

Software Engineering Internship – Assignment

# **Library Management System**

By Mihika Pathirana

21<sup>th</sup> November 2024

## Table of Contents

<b>1. Introduction</b>	3
<b>2. Development Process</b>	4
2.1. Backend Implementation	4
Overview of the backend architecture	4
Technologies Used For Backend Development	4
Backend Process	6
Error Handling and Validation Mechanisms	11
Challenges Faced During Backend Development	13
2.2. Frontend Implementation	16
Overview of the Frontend Architecture	16
Technologies Used For Frontend Development	17
Implementation of CRUD Operations	18
State Management and Routing Strategies	21
Validation and Error Handling on the Frontend	23
Challenges Faced During Frontend Development	26
<b>3. User Interface Designs</b>	28
<b>4. Additional Features Developed</b>	32
<b>5. Key Insights and Learnings</b>	34
<b>6. Conclusion</b>	36

# 1. Introduction

Managing library resources efficiently is a critical aspect of modern libraries, where digital solutions play a transformative role. The Library Management System is a comprehensive application designed to streamline administrative tasks, enhance user experience, and ensure secure access to valuable resources. This project incorporates state-of-the-art web development techniques to deliver a robust, scalable, and user-friendly solution tailored to meet the needs of library administrators.

The system lies its ability to perform essential operations seamlessly, empowering administrators to manage the library effectively. The application enables core functionalities such as adding, viewing, updating, and deleting book records (CRUD operations), ensuring data accuracy and consistency. Additionally, it integrates user authentication mechanisms to safeguard sensitive information and restrict access to unauthorized users.

The Library Management System boasts a responsive and intuitive user interface, compatible with a variety of devices including desktops, tablets, and smartphones. This ensures an optimal experience for users across all platforms. The backend, powered by C# and .NET, communicates efficiently with the frontend, developed using React and TypeScript, through well-defined RESTful APIs.

## Key Features:

- **CRUD Operations:** Effortlessly manage book records, enabling administrators to create, read, update, and delete information with precision.
- **User Authentication:** Secure access through a login and registration system, ensuring only authorized users can perform administrative functions.
- **Responsive Design:** A user-friendly interface optimized for different screen sizes, providing a seamless experience across devices.
- **Scalability and Integration:** A robust architecture designed for future enhancements, leveraging modern development frameworks and best practices.

By combining these features into one cohesive system, the Library Management System addresses the challenges faced by traditional libraries, paving the way for a more organized and efficient resource management experience.

## 2. Development Process

### 2.1. Backend Implementation

Overview of the backend architecture.

The backend of the Library Management System was designed to efficiently handle operations like book management and user authentication. It employs a layered architecture to separate concerns, ensuring scalability, maintainability, and security. The architecture consists of the following layers:

- **Models:** Represent the application's data structure.
- **Database Context:** Facilitates interaction with the database.
- **Controllers:** Handle HTTP requests and implement business logic.
- **Security Measures:** Incorporate password hashing for secure user authentication.

This modular architecture ensures flexibility, making it easier to add features and maintain the system over time.

### Technologies Used For Backend Development

The backend of the Library Management System is built using modern technologies and frameworks to ensure robustness, scalability, and ease of development. Below are the primary technologies used:

#### 1. Programming Language

- **C#:** The backend is developed in C#, a versatile and powerful language commonly used for enterprise applications.

#### 2. Framework

- **ASP.NET Core:** A high-performance, cross-platform framework used for building the backend. It supports RESTful APIs, middleware integration, and dependency injection, making it ideal for scalable web applications.

#### 3. Database

- **SQLite:** A lightweight, serverless database used for data storage. It simplifies setup and ensures compatibility with Entity Framework Core for seamless integration.

#### 4. **Object-Relational Mapper (ORM)**

- **Entity Framework Core:** A modern ORM that enables interaction with the database using LINQ (Language-Integrated Query). It automates database queries and updates, simplifying CRUD operations.

#### 5. **Security**

- **BCrypt.Net:** A library used for hashing passwords. It ensures secure storage and verification of user credentials, enhancing the application's security against breaches.

#### 6. **HTTP Client**

- **Swagger UI:** Integrated with ASP.NET Core to test and validate API endpoints. It provides an interactive user interface for exploring the API, ensuring proper functioning of CRUD operations like creating, retrieving, updating, and deleting records.

#### 7. **Development Environment**

- **Visual Studio Code:** A lightweight, yet powerful editor used for writing, debugging, and testing the backend code. Its extensions for C# and .NET Core enhance the development experience.

#### 8. **Version Control**

- **Git and GitHub:** Used for source code version control, collaboration, and repository management. This ensures proper tracking of changes and facilitates teamwork.

## Backend Process

The backend process of the Library Management System is structured into three primary components: Models, Database Context, and Controllers. Each plays a distinct role in ensuring smooth data handling, logical operations, and seamless communication between the application and the database.

### 1. Models

The models represent the data structure of the application and serve as the blueprint for the database. Each model defines the properties and relationships of a specific entity, such as books and users. This layer is directly mapped to the database tables using **Entity Framework Core**, providing a seamless connection between the application and the database.

- Book Model

The Book model represents the book entity in the library system. It includes properties such as Id, Title, Author, and Description. These attributes capture essential information about each book.



```
1 namespace LibraryManagementSystemBackend.Models
2 {
3     public class Book
4     {
5         public int Id { get; set; }
6         public string Title { get; set; }
7         public string Author { get; set; }
8         public string Description { get; set; }
9     }
10 }
11 }
```

Figure 1: Code Snippet of Book Model

- User Model

The User model represents application users and includes validation attributes to ensure robust user authentication. The Password and ConfirmPassword fields ensure passwords are securely stored and matched during registration.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in C# and defines a User model within the LibraryManagementSystemBackend.Models namespace. The User class has four properties: Id (int, required, key), Username (string, required, max length 50), Password (string, required, max length 50), and ConfirmPassword (string, required, max length 50, compared to Password with an error message "Passwords do not match.").

```
1 namespace LibraryManagementSystemBackend.Models
2 {
3     public class User
4     {
5         [Key]
6         public int Id { get; set; }
7
8         [Required]
9         [MaxLength(50)]
10        public string Username { get; set; }
11
12        [Required]
13        [MaxLength(50)]
14        public string Password { get; set; }
15
16        [NotMapped] // Ensure this is included
17        [Compare("Password", ErrorMessage = "Passwords do not match.")]
18        public string ConfirmPassword { get; set; }
19    }
20 }
21
```

Figure 2: Code Snippet of User Model

- Login Request DTO

The LoginRequestDto model is a lightweight data transfer object to handle login requests. It ensures the separation of concerns by limiting unnecessary data exposure during login.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in C# and defines a LoginRequestDto model within the LibraryManagementSystemBackend.Models namespace. The LoginRequestDto class has two properties: Username (string) and Password (string), both with get and set methods.

