

Software Engineering Internship – Assignment

# **Library Management System**

Hirushan Supun Keerthiratne

21<sup>th</sup> November 2024

## Contents

1. Introduction.....	4
1.1. Purpose and Objectives.....	4
1.2. Technological Overview.....	4
1.3. Project Scope .....	5
1.4. Project Structure.....	5
2. Backend Development.....	6
2.1. Technologies and Frameworks .....	6
2.2. Database Context .....	7
2.3. Book Entity .....	8
2.4. User Entity .....	9
2.5. CRUD Operations for Book Records .....	10
2.6. Authentication and Authorization.....	13
2.7. Middleware Configuration.....	15
3. Frontend Development.....	16
3.1. Technologies and Tools .....	16
3.2. Component Structure .....	17
3.3. Routing.....	19
3.4. State Management.....	20
3.5. CRUD Operations.....	21
3.6. Error Handling .....	23
3.7. Styling.....	24
4. Challenges and Solutions.....	25
4.1. Managing Authentication State Across Components .....	25
5. Ensuring Secure Communication Between Frontend and Backend .....	25
6. Responsive Design.....	26
7. Error Handling for User Input and API Failures .....	26
8. Securing Protected Routes .....	27
9. Debugging API Calls and Network Issues.....	27
10. Validation of Backend Responses .....	28
11. Key Learnings.....	32
11.1. Technical Skills.....	32

11.2.	Problem-Solving .....	32
11.3.	Project Management .....	33
11.4.	Personal Growth.....	33
12.	Conclusion .....	33

# 1. Introduction

The Library Management System is a robust software solution designed to facilitate efficient management of books within a library. This project aims to streamline the processes involved in creating, viewing, updating, and deleting book records, with the added functionality of user authentication and registration for enhanced security and personalized access.

## 1.1.Purpose and Objectives

The primary purpose of this Library Management System is to provide an easy-to-use platform for managing book records while ensuring secure and restricted access through user authentication. The application offers the following core functionalities:

- **Authentication and Registration:** Allows users to create an account, log in securely, and access personalized features.
- **Create** new book records with relevant details such as title, author, and description.
- **Read** and display a list of existing book records in a structured format.
- **Update** the details of existing book records.
- **Delete** book records that are no longer needed.

## 1.2.Technological Overview

The project leverages a combination of modern tools and technologies:

1. **Backend:** Built using **C#** and **.NET Core**, the backend includes:
  - RESTful API endpoints for CRUD operations.
  - SQLite for database management using Entity Framework Core.
  - Authentication and registration features using **ASP.NET Identity** for secure user management.

2. **Frontend:** Developed with **React** and **TypeScript**, the frontend ensures:
  - A responsive and interactive user interface for managing book records.
  - Seamless integration with backend APIs for authentication and CRUD operations.
  - Proper state management, routing, and error handling.

### 1.3. Project Scope

- Implement full CRUD functionality for managing book records.
- Add secure user authentication and registration features.
- Seamlessly integrate backend and frontend for a smooth user experience.
- Follow modern development best practices for scalability and maintainability.

### 1.4. Project Structure

The application consists of two main components:

1. **Backend:**

- Controllers: Handles API endpoints for CRUD operations and authentication.
- Data: Manages database contexts and migrations using Entity Framework Core.
- Models: Defines schemas for book records and user data.
- Services: Implements business logic for authentication and CRUD functionality.

2. **Frontend:**

- components: Contains reusable UI components such as Login, Register, BookList, and EditBook.
- services: Handles API communication for authentication and book operations.
- contexts: Manages user authentication state and application-wide context.

This report will detail the development process, highlighting how authentication and registration were implemented alongside book management features, the challenges faced, and key insights gained during the project.

## 2. Backend Development

The backend of the Library Management System was developed using **C#** and **ASP.NET Core** to provide a robust and secure RESTful API for managing book records and user authentication. It incorporates an SQLite database with Entity Framework Core for efficient data storage and retrieval. The backend supports CRUD operations for books and implements user authentication through JSON Web Tokens (JWT), ensuring secure and controlled access.

