UNIVERSITY OF GHANA

COLLEGE OF BASIC AND APPLIED SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

SCHOOL OF ENGINEERING SCIENCES

SECOND SEMESTER 2023/2024 ACADEMIC YEAR

CPEN 421 – MOBILE AND WEB SOFTWARE DESIGN AND ARCHITECTURE

MAWUENA EFUA AGBESI

ID: 10941083

LAB 3: Exploring Model-View-Controller Architecture in Web Applications

ABSTRACT
This project applies the Model-View-Controller (MVC) architecture in developing a web application for managing a book collection. Core features, including adding, viewing, editing, and deleting books, are implemented using RESTful APIs and HTTP protocols. The backend leverages Node.js and Express, structured into models, views, and controllers to enhance scalability and maintainability. PostgreSQL is used for persistent data storage, while the frontend employs HTML, Bootstrap, and JavaScript to provide a user-friendly interface. This report examines the system's design, implementation methodology, challenges faced, and its alignment with MVC architecture principles.

INTRODUCTION

The Model-View-Controller (MVC) architecture is a popular design pattern that promotes modularity, scalability, and maintainability in application development. By distinctly separating the data layer (model), the user interface (view), and the application logic (controller), MVC simplifies the development process and enhances code organization.

This project employs MVC principles to create a Book Collection Management System, consisting of a backend server integrated with a PostgreSQL database and a frontend client that communicates via RESTful APIs. The goal is to deliver a modular, efficient, and intuitive application for managing book collections.

This report outlines the implementation process, encompassing database schema design, MVC-driven backend and frontend development, system testing, and deployment. It further discusses how the MVC architecture contributes to the system’s modular design, scalability, and ease of maintenance.

METHODOLOGY

1. Database Design

- Schema Design:
  - Books: Includes id (primary key), title (string), author (string), price (float), genre_id (foreign key), and copies_left (integer).
  - Genres: Includes id (primary key) and name (string).
- Database Management System: PostgreSQL was used for its robust handling of relational data.
- Setup: Tables were created with primary and foreign key constraints. Data types were chosen to ensure precision (e.g., float for price).

2. Backend Development

- Framework: Node.js and Express were used for simplicity and efficiency in handling HTTP requests.

- **Architecture**: The backend adhered to the MVC pattern:
  - **Models**: Represented the database tables (Books and Genres) using Sequelize ORM for object-relational mapping.
  - **Views**: Templates built with EJS rendered dynamic HTML pages.
  - **Controllers**: Handled the application logic for CRUD operations.

- Endpoints:
- GET /books: Retrieve all books in the collection.
- GET /books/:id: Fetch a specific book by ID.
- POST /books: Add a new book.
- PUT /books/:id: Update an existing book.
- DELETE /books/:id: Remove a book by ID.
- GET /genres: Retrieve all genres for filtering.

3. Frontend Development

HTML and Bootstrap: The frontend used Bootstrap for styling, ensuring a clean, responsive design without custom CSS. EJS templates were used to render dynamic pages with real-time updates from the backend.

Features:

- A form to add and edit books.
- A table displaying the book list with options to edit and delete entries.
- A search bar for querying books by title or author.
- A genre-based filter to sort books dynamically.
- JavaScript enabled dynamic interaction and real-time UI updates.
- The Fetch API was used to send HTTP requests to the backend and update the UI based on server responses.

4. Testing

- Postman: API endpoints were tested using Postman to ensure they returned correct responses for valid and invalid inputs.
- Integration Testing: The frontend and backend were tested together to verify seamless communication.
- Scenarios: Operations such as adding, editing, and deleting books were tested for functionality and edge cases.

5. Deployment

Server Deployment: The backend was deployed on Render to provide a live server for the client to interact with.

Frontend Deployment: The client was deployed on Vercel, ensuring the application was accessible online.

LINK: https://book-collection-client.vercel.app-

RESULTS AND DISCUSSION

Results and Discussion

The implementation of the Book Collection System successfully demonstrated the principles of MVC architecture by separating concerns between the client and server, leveraging RESTful APIs, and achieving effective communication via HTTP. Below are the key outcomes and an in-depth discussion of the results:

Results

1. **Core Functionalities**
    - **Add a New Book**: Users can input book details (e.g., title, author, price, genre, and copies available) via a form. The data is successfully sent to the server using a `POST` request and stored in the PostgreSQL database.
    - **View All Books**: A dynamically updated list of books is displayed on the client-side by fetching data from the server using a `GET` request.
    - **View Book Details**: Users can retrieve detailed information about a specific book by clicking on it, triggering a `GET` request with the book's unique ID.
    - **Edit Book Details**: Book details can be modified through a form that uses a `PUT` request to update the data in the database.
    - **Delete a Book**: Users can remove a book from the collection by clicking a delete button, which sends a `DELETE` request to the server.
2. **Database Operations**
    - The PostgreSQL database was designed with two main entities: `Books` and `Genres`.
    - Relationships and data types were well-defined, ensuring seamless querying and manipulation of data.
3. **User Interface**

    • The user interface, built with Bootstrap and EJS templates, is clean, responsive, and user-friendly.

- Dynamic updates ensure a seamless user experience without the need for manual page refreshes.

System Communication

  - HTTP requests from the client were handled correctly by the server, ensuring smooth communication between the two components.
  - The RESTful API endpoints were tested rigorously using Postman to verify their functionality.

Discussion

 Adherence to MVC Principles
- The MVC architecture facilitated clear separation of concerns:
  - **Model**: Encapsulated database logic, enabling efficient interaction with PostgreSQL through Sequelize.
  - **View**: Presented dynamic content using EJS templates, making it easy to render data fetched from the server.
  - **Controller**: Centralized application logic, managing user requests and server responses.
- This separation improved code readability, maintainability, and scalability.

 Challenges Faced
- **Database Connection**:
  - Configuring the PostgreSQL database for deployment required careful handling of environment variables and SSL settings.
- **Search and Filter Integration**:
  - Implementing simultaneous search and filter functionalities required precise handling of query parameters and database queries.
- **Dynamic View Updates**:
  - Ensuring that the views reflected real-time changes (e.g., after adding or editing a book) was initially challenging but resolved using the Fetch API and proper routing.

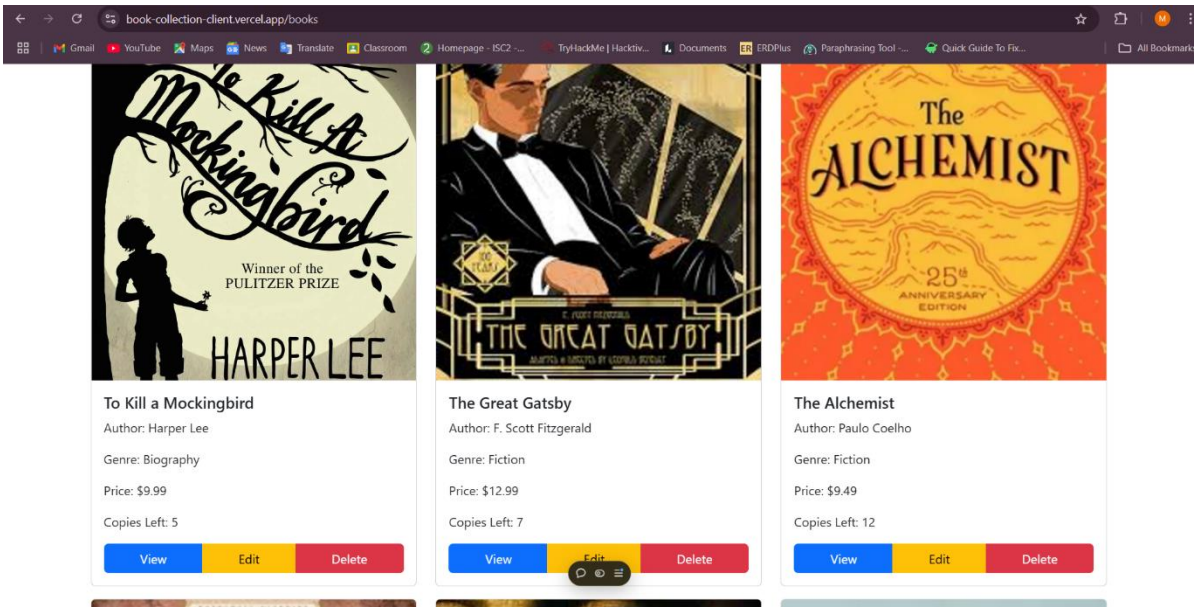
 System Performance
- The application performed efficiently, even with multiple concurrent requests.
- Render and Vercel provided reliable hosting solutions, ensuring low latency and high availability.

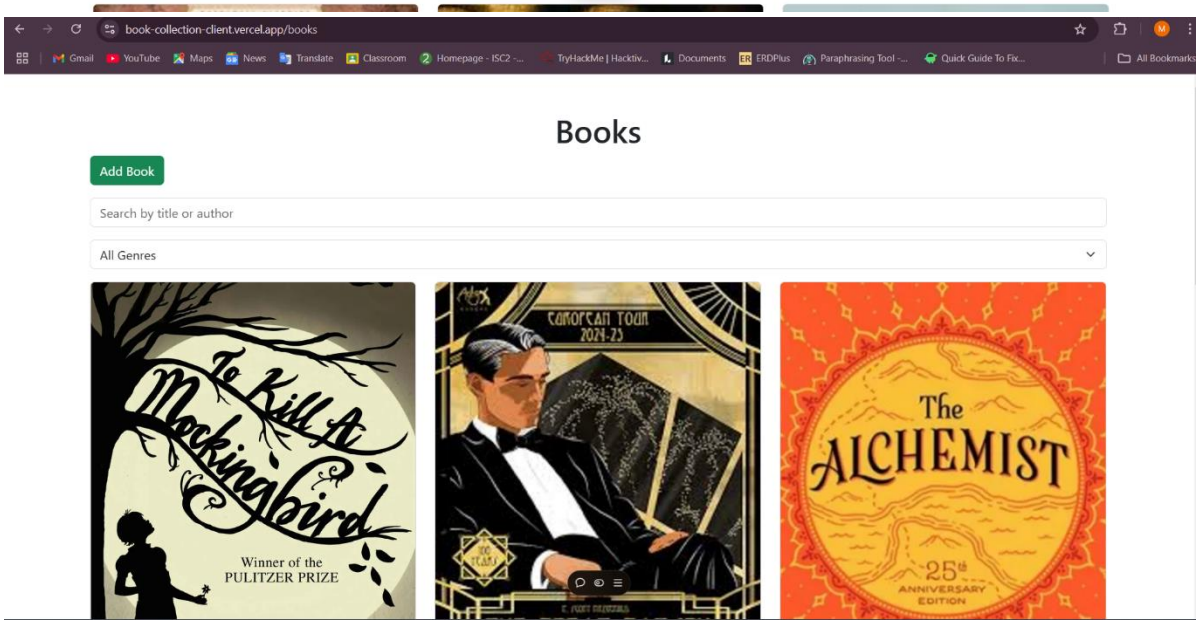
 User Experience
- The responsive design, combined with the intuitive interface, ensured a positive user experience.
- The integration of search and filter functionalities added convenience and efficiency for users managing large book collections.

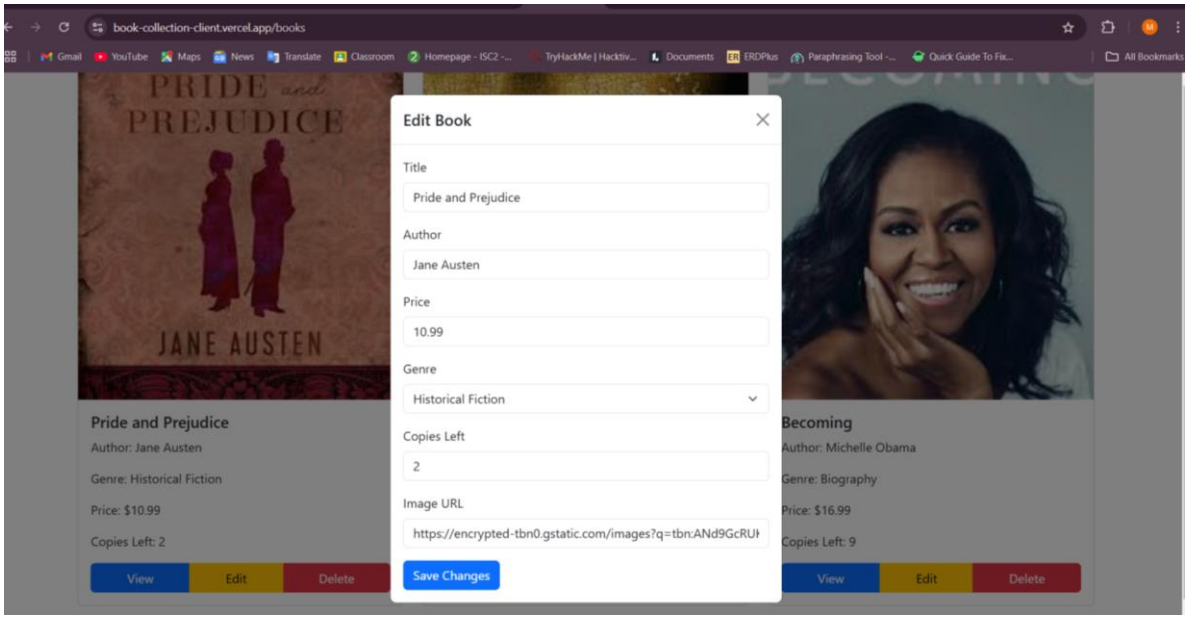
 Impact of MVC on Development

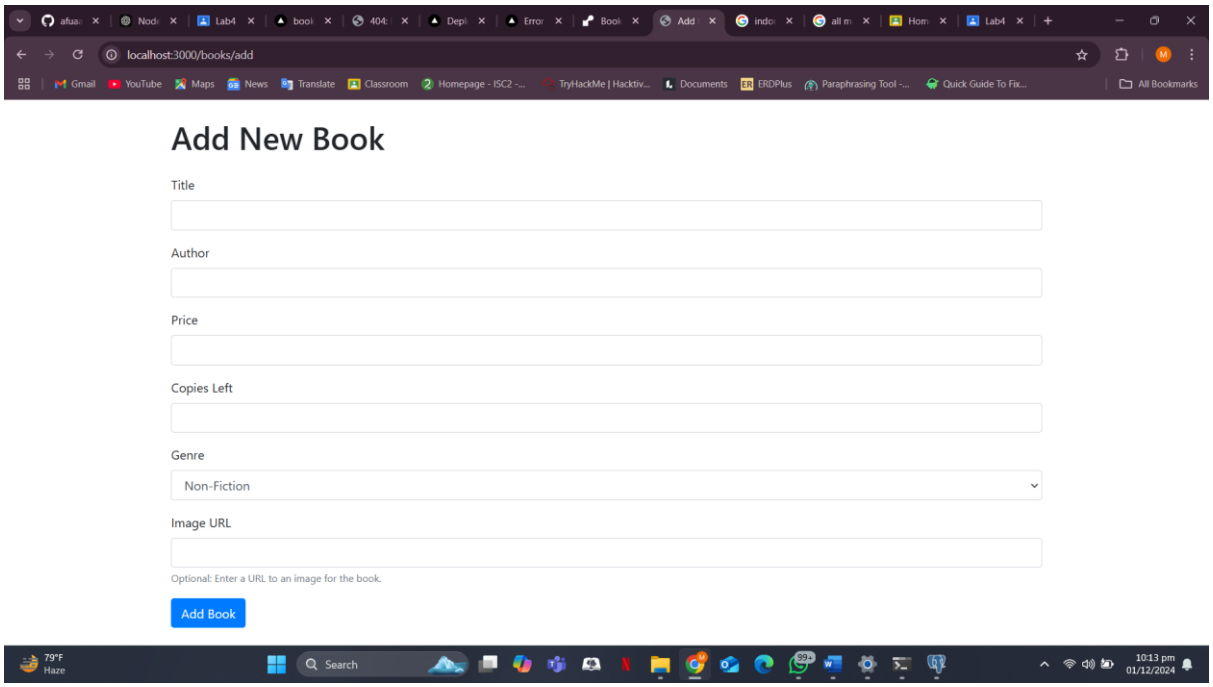
- The MVC architecture simplified debugging and allowed independent modification of components.
- Future enhancements (e.g., user authentication or analytics) can be integrated seamlessly without disrupting existing functionality.

FRONTEND OF ASSIGNMENT









ANSWERS TO QUESTIONS

Answers

1. What were the key design decisions made while building the system?

1. **Adopting the MVC Architecture**:
    - The decision to use the Model-View-Controller (MVC) architecture was crucial to achieving a clear separation of concerns. This allowed models to handle database logic, views to manage the user interface, and controllers to handle application logic, ensuring organized and maintainable code.

2. **Database Design**:
    - A relational schema was chosen to manage the book collection effectively, with tables for books and genres linked through foreign keys. PostgreSQL was selected for its robust handling of relational data.
3. **Framework and Tools**:
    - Node.js and Express were selected for backend development due to their efficiency in handling HTTP requests and extensibility with middleware.
    - Sequelize ORM was used to simplify database operations, improve security against SQL injection, and enable easy migrations.
4. **Frontend Framework**:
    - Bootstrap was chosen for styling to create a responsive and professional-looking interface without the complexity of custom CSS.
    - EJS was used as the templating engine to dynamically render data from the backend.
5. **Deployment**:
    - Render was used for deploying the backend and database due to its support for Node.js applications and PostgreSQL.
    - Vercel was chosen for the frontend for its simplicity and ability to host static files with dynamic integration through API calls.
6. **Search and Filter Integration**:
    - The system incorporated a search bar for title/author searches and a genre filter dropdown. Both were implemented using query parameters to ensure flexibility and scalability.

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

1. **Scalability**:
    - **Independent Scaling**: The server (backend) and client (frontend) are deployed separately, enabling them to be scaled independently based on demand. For

example, if user traffic increases, the backend server can be upgraded without affecting the frontend deployment.

- **Flexible Database Integration**: Since the database is hosted on a separate platform, it can be scaled independently to handle larger datasets or more queries.

2. **Maintainability**:
    - **Modular Codebase**: The MVC architecture ensures that changes to the user interface (view) do not affect backend logic (controller) or database models, and vice versa. This modular approach simplifies debugging and updating the system.
    - **Easier Updates**: Features can be added to the client or server independently. For instance, a new API endpoint can be added to the backend without requiring changes to the frontend until necessary.
    - **Reusability**: The backend can be reused for other clients, such as a mobile application, by exposing the same RESTful API.
3. **Enhanced User Experience**:
    - A separate client ensures a faster, more responsive interface, as it can fetch only the required data from the server without reloading the entire application.
4. **Future-Proofing**:
    - This separation allows easy integration of new technologies or frameworks. For example, the current frontend can be replaced with a React or Angular-based interface without altering the backend.

By separating the client and server, the project adheres to modern web development practices, ensuring long-term scalability and maintainability while delivering an efficient and user-friendly application.

CONCLUSION

Conclusion

The successful implementation of this project highlights the benefits of using the MVC architecture to develop a scalable, maintainable, and user-friendly book collection management system. By separating the client, server, and database layers, the project achieves

a modular structure that facilitates independent development, testing, and deployment of each component.

Key features such as a search bar, genre-based filtering, and CRUD operations were efficiently integrated using RESTful APIs and a PostgreSQL database, ensuring a robust backend. The frontend, styled with Bootstrap and dynamically rendered using EJS, delivers an intuitive and responsive user experience. Deployment on Render and Vercel further demonstrates the flexibility of the architecture in adapting to modern hosting environments.

Challenges faced during the project, including database integration and API testing, were mitigated through rigorous testing and adherence to best practices. This iterative approach ensured the system's reliability, security, and performance.

In conclusion, the project successfully met its objectives, showcasing how MVC architecture and modern web technologies can be applied to build scalable and maintainable web applications. It also serves as a strong foundation for future enhancements, such as adding user authentication, implementing advanced search capabilities, or integrating analytics features. This work underscores the importance of structured design and systematic development in achieving project goals efficiently and effectively.