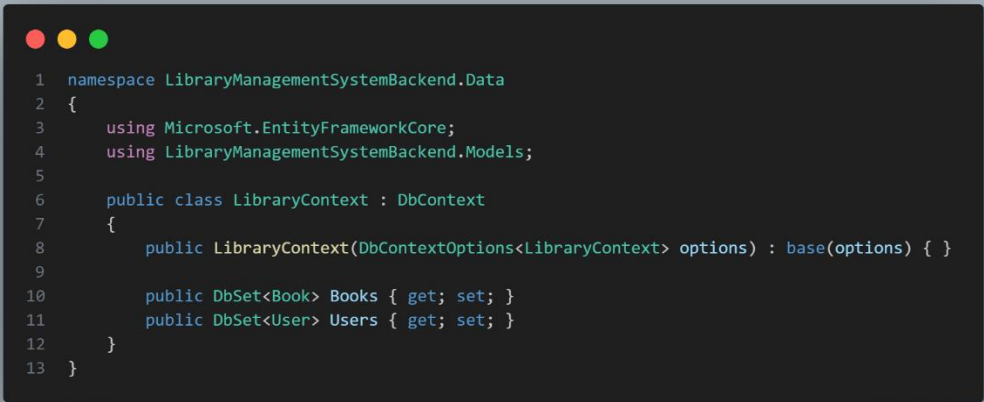
```
1 namespace LibraryManagementSystemBackend.Models
2 {
3     public class LoginRequestDto
4     {
5         public string Username { get; set; }
6         public string Password { get; set; }
7     }
8 }
```

Figure 3: Code Snippet of Login Request DTO Model

## 2. Database Context

The LibraryContext acts as the bridge between the models and the database. It uses Entity Framework Core to manage database operations for Book and User entities.

- It registers Books and Users as DbSet<T> for CRUD operations.
- It simplifies database interaction by providing methods for querying and saving changes.



```
1 namespace LibraryManagementSystemBackend.Data
2 {
3     using Microsoft.EntityFrameworkCore;
4     using LibraryManagementSystemBackend.Models;
5
6     public class LibraryContext : DbContext
7     {
8         public LibraryContext(DbContextOptions<LibraryContext> options) : base(options) { }
9
10        public DbSet<Book> Books { get; set; }
11        public DbSet<User> Users { get; set; }
12    }
13 }
```

Figure 4: Code Snippet of Database Context

## 2. Controllers

The controllers handle the business logic and act as the intermediaries between the frontend and the database. They process incoming HTTP requests, interact with the database through the context, and return appropriate responses. For example:

- BookController: Manages CRUD operations for books.
- AuthController: Handles user registration and authentication.

### ➤ Book Controller

The BookController manages CRUD operations for books, including retrieving all books, retrieving a book by its ID, creating a new book, updating an existing book, and deleting a book.

### Key Endpoints:

- GET /api/Book - Retrieves all books.
- GET /api/Book/{id} - Retrieves a book by ID.
- POST /api/Book - Creates a new book.
- PUT /api/Book/{id} - Updates a book.
- DELETE /api/Book/{id} - Deletes a book.



```
1 // POST: api/Book
2 [HttpPost]
3 public async Task<ActionResult<Book>> CreateBook(Book book)
4 {
5     _context.Books.Add(book);
6     await _context.SaveChangesAsync();
7
8     return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
9 }
10
11 // PUT: api/Book/5
12 [HttpPut("{id}")]
13 public async Task<IActionResult> UpdateBook(int id, Book updatedBook)
14 {
15     if (id != updatedBook.Id)
16     {
17         return BadRequest();
18     }
19
20     _context.Entry(updatedBook).State = EntityState.Modified;
21
22     try
23     {
24         await _context.SaveChangesAsync();
25     }
26     catch (DbUpdateConcurrencyException)
27     {
28         if (!BookExists(id))
29         {
30             return NotFound();
31         }
32         else
33         {
34             throw;
35         }
36     }
37
38     return NoContent();
39 }
40
```

Figure 5: Code Snippet of Book Controller

## ➤ Auth Controller

The AuthController manages user authentication, including registration and login functionality.

### Key Endpoints:

- POST /api/Auth/Register - Registers a new user.
  - Validates that Password matches ConfirmPassword.
  - Hashes the password before storing it securely in the database.
- POST /api/Auth/Login - Authenticates a user.
  - Verifies the username and password using the database.

```
1 // Register Endpoint
2 [HttpPost("Register")]
3 public async Task<IActionResult> Register([FromBody] User user)
4 {
5     if (user.Password != user.ConfirmPassword)
6     {
7         return BadRequest("Password and Confirm Password must match.");
8     }
9
10    // Hash the password
11    user.Password = BCrypt.Net.BCrypt.HashPassword(user.Password);
12
13    _context.Users.Add(user);
14    await _context.SaveChangesAsync();
15
16    return Ok("Registration successful.");
17 }
18
19 // Login Endpoint
20 [HttpPost("Login")]
21 public async Task<IActionResult> Login([FromBody] LoginRequestDto loginRequest)
22 {
23     if (string.IsNullOrEmpty(loginRequest.Username) || string.IsNullOrEmpty(loginRequest.Password))
24     {
25         return BadRequest("Username and Password are required.");
26     }
27
28     // Validate the user from the database
29     var user = await _context.Users.FirstOrDefaultAsync(u => u.Username == loginRequest.Username);
30
31     if (user == null)
32     {
33         return Unauthorized("Invalid username.");
34     }
35
36     // Verify the hashed password
37     if (!BCrypt.Net.BCrypt.Verify(loginRequest.Password, user.Password))
38     {
39         return Unauthorized("Invalid password.");
40     }
41
42     // Return username on successful login
43     return Ok(new { message = "Login successful", username = user.Username });
44 }
45
46 }
```

Figure 6: Code Snippet of Auth Controller

## Error Handling and Validation Mechanisms

Error handling and validation mechanisms play a crucial role in ensuring that the backend operates smoothly and provides meaningful feedback to the client when issues arise. In the Library Management System, error handling is implemented to capture and manage exceptions while validation mechanisms enforce data integrity and reliability.

### 1. Error Handling

Error handling is implemented in the backend to manage potential issues during operations such as database queries, user authentication, and CRUD operations for books. By handling errors gracefully, the system ensures stability and provides clear messages to the client.

- Try-Catch Blocks

Used to handle exceptions and ensure the application doesn't crash when errors occur.

**Example:** Handling exceptions during database updates in the BookController.



```
1  // PUT: api/Book/5
2  [HttpPut("{id}")]
3  public async Task<IActionResult> UpdateBook(int id, Book updatedBook)
4  {
5      if (id != updatedBook.Id)
6      {
7          return BadRequest();
8      }
9
10     _context.Entry(updatedBook).State = EntityState.Modified;
11
12     try
13     {
14         await _context.SaveChangesAsync();
15     }
16     catch (DbUpdateConcurrencyException)
17     {
18         if (!BookExists(id))
19         {
20             return NotFound();
21         }
22         else
23         {
24             throw;
25         }
26     }
27
28     return NoContent();
29 }
```

Figure 7: Code Snippet of Try-Catch Blocks

### Explanation:

- The try block attempts to update the book in the database.
  - If a concurrency issue or database exception occurs, it is caught and handled with a detailed response to the client.
- HTTP Response Codes

The backend uses appropriate HTTP status codes to indicate the result of an operation.

Examples:

- **200 OK:** Operation succeeded.
- **400 Bad Request:** Invalid data input.
- **404 Not Found:** Requested resource does not exist.
- **500 Internal Server Error:** Unhandled server-side exceptions.

## 2. Validation Mechanisms

Validation ensures that only valid and meaningful data is processed by the backend. This is achieved through annotations and custom validation logic.

- Model Validation with Data Annotations

ASP.NET Core uses data annotations on models to enforce constraints.



```
1 namespace LibraryManagementSystemBackend.Models
2 {
3     public class User
4     {
5         [Key]
6         public int Id { get; set; }
7
8         [Required]
9         [MaxLength(50)]
10        public string Username { get; set; }
11
12        [Required]
13        [MaxLength(50)]
14        public string Password { get; set; }
15
16        [NotMapped] // Ensure this is included
17        [Compare("Password", ErrorMessage = "Passwords do not match.")]
18        public string ConfirmPassword { get; set; }
19    }
20 }
21
```

Figure 8: Code Snippet for Model Validation with Data Annotations

### Explanation:

- [Required]: Ensures the Username and Password fields are not empty.
  - [MaxLength(50)]: Limits the length of Username and Password to prevent overflow.
  - [Compare]: Validates that Password and ConfirmPassword match.
- Custom Validation Logic

Implemented in controllers to handle specific business rules, such as checking if a username already exists during registration.



Figure 9: Code Snippet for Custom Validation Logic

### Challenges Faced During Backend Development

The backend development process for the Library Management System involved overcoming various challenges to ensure the application's robustness and usability. Among the most significant challenges was implementing **user authentication**, which required careful planning, debugging, and the integration of secure practices. Here's a detailed account of the challenges faced and how they were resolved.

#### ➤ User Authentication Challenges

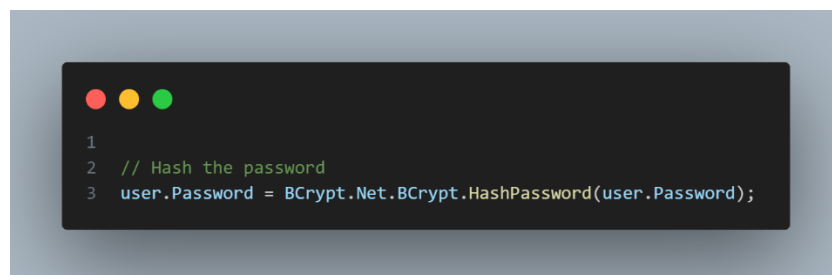
Implementing a secure and user-friendly authentication system posed multiple challenges. These challenges spanned from securely storing user credentials to verifying login details and handling errors effectively.

## Challenges:

1. Password Security:
  - Storing plain text passwords was not an option due to security concerns.
  - Ensuring that passwords were hashed and stored securely while still being verifiable during login was a complex task.
2. Validating Login Credentials:
  - Validating the user-provided credentials against the stored hashed password required proper understanding and integration of hashing algorithms.
  - Ensuring that unauthorized access was correctly blocked when invalid credentials were provided.
3. Meaningful Error Messages:
  - During login and registration, users needed clear feedback to understand why an action failed (e.g., incorrect password, username already exists).
4. Password Matching During Registration:
  - Ensuring that the Password and ConfirmPassword fields matched before allowing the registration process to proceed.
5. Consistency and Debugging:
  - Debugging issues with bcrypt during password verification required a good understanding of how the hashing and verification process worked.

## Solutions Implemented:

1. Password Hashing:
  - The **bcrypt** library was used to hash passwords securely before storing them in the database.
  - During registration, passwords were hashed as follows:

A screenshot of a code editor window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) in the top-left corner. The code is as follows:

```
1
2 // Hash the password
3 user.Password = BCrypt.Net.BCrypt.HashPassword(user.Password);
```

*Figure 10: Hash the password*

- During login, the hashed password was verified against the stored hash:

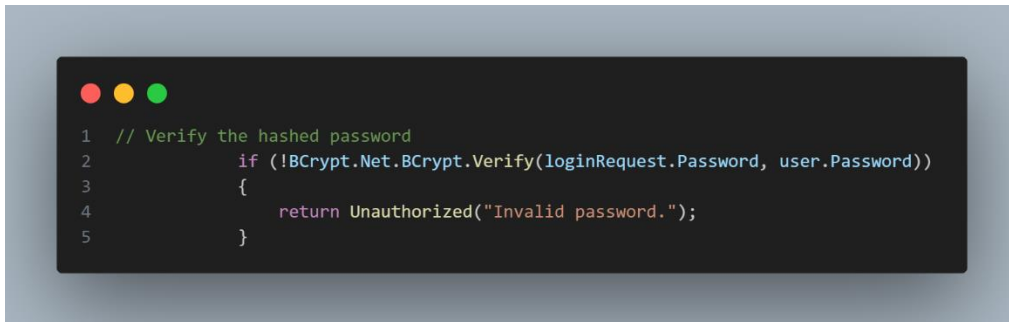


Figure 11: Verify the Hashed Password

## 2. Custom Validation for Registration:

- Additional checks were added to ensure the username is valid.



Figure 12: Custom Validation

## 3. Clear Error Messaging:

- User feedback was implemented to ensure clarity. For example:
  - “Username and Password are required” if fields were empty.
  - “Passwords do not match” if Password and ConfirmPassword did not align.

## 4. Debugging bcrypt Issues:

- Testing with different password lengths and hashes in local development helped resolve verification inconsistencies.
- A consistent salt factor for hashing was ensured to avoid mismatched results.

## 2.2. Frontend Implementation

### Overview of the Frontend Architecture

The frontend of the Library Management System is built on a structured and modular component-based architecture. Each feature of the application is encapsulated into its respective component, ensuring separation of concerns and easier maintenance. The architecture supports a **Single Page Application (SPA)**, allowing smooth navigation between pages without full page reloads. This improves user experience by reducing latency and ensuring a seamless flow.

Key aspects of the architecture include:

1. **Component-based Design**

The application is divided into reusable components such as Login, Register, Dashboard, ViewBooks, and AddBook. Each component manages its own logic, UI, and state independently while interacting cohesively with the rest of the application.

2. **Routing and Navigation**

The frontend incorporates navigation with React Router, which maps specific URLs to their respective components (/login, /register, /dashboard, /view-books, /add-book). This enables intuitive navigation between pages, essential for an SPA structure.

3. **State Management**

State within the application is managed using React's useState and useEffect hooks. For example:

- Login and registration pages manage user input and error messages locally.
- The ViewBooks page fetches and manages the list of books dynamically.

4. **Responsive Design**

The frontend incorporates responsive design principles to ensure the application is accessible across a variety of devices, including desktops, tablets, and mobile phones. The layout adapts dynamically to different screen sizes, providing an optimal user experience.

5. **Integration with Backend**

The frontend communicates with the backend through REST API endpoints using Axios. This ensures data synchronization for CRUD operations and user authentication.



## Technologies Used For Frontend Development

The frontend of the Library Management System is designed using modern technologies and tools to deliver a responsive, user-friendly, and visually appealing interface. Below are the primary technologies used:

### 1. Programming Language

- **JavaScript with TypeScript:** The frontend is built using React with TypeScript, which adds static typing to JavaScript for better maintainability and error prevention. TypeScript helps in defining types and interfaces for components and props, ensuring a robust development process.

### 2. Library and Framework

- **React.js:** A powerful JavaScript library for building dynamic and reusable user interfaces. React enables the development of component-based architecture, making the code modular, maintainable, and scalable. Features like React Hooks (useState, useEffect) are extensively used for state management and lifecycle handling.

### 3. HTTP Client

- **Axios:** A promise-based HTTP client used for making API requests. It simplifies communication with the backend, including fetching and sending data. Axios is used in components like ViewBooks to retrieve books and AddBook to send book data to the backend.

### 4. Router

- **React Router:** Used for managing navigation within the application. It allows dynamic routing without page reloads, improving the user experience. The useNavigate hook is utilized for programmatic redirection, such as navigating to the dashboard after login or registration.

### 5. UI and Styling

- **CSS:** Cascading Style Sheets (CSS) are used for designing the layout and visuals of the application. Each component has its own scoped CSS file (Login.css, Dashboard.css) to maintain separation of concerns.
- **Responsive Design:** Media queries are applied to ensure responsiveness across devices, including mobile, tablet, and desktop views.

## Implementation of CRUD Operations

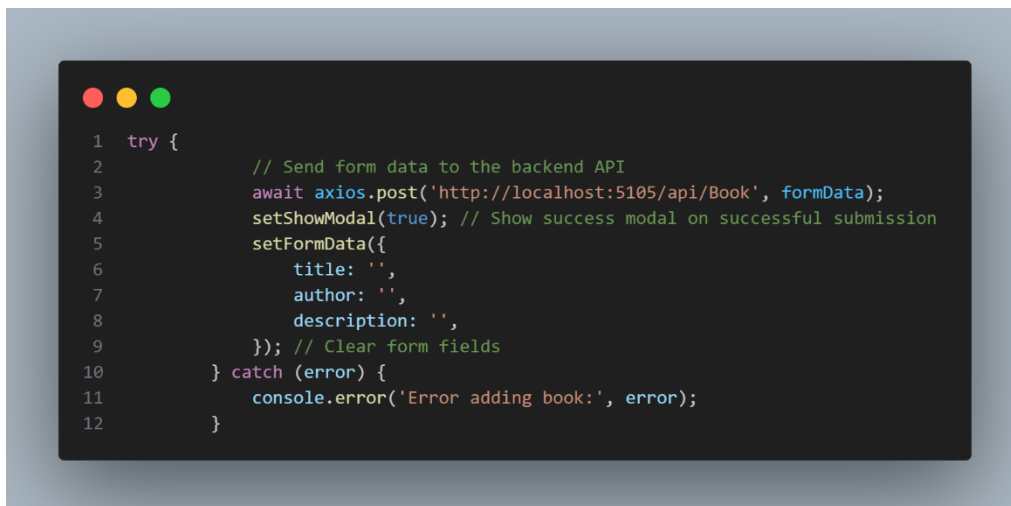
The frontend of the Library Management System includes Create, Read, Update, and Delete (CRUD) operations to manage books effectively. These operations are implemented using React.js components and interact with the backend through HTTP requests via Axios. Below is the detailed explanation of how each operation is implemented with corresponding code snippets for the report.

### 1. Create (Add a New Book)

The "Add Book" functionality allows users to add a new book to the system. The AddBook component includes a form for inputting book details, such as title, author, and description. Upon form submission, the data is sent to the backend API.

Key Features:

- Form validation ensures that required fields are not empty.
- A success modal is displayed after successfully adding a book.



```
1  try {
2      // Send form data to the backend API
3      await axios.post('http://localhost:5105/api/Book', formData);
4      setShowModal(true); // Show success modal on successful submission
5      setFormData({
6          title: '',
7          author: '',
8          description: '',
9      }); // Clear form fields
10 } catch (error) {
11     console.error('Error adding book:', error);
12 }
```


Figure 13: Code Snippet for Add New Book

### 2. Read (View All Books and Search for a Book)

The ViewBooks component displays all books fetched from the backend and allows users to search for a specific book by its ID. The fetched data is displayed in a card layout for better readability.

Key Features:

- Book details such as ID, title, author, and description are displayed.
- A search bar is provided for filtering books by ID.



```

1 // Function to fetch all books from the API
2 const fetchBooks = async () => {
3   try {
4     const response = await axios.get("http://localhost:5105/api/Book");
5     setBooks(response.data); // Update the books state with the fetched data
6     setSearchResult(null); // Clear any existing search results
7   } catch (error) {
8     console.error("Error fetching books:", error);
9   }
10 };
11
12 // Function to handle book search by ID
13 const handleSearch = async () => {
14   if (!searchTerm.trim()) {
15     setSearchResult(null); // Clear search results
16     setHasSearched(false); // Reset the search flag
17     return;
18   }
19
20   setHasSearched(true); // Mark that a search attempt was made
21
22   try {
23     const response = await axios.get(
24       `http://localhost:5105/api/Book/${searchTerm}`
25     );
26     setSearchResult(response.data); // Update the search result state
27   } catch (error) {
28     setSearchResult(null); // No results found
29     console.error("Error searching for book:", error);
30   }
31 };

```

Figure 14: Code Snippet For View All Books and Search for a Book

### 3. Update (Modify an Existing Book)

The update functionality is implemented through a modal that allows users to modify the details of a selected book. Once the user submits the updated details, the changes are sent to the backend API.

Key Features:

- Editable fields for title, author, and description.
- Updates the book details in the database and frontend.

```

1 // Function to handle book update
2 const handleUpdate = async () => {
3   if (selectedBook) {
4     try {
5       await axios.put(
6         `http://localhost:5105/api/Book/${selectedBook.id}`,
7         selectedBook
8       );
9       // Update the book list with the updated book details
10      setBooks((prevBooks) =>
11        prevBooks.map((book) =>
12          book.id === selectedBook.id ? selectedBook : book
13        )
14      );
15      setShowUpdateModal(false); // Hide update modal
16      setShowUpdateSuccess(true); // Show update success modal
17    } catch (error) {
18      console.error("Error updating book:", error);
19    }
20  }
21 };
22
23 // Function to open the update modal
24 const openUpdateModal = (book: Book) => {
25   setSelectedBook(book); // Set the selected book for updating
26   setShowUpdateModal(true); // Show update modal
27 };

```

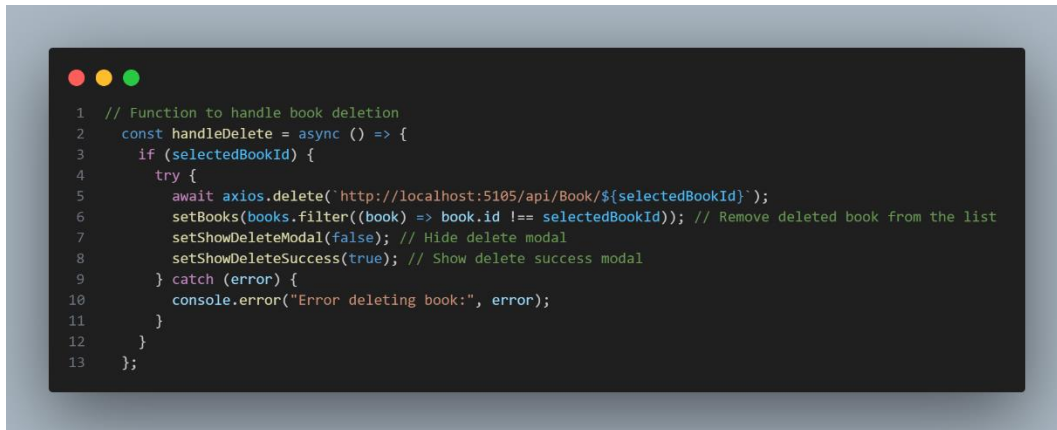
Figure 15: Code Snippet for Modify an Existing Book

#### 4. Delete (Remove a Book)

The delete functionality allows users to remove a book from the system. A confirmation modal is displayed before the deletion to prevent accidental removal.

Key Features:

- Confirmation modal ensures user validation before deletion.
- Removes the book from both the backend and the frontend.

A code editor window with a dark background and light-colored text. The code is a JavaScript function named `handleDelete` that is asynchronous. It checks if `selectedBookId` is provided. If it is, it uses `axios.delete` to remove the book from the API. After a successful deletion, it filters the `books` array to remove the deleted book, updates the state, hides the delete modal, and shows a success modal. It also includes a `catch` block for error handling.

```
1 // Function to handle book deletion
2 const handleDelete = async () => {
3   if (selectedBookId) {
4     try {
5       await axios.delete(`http://localhost:5105/api/Book/${selectedBookId}`);
6       setBooks(books.filter((book) => book.id !== selectedBookId)); // Remove deleted book from the list
7       setShowDeleteModal(false); // Hide delete modal
8       setShowDeleteSuccess(true); // Show delete success modal
9     } catch (error) {
10      console.error("Error deleting book:", error);
11    }
12  }
13 };
```

Figure 16: Code Snippet for Remove a Book

## State Management and Routing Strategies

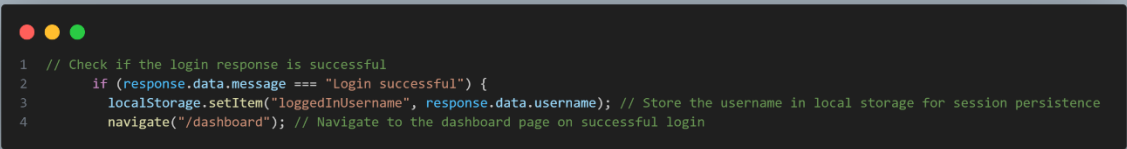
Effective state management and routing are critical to creating a seamless user experience in a single-page application like the Library Management System. Below is an explanation of how state management and routing are implemented in this project.