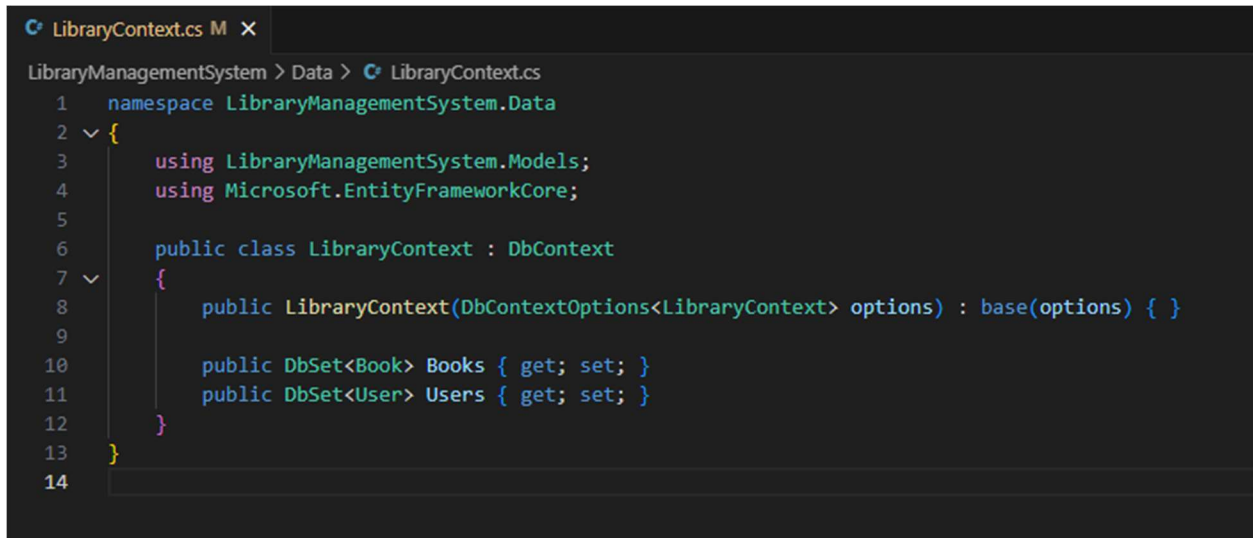
### 2.1. Technologies and Frameworks

The backend leverages the following technologies:

- **ASP.NET Core:** A high-performance framework for building web APIs.
- **Entity Framework Core:** A powerful Object-Relational Mapping (ORM) tool for interacting with the SQLite database.
- **SQLite:** A lightweight and efficient database used to store user and book data.
- **JWT Authentication:** Provides secure and stateless authentication for users.

## 2.2.Database Context

The **LibraryContext** class serves as the central hub for database operations. It defines the structure of the database and manages interactions with the **Books** and **Users** tables.



```
1 namespace LibraryManagementSystem.Data
2 {
3     using LibraryManagementSystem.Models;
4     using Microsoft.EntityFrameworkCore;
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 }
14
```

### Explanation:

This class is configured to use SQLite, as specified in the **appsettings.json** file. The **DbSet<Book>** and **DbSet<User>** properties represent tables in the database, making it easy to perform CRUD operations on these entities.

## 2.3.Book Entity

The Book class defines the schema for the **Books** table in the database. It uses data annotations to enforce validation rules.

```
Book.cs M X
LibraryManagementSystem > Models > Book.cs
1  using System.ComponentModel.DataAnnotations;
2
3  namespace LibraryManagementSystem.Models
4  {
5      public class Book
6      {
7          public int Id { get; set; }
8
9          [Required(ErrorMessage = "Title is required.")]
10         [MaxLength(100, ErrorMessage = "Title cannot exceed 100 characters.")]
11         public string Title { get; set; } = string.Empty;
12
13         [Required(ErrorMessage = "Author is required.")]
14         [MaxLength(100, ErrorMessage = "Author cannot exceed 100 characters.")]
15         public string Author { get; set; } = string.Empty;
16
17         [Required(ErrorMessage = "Description is required.")]
18         [MaxLength(500, ErrorMessage = "Description cannot exceed 500 characters.")]
19         public string Description { get; set; } = string.Empty;
20     }
21 }
22
```

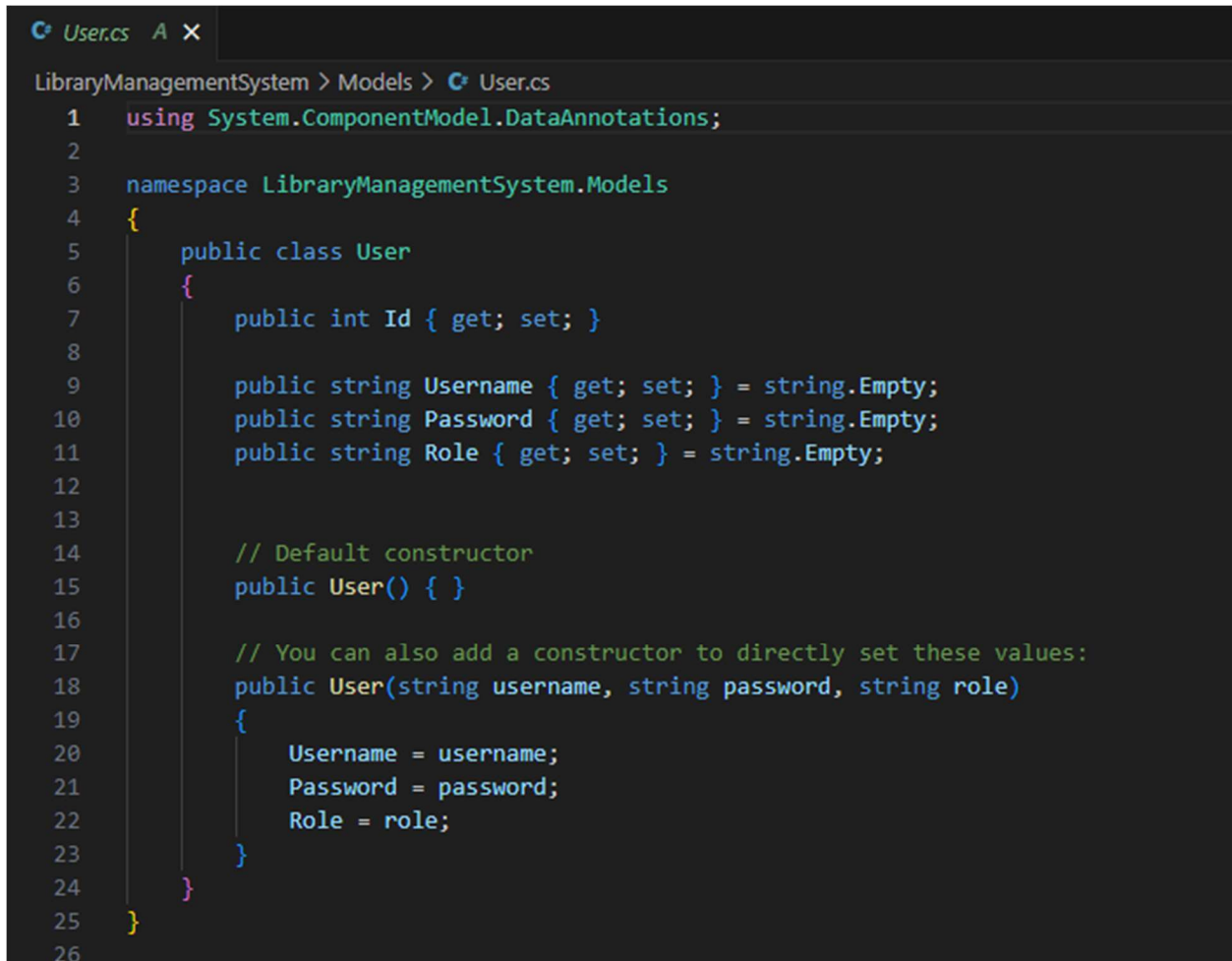
### Explanation:

- The Id property is the primary key.
- The Title, Author, and Description fields are validated to ensure meaningful and properly formatted data is stored in the database.



## 2.4. User Entity

The User class defines the schema for the **Users** table and includes fields for username, password, and role.



```
1  using System.ComponentModel.DataAnnotations;
2
3  namespace LibraryManagementSystem.Models
4  {
5      public class User
6      {
7          public int Id { get; set; }
8
9          public string Username { get; set; } = string.Empty;
10         public string Password { get; set; } = string.Empty;
11         public string Role { get; set; } = string.Empty;
12
13         // Default constructor
14         public User() { }
15
16         // You can also add a constructor to directly set these values:
17         public User(string username, string password, string role)
18         {
19             Username = username;
20             Password = password;
21             Role = role;
22         }
23     }
24 }
25
26
```

### Explanