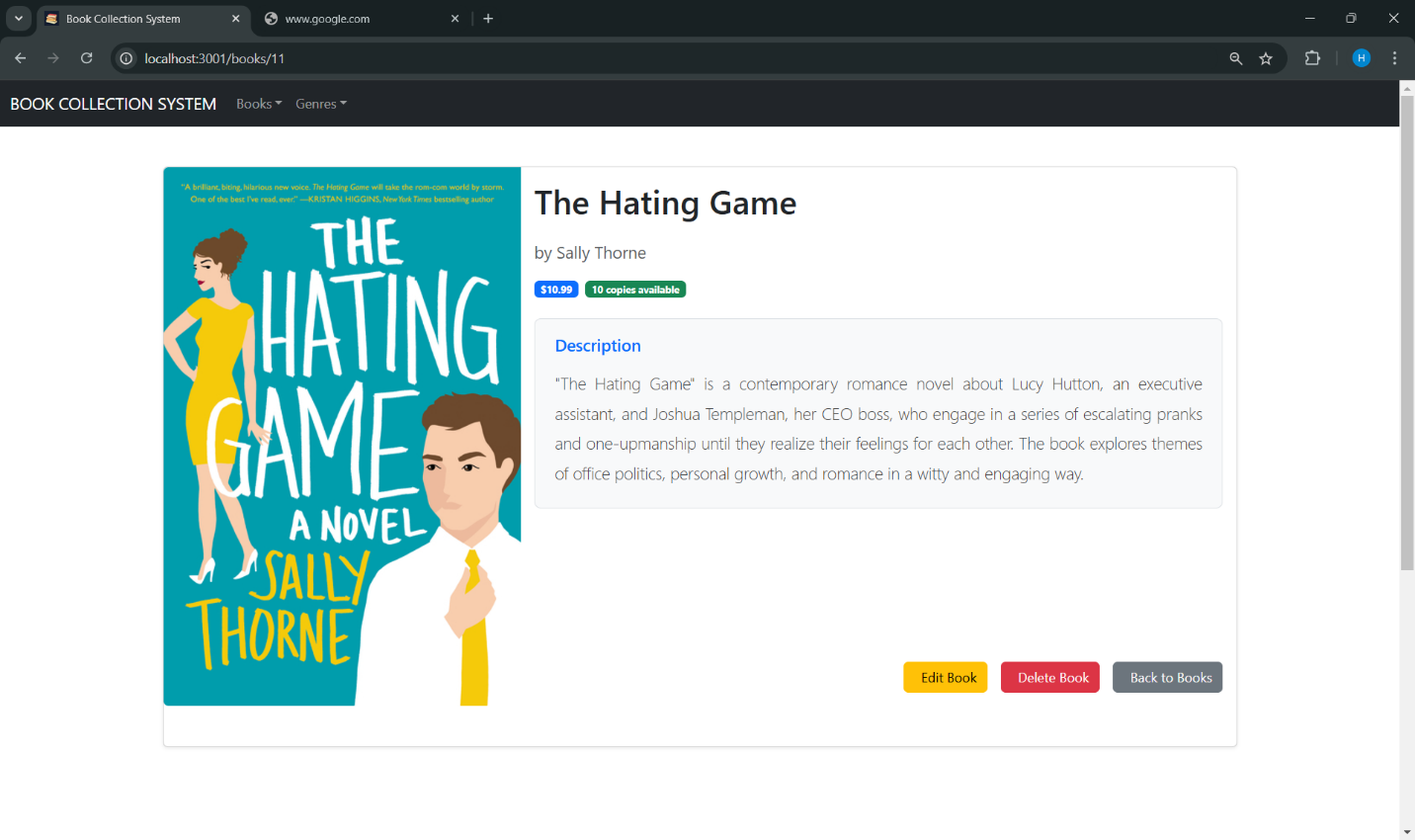
* **GenreForm.js**



* **GenreList.js**



* **BookDetail.js**



# REFLECTION QUESTIONS

## ****How does your implementation align with the client-server architecture principles?****

Our implementation adheres to the client-server architecture by clearly separating the concerns of the client (React frontend) and the server (Node.js backend). Here's how it aligns with the principles:

* **Separation of Concerns:**  
  The React frontend handles the user interface and presentation logic, while the Node.js backend focuses on application logic, database interactions, and API endpoints. This ensures a modular and scalable architecture.
* **Stateless Communication:**  
  The backend APIs are designed to follow RESTful principles, where each request contains all the necessary information to process it, making the communication stateless. This simplifies server logic and ensures scalability.
* **Centralized Server Logic:**  
  The Node.js backend serves as the centralized hub for business logic and interacts with the database via Sequelize ORM. It handles routes for books and genres, processes client requests, and ensures data consistency.
* **Static Asset Serving:**  
  After running npm run build for the React project, the static files are served from the build directory using Express. This ensures the React frontend is integrated with the Node.js backend, creating a seamless full-stack application.
* **Efficient Communication via APIs:**  
  The client fetches data from the server using RESTful APIs (e.g., /api/books and /api/genres). This encapsulation enables flexible client-server interactions and supports future scalability if I want to add mobile or other client applications.

By adhering to these principles, our implementation creates a decoupled and efficient system that supports maintainability, scalability, and future expansion.

## ****What challenges did you encounter while setting up the client-server communication?****

Setting up the client-server communication posed several challenges, which I resolved systematically. Here are the main challenges and how I addressed them:

**Challenge 1: Serving the React Build Output from Node.js**  
Initially, I had to ensure that the React application (after running npm run build) was correctly integrated into the Node.js server. Configuring the app.js to serve static files from the build folder required proper path handling and ensuring the frontend was accessible through the server root.

* **Solution:** I used express.static middleware to serve the build directory and added a wildcard route (\*) to ensure React could handle routing for all client-side paths.

**Challenge 2: CORS Issues**  
During development, the frontend (running on http://localhost:3000) and backend (on http://localhost:3001) Ire hosted on different ports. This caused CORS (Cross-Origin Resource Sharing) errors when the client attempted to fetch data from the server.

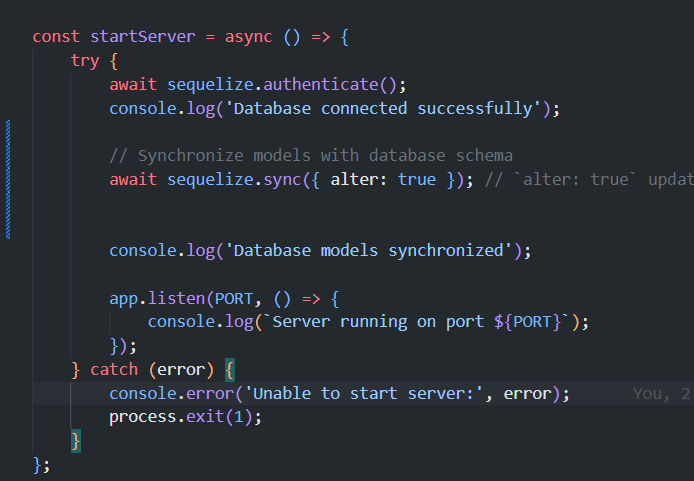
* **Solution:** I used the cors middleware in Express to allow cross-origin requests from the client by specifying http://localhost:3000 as an alloId origin. This resolved the CORS errors and enabled seamless communication betIen the frontend and backend.

**Challenge 3: Dropdown Functionality in Navbar**  
After integrating the React build with the Node.js server, I noticed that the Bootstrap dropdown functionality in the navbar was not working. This was due to missing Bootstrap JavaScript dependencies, which Ire not included in the npm run build output.

* **Solution:** I installed Bootstrap, Popper.js, and jQuery (for Bootstrap 4) or ensured Bootstrap’s JavaScript bundle (for Bootstrap 5) was included in the project. Additionally, I verified that the required scripts Ire correctly linked in the public/index.html file.

**Challenge 4: Database Synchronization and Error Handling**  
During the server setup, there Ire challenges with database synchronization using Sequelize, particularly when the database was not properly connected or tables Ire not defined correctly. There were also issues when making changes (adding a new attribute) to the Book model (table) using Sequelize. I added an image and description field after the initial database synchronization, but upon making POST requests from the frontend, neither field was being populated.

* **Solution:** I implemented error handling in the app.js file to log database connection issues and prevent the server from starting if the database was not synchronized. This ensured a robust connection between the server and the database.



## 3. ****What were the key design decisions made while building the system?****

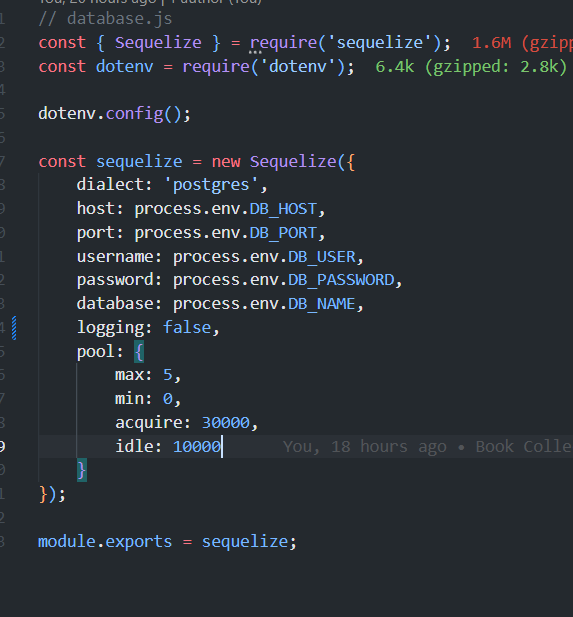
While building the system, several key design decisions were made to ensure efficiency, scalability, and maintainability:

a) **Client-Server Architecture**

We opted for a clear separation between the client (React-based frontend) and the server (Node.js + Express backend). This separation ensures independent development, testing, and deployment of both parts.

**b) Sequelize ORM**

We used Sequelize as the ORM to abstract database interactions, which simplifies database schema management, querying, and model associations (e.g., Book and Genre). This approach allows for easier management of the database schema through sync and migrations.



**c) Dynamic Frontend Rendering**

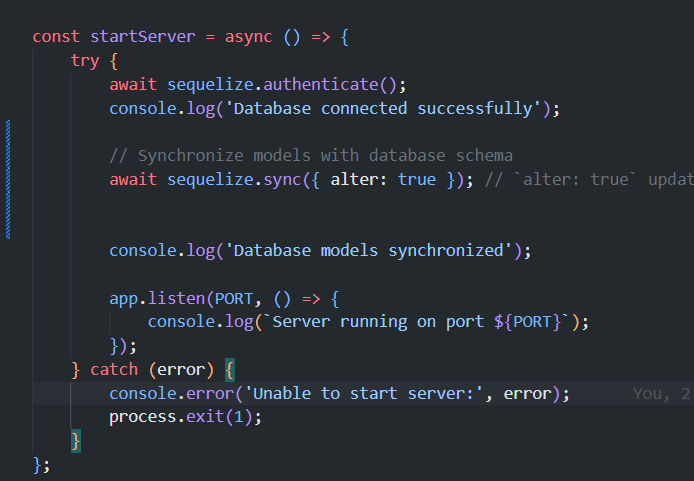
In the React frontend, dynamic routing and reusable components (e.g., Navbar and button grids) were employed to ensure that the application could scale as more views or pages are added without needing extensive code duplication.

**d) Build and Integration**

We decided to integrate the React app with the Node.js server using the npm run build output and configured Express to serve static files from the build directory. This eliminated the need for a separate hosting environment for the client, streamlining deployment.

e) **Database Synchronization**

For schema evolution, we chose to use Sequelize's sync with the { alter: true } option during development. This allowed us to update the database schema seamlessly while preserving data.



f) **RESTful API Design**

We designed the backend API to follow REST principles. Endpoints like /api/books and /api/genres were created to handle core functionality, ensuring a predictable and modular API structure that can be extended in the future.

g) **CORS Configuration**

To support client-server communication during development, we enabled CORS for http://localhost:3000, allowing the React frontend to interact with the Node.js backend securely.

h) **Error Handling**

Centralized error-handling middleware was implemented on the server to ensure robust handling of unexpected issues, enhancing reliability and user experience.

## 4. ****How does the separation of the client and server improve scalability and maintainability in this project?****

a) **Independent Scalability**

Separating the client (React) and server (Node.js) allows each component to scale independently:

* **Frontend:** The React app can be optimized for performance by implementing techniques like lazy loading or splitting bundles.
* **Backend:** The server can scale horizontally by adding more instances or vertically by improving database performance and increasing server resources.

b) **Modular Development**

With a clear boundary between the client and server, the development of both components can occur in parallel. Changes to one part do not directly affect the other, reducing the risk of introducing bugs.

c) **Improved Deployment Flexibility**

The separation allows the client and server to be hosted and deployed independently. For instance:

* The React app can be hosted on a CDN (e.g., AWS S3, Cloudflare) for faster delivery.
* The Node.js server can be deployed on a backend service (e.g., Heroku, AWS, or Docker).

d) **Maintainability**

* **Frontend:** The use of reusable React components (like Navbar and buttons) ensures that any updates to shared elements propagate throughout the app. Additionally, React's virtual DOM makes UI updates more efficient.
* **Backend:** The modular nature of the Express app (with separate route files for books and genres) makes it easier to add new features without cluttering the codebase.

e) **Easier Debugging**

With separate client and server layers, issues can be isolated to either the frontend or the backend, reducing debugging complexity. For example, we could easily identify whether the problem with dropdown functionality was due to React or Bootstrap.

f) **Enhanced Security**

By separating the client and server:

* Sensitive business logic and database operations remain on the server, reducing the risk of exposing vulnerabilities on the client side.
* APIs can be protected using authentication mechanisms (e.g., tokens), preventing unauthorized access.

g) **Future Proofing**

* The React frontend can be replaced with a different technology (e.g., Vue or Angular) without changing the server.
* The backend API can support multiple clients (e.g., mobile apps, other web apps) without requiring significant changes.

In summary, this separation improves the system's ability to adapt, scale, and be maintained in the long term while allowing for flexibility and clean architecture principles.

# CONCLUSION

This project successfully implements a scalable and maintainable web application using a modern **client-server architecture**. The **React-based frontend** delivers a dynamic user experience, while the **Node.js and Express backend**, supported by **Sequelize ORM**, ensures efficient data management and business logic handling.

Key design choices, such as RESTful APIs, reusable components, and clear separation of concerns, enhance flexibility and scalability. Challenges like synchronizing the database schema and resolving UI issues were addressed systematically, ensuring a seamless integration of the React build with the backend.

By separating the client and server, the system is future-proof and adaptable to evolving needs, allowing for easy updates and the potential integration of additional clients like mobile apps. This well-structured approach ensures a robust foundation for ongoing development and scalability.