### 1. State Management

State management is handled using React's built-in **state hooks** (`useState`) and **effect hooks** (`useEffect`). These hooks ensure that the application's data and UI remain synchronized as the user interacts with the app.

Key Features of State Management:

1. Dynamic UI Updates:
  - State variables like `books`, `searchTerm`, and `formData` dynamically update the UI whenever their values change.
  - Example: When a user searches for a book, the `searchResult` state is updated and displayed immediately.
2. Global State Persistence:
  - Local storage is used for persisting the logged-in user's session.
  - Example: The `loggedInUsername` is stored in local storage to maintain session persistence across page reloads.



```
1 // Check if the login response is successful
2 if (response.data.message === "Login successful") {
3   localStorage.setItem("loggedInUsername", response.data.username); // Store the username in local storage for session persistence
4   navigate("/dashboard"); // Navigate to the dashboard page on successful login
```

Figure 18: Code Snippet for Store Logged in Username



```
1 // Function to fetch all books from the API
2 const fetchBooks = async () => {
3   try {
4     const response = await axios.get("http://localhost:5105/api/Book");
5     setBooks(response.data); // Update the books state with the fetched data
6     setSearchResult(null); // Clear any existing search results
7   } catch (error) {
8     console.error("Error fetching books:", error);
9   }
10  };

```

Figure 17: Code Snippet for Display Searched Book

## 2. Routing Strategies

Routing is implemented using **React Router**, enabling seamless navigation between different views such as login, registration, dashboard, viewing books, adding books, and editing books.

Key Features of Routing:

1. Dynamic Navigation:
  - React Router's `useNavigate` hook is used for programmatic navigation, such as redirecting to the dashboard after a successful login.
2. Protected Routes:
  - Some routes (e.g., `/dashboard`, `/view-books`, `/add-book`) are protected by checking if the user is logged in. If not, the user is redirected to the login page.
3. Breadcrumb-Like Navigation:
  - Intuitive links are provided to switch between pages like "View Books" and "Add Books" from the sidebar.

## Validation and Error Handling on the Frontend

Input validation and error handling are critical aspects of the frontend implementation to ensure data integrity, user-friendly feedback, and robust application behavior. Below is an explanation of how input validation and error handling are implemented in the Library Management System.

### 1. Input Validation

Input validation ensures that the data entered by users is valid and meets the required format before it is sent to the backend. The application uses JavaScript checks and state-based validation mechanisms for form fields.

Key Features of Input Validation:

#### 1. Field-Specific Validation:

- Each form field (e.g., username, password, book title) has specific validation rules to check for missing or invalid data.
- Example: Ensuring the username and password fields are not empty during login or that the "Confirm Password" matches the "Password" during registration.

#### 2. Dynamic Feedback:

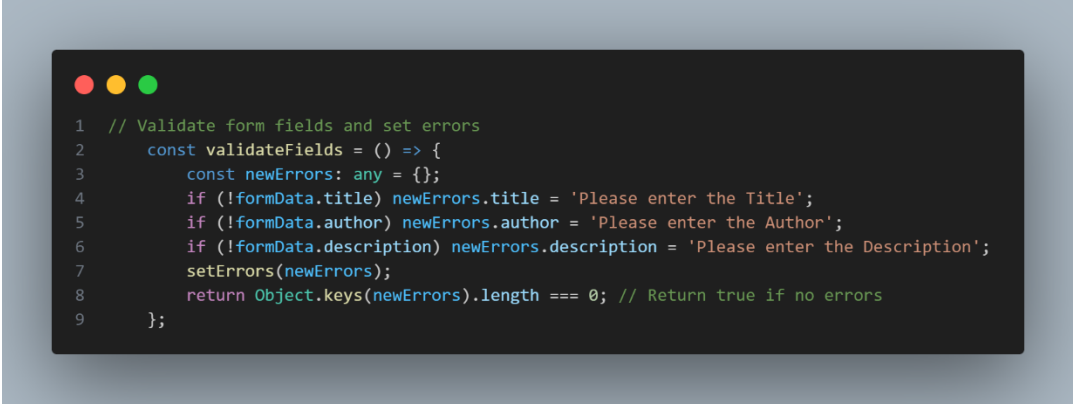
- Real-time feedback is provided to users by displaying error messages next to invalid fields.

#### 3. Reset Mechanism:

- Input fields and error messages are cleared when the user clicks the "Reset" button in forms.



Figure 19: Registration Password Validation



```
1 // Validate form fields and set errors
2 const validateFields = () => {
3   const newErrors: any = {};
4   if (!formData.title) newErrors.title = 'Please enter the Title';
5   if (!formData.author) newErrors.author = 'Please enter the Author';
6   if (!formData.description) newErrors.description = 'Please enter the Description';
7   setErrors(newErrors);
8   return Object.keys(newErrors).length === 0; // Return true if no errors
9 };
```

*Figure 20: Code Snippet for Add Book Validation*

## 2. Error Handling

Error handling is implemented to manage unexpected situations gracefully, such as invalid API responses or connection issues.

Key Features of Error Handling:

1. API Response Validation:
  - The application checks the API response status or message to determine if an operation was successful or if an error occurred.
  - Example: If a book search fails, a "No Results Found" message is displayed to the user.
2. Global Error Handling:
  - Errors from API requests are caught using try-catch blocks, and meaningful messages are displayed to users.
3. Fallback Mechanism:
  - Fallback messages (e.g., "An error occurred. Please try again.") are shown when specific error details are unavailable.



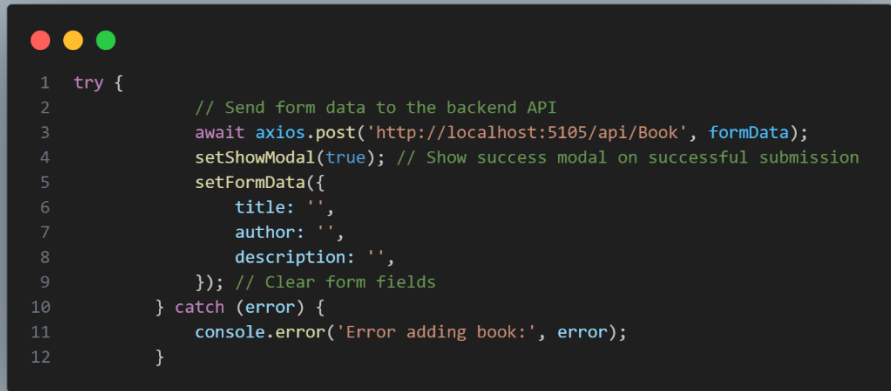


```

1  try {
2      const response = await axios.get(
3          `http://localhost:5105/api/Book/${searchTerm}`
4      );
5      setSearchResult(response.data); // Update the search result state
6  } catch (error) {
7      setSearchResult(null); // No results found
8      console.error("Error searching for book:", error);
9  }

```

Figure 23: Code Snippet for Search Error Handling

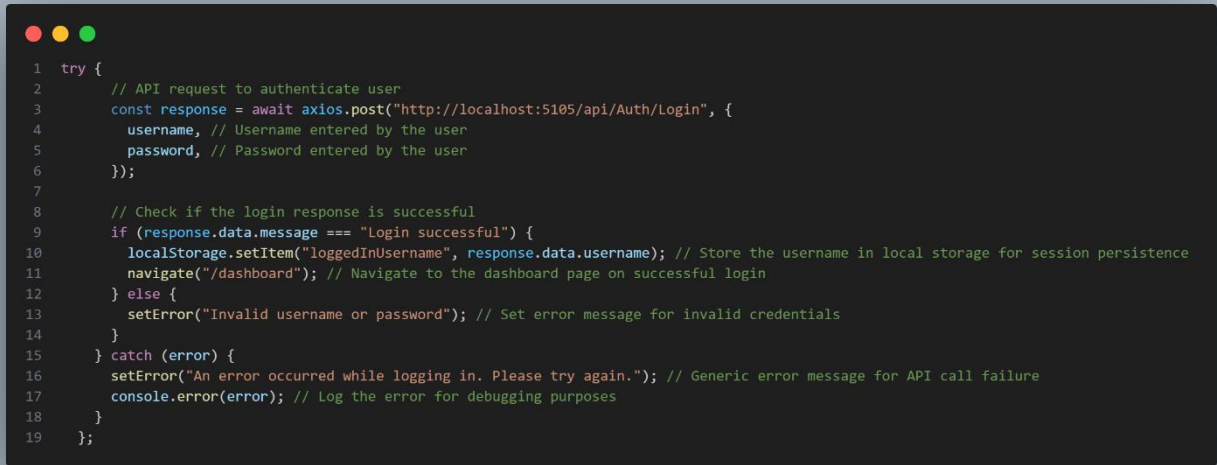


```

1  try {
2      // Send form data to the backend API
3      await axios.post('http://localhost:5105/api/Book', formData);
4      setShowModal(true); // Show success modal on successful submission
5      setFormData({
6          title: '',
7          author: '',
8          description: '',
9      }); // Clear form fields
10 } catch (error) {
11     console.error('Error adding book:', error);
12 }

```

Figure 22: Code Snippet for Add Book Error Handling



```

1  try {
2      // API request to authenticate user
3      const response = await axios.post("http://localhost:5105/api/Auth/Login", {
4          username, // Username entered by the user
5          password, // Password entered by the user
6      });
7
8      // Check if the login response is successful
9      if (response.data.message === "Login successful") {
10         localStorage.setItem("loggedInUsername", response.data.username); // Store the username in local storage for session persistence
11         navigate("/dashboard"); // Navigate to the dashboard page on successful login
12     } else {
13         setError("Invalid username or password"); // Set error message for invalid credentials
14     }
15 } catch (error) {
16     setError("An error occurred while logging in. Please try again."); // Generic error message for API call failure
17     console.error(error); // Log the error for debugging purposes
18 }
19 );

```

Figure 21: Code Snippet for Login Error Handling

## Challenges Faced During Frontend Development

Developing the frontend for the Library Management System posed several challenges that required thoughtful solutions to ensure a seamless user experience. Below are the major challenges encountered and how they were addressed:

### 1. Integrating State Management Across Components

- **Challenge:** Managing state effectively for components like ViewBooks, AddBook, and Login was challenging, especially when dealing with shared data or propagating state updates (e.g., after adding or deleting a book).
- **Solution:**
  - Used React's **useState** and **useEffect** hooks to manage local component state and handle updates dynamically.
  - Component-specific state variables were created (e.g., books, selectedBook, searchResult) to maintain isolation and prevent unintended side effects.
  - Leveraged controlled components for form inputs to ensure the state was synchronized with the UI.
  - Where global state was not required, state management libraries like Redux were avoided to reduce complexity.

### 2. Implementing Responsive Design

- **Challenge:** Making the application responsive and ensuring proper rendering across different screen sizes (mobile, tablet, and desktop) was challenging due to the complex layouts of the sidebar, modals, and content areas.
- **Solution:**
  - Utilized CSS media queries to adjust layout properties (e.g., sidebar width, font sizes, modal positions) for various screen sizes.
  - Ensured proper alignment and spacing using flexbox and grid layouts.
  - Tested extensively on browser dev tools and emulators to ensure compatibility across different resolutions.

### 3. Dynamic Error Handling

- **Challenge:** Handling dynamic errors (e.g., API errors, input validation errors) in a user-friendly way was challenging. For instance, showing relevant messages when a book search failed or when a login attempt was invalid.
- **Solution:**
  - Implemented error handling using try-catch blocks for API calls to capture errors and provide meaningful feedback to users.
  - Displayed contextual error messages for each form field during validation, ensuring users could quickly correct their input.

- Created a consistent UI for error messages and success modals using reusable CSS classes.

#### 4. **Synchronizing Frontend and Backend**

- **Challenge:** Synchronizing the frontend with the backend API endpoints was challenging, especially when dealing with unexpected response formats or server-side errors.
- **Solution:**
  - Used tools like Swagger to test and document backend APIs.
  - Debugged issues by logging API responses and using browser developer tools.
  - Implemented fallback error messages for unhandled scenarios to ensure the application did not crash.

### 3. User Interface Designs

For the Library Management System, the UI was designed with a focus on simplicity, responsiveness, and functionality, ensuring a seamless user experience across all devices. Each page in the application serves a specific purpose and has been structured to enhance usability.

#### 1. Register Page

The Register Page is designed to allow new users to create accounts effortlessly. It features a simple form where users can input their username, password, and confirm their password. Input validation ensures that passwords match, and any errors are displayed clearly. A success message confirms the registration process, while a link redirects users to the login page if they already have an account.

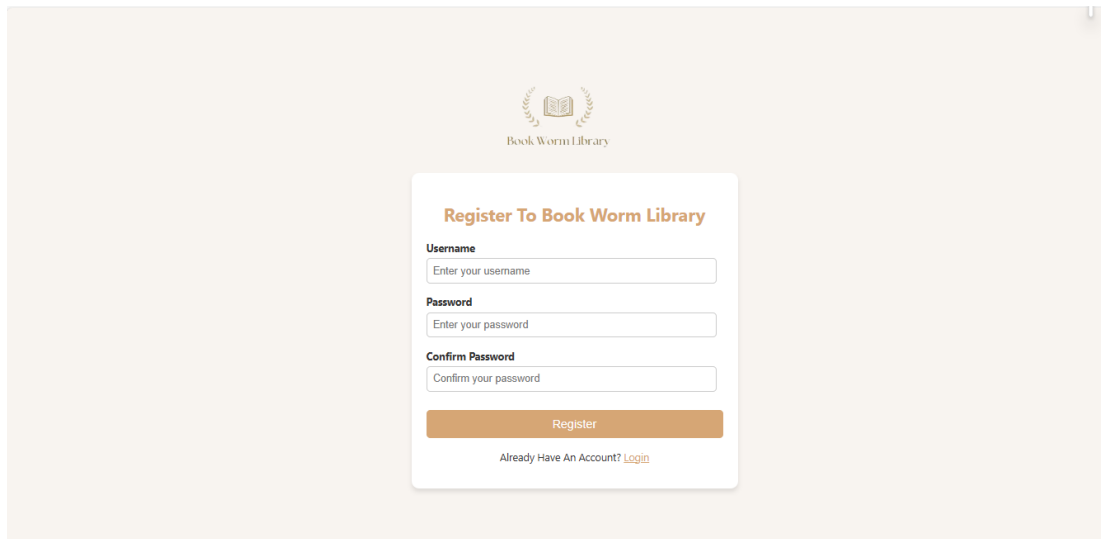
The image shows a registration form for 'Book Worm Library'. At the top center is a logo featuring an open book with a worm inside, flanked by laurel leaves, with the text 'Book Worm Library' below it. The form itself is a white card with a light orange border. It has a title 'Register To Book Worm Library' in orange. Below the title are three input fields: 'Username' with placeholder text 'Enter your username', 'Password' with placeholder text 'Enter your password', and 'Confirm Password' with placeholder text 'Confirm your password'. Each field has a small orange border. Below the fields is a large orange 'Register' button. At the bottom of the card, it says 'Already Have An Account? [Login](#)'.

Figure 24: User Interface for Register Page

## 2. Login Page

The Login Page serves as the entry point to the system, enabling users to log in securely. The page consists of fields for entering the username and password, with error messages displayed in case of invalid credentials. A link to the register page is provided for new users, ensuring an easy and guided navigation process. The welcoming design includes a logo and a friendly heading to enhance user engagement.

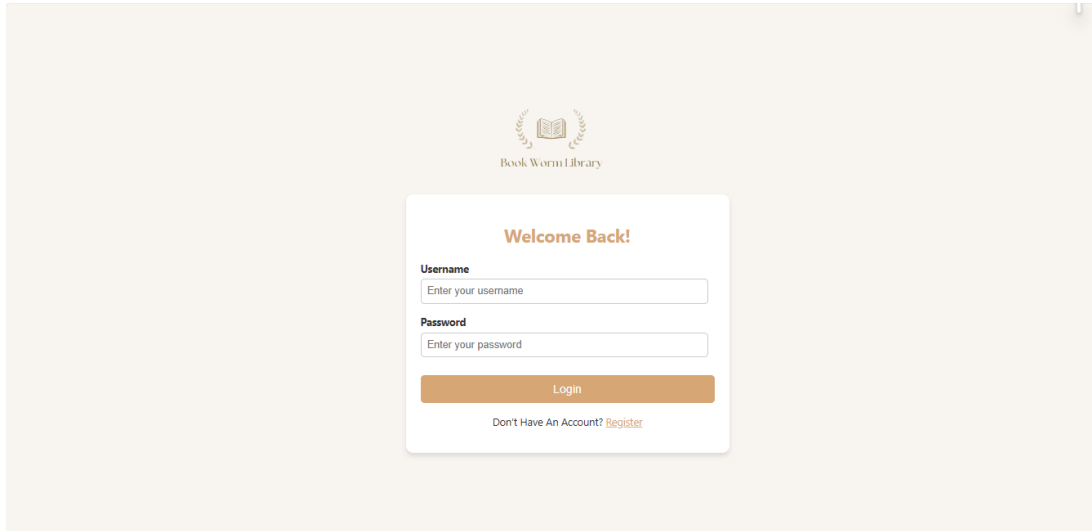


Figure 25: User Interface for Login Page

## 3. Dashboard Page

The Dashboard Page acts as the control center for the application. Upon successful login, users are greeted with a welcome message displaying their name. The dashboard provides an overview of the system, with quick navigation options to view or add books.

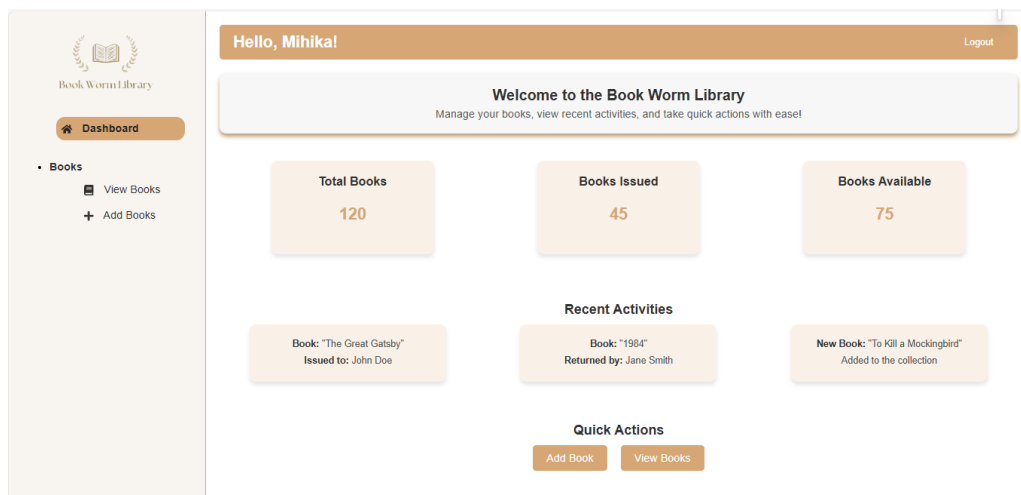


Figure 26: User Interface for Dashboard Page

## 4. View Book Page

The View Book Page allows users to browse, search, update, or delete books in the library. The page features a search bar for finding specific books by ID and a card-based layout displaying book details like ID, title, author, and description. Each book card includes options to update or delete, with modals ensuring smooth confirmation or editing processes.

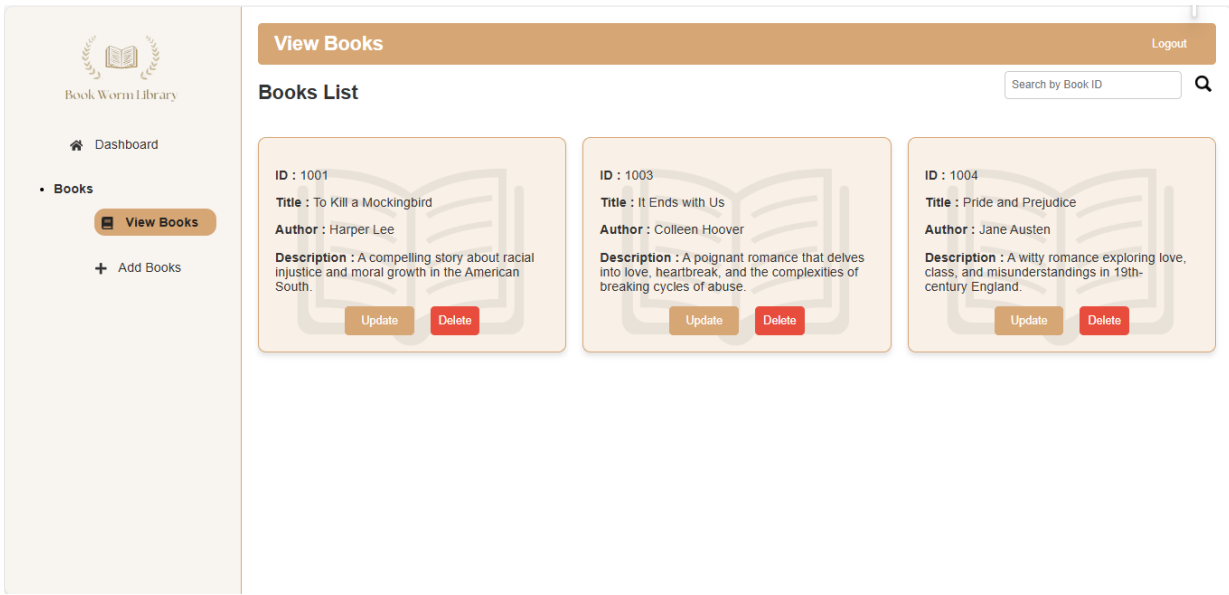
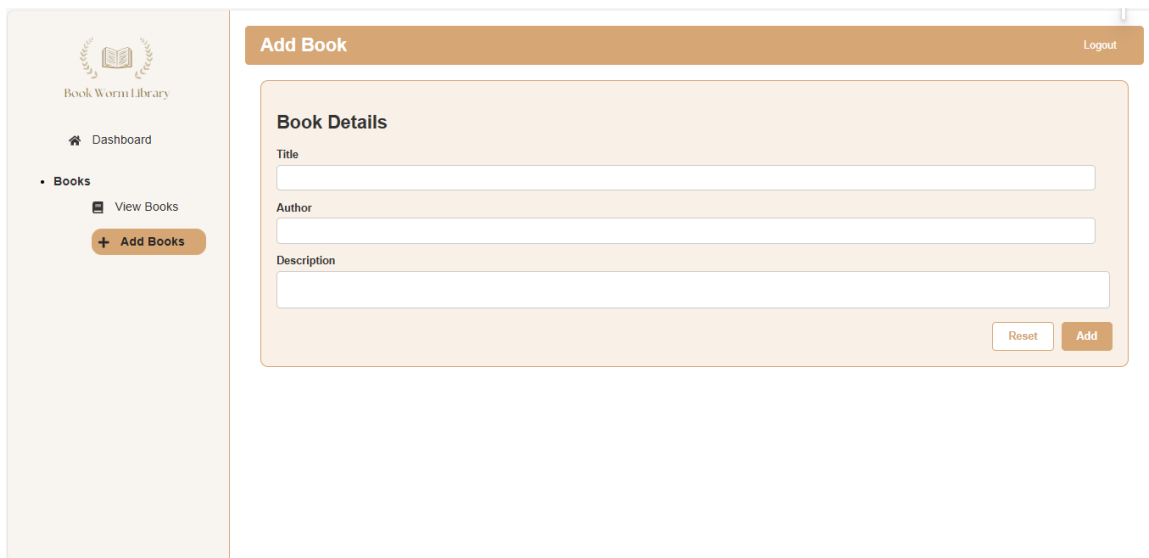



Figure 27: User Interface for View Book Page

## 5. Add Book Page

The Add Book Page is tailored for adding new books to the library. A clean and straightforward form allows users to input the title, author, and description of a book, with validation ensuring no field is left empty. Upon successful addition, a confirmation modal appears. The page also includes options to reset the form or navigate back to the dashboard or other sections, maintaining consistency and ease of use.



The image shows a web application interface for adding a new book. On the left is a sidebar with the 'Book Worm Library' logo and navigation links for 'Dashboard', 'Books' (with sub-links 'View Books' and '+ Add Books'), and a 'Logout' link. The main content area is titled 'Add Book' and contains a 'Book Details' form. The form has three input fields: 'Title', 'Author', and 'Description'. At the bottom right of the form are 'Reset' and 'Add' buttons.

  
Book Worm Library

[Dashboard](#)

• Books

[View Books](#)

[+ Add Books](#)

[Logout](#)

Add Book

Book Details

Title

Author

Description

Reset

Add

Figure 28:User Interface for Add Book Page

## 4. Additional Features Developed

### ➤ Search Feature

One of the standout additional features implemented in the Library Management System is the **search functionality**, which allows users to efficiently locate books using their unique IDs. This feature significantly enhances the usability and convenience of the system, particularly in scenarios where a large number of books are stored.

### Implementation of the Search Feature

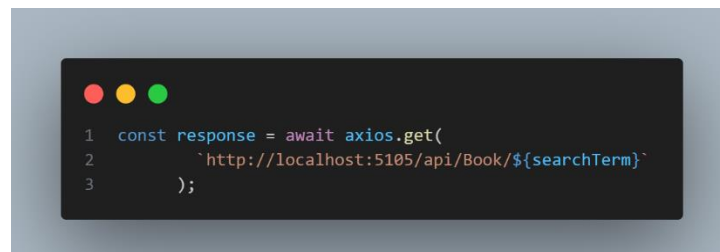
The search functionality is integrated into the ViewBooks component, enabling users to input a **Book ID** and fetch the corresponding book details. Below is an explanation of how this feature works:

#### 1. Search Input Field:

- A dedicated input field is provided in the ViewBooks component where users can enter the Book ID.
- The entered value is captured and stored in a searchTerm state variable using the useState hook.

#### 2. API Integration:

- On clicking the search button, the application sends a **GET request** to the backend API endpoint:



```
1  const response = await axios.get(  
2    `http://localhost:5105/api/Book/${searchTerm}`  
3  );
```

Figure 29: API Endpoint for Search Functionality

- This fetches the book details corresponding to the entered Book ID.

#### 3. Error Handling:

- If a book with the provided ID does not exist, an error message, such as "No Results Found," is displayed to inform the user.

#### 4. Conditional Rendering:

- If a valid Book ID is provided, the details of the searched book are displayed dynamically in a visually distinct card format.
- If the search input is empty or invalid, all books or an error message is shown, respectively.



## **Benefits of the Search Feature**

### **1. Enhanced User Experience:**

- Instead of scrolling through the entire list of books, users can quickly locate a book by its ID.
- Reduces the time required to perform specific operations like updates or deletions.

### **2. Scalability:**

- As the library grows with more books, manually locating a book becomes increasingly tedious. The search feature ensures the application remains scalable and user-friendly.

### **3. Error-Handling Mechanism:**

- By providing meaningful feedback, such as "No Results Found," the system ensures a smooth user experience, even when incorrect inputs are provided.

### **4. Real-Time Feedback:**

- Users can instantly view the results or receive relevant error messages based on their input, enhancing responsiveness.

### **5. Consistency:**

- The search results are displayed in a card layout similar to the book list, ensuring consistent design across the application.

## 5. Key Insights and Learnings

The development of the Library Management System was an enriching experience, offering both **technical** and **personal** growth. This section outlines the major insights and learnings gained during the course of the project.

### ➤ Technical Learnings

#### 1. Full-Stack Development:

- Building a project that integrates both frontend and backend components provided a holistic understanding of full-stack development.
- Developing RESTful APIs on the backend and consuming them on the frontend offered practical experience in seamless client-server communication.

#### 2. State Management in React:

- Managing application state effectively was crucial for creating a dynamic and responsive user experience.
- Handling multiple states in components like ViewBooks (e.g., book list, search results, modals) deepened my understanding of React's `useState` and `useEffect` hooks.

#### 3. Backend Development with ASP.NET Core:

- Working with ASP.NET Core taught me how to:
  - Set up a scalable and efficient backend.
  - Use Entity Framework Core for database operations and seamless data management.
  - Implement authentication with password hashing using BCrypt.Net.

#### 4. Database Integration:

- Designing and managing a relational database using SQLite allowed me to understand the principles of schema design and data modeling.
- Working with DbContext and performing CRUD operations using Entity Framework improved my proficiency with database interaction in C#.

### ➤ Personal Learnings

#### 1. Time Management:

- Balancing the various aspects of the project, including backend implementation, frontend design, testing, and documentation, taught me the importance of time allocation and prioritization.

#### 2. Problem-Solving:

- Debugging issues such as user authentication errors and integrating the search functionality required critical thinking and perseverance.
- Researching solutions and testing different approaches strengthened my problem-solving capabilities.

3. **Adaptability:**

- Encountering challenges like implementing password hashing or designing a scalable frontend required me to adapt and learn new technologies quickly.

## 6. Conclusion

The Library Management System project successfully achieved its objectives, providing a reliable and user-friendly platform for managing library operations. From user registration and login to book management functionalities such as adding, viewing, updating, and deleting records, the system was designed to streamline tasks efficiently.

This project provided valuable insights into software development. Technically, it strengthened understanding of backend technologies like ASP.NET Core, frontend frameworks like React, and key concepts such as state management, routing, and error handling. Personally, it enhanced problem-solving skills and adaptability, especially when addressing challenges like user authentication and responsive design.

Overall, this project was a fulfilling experience, blending technical learning with practical application. The resulting system is both functional and scalable, laying the foundation for future improvements and additional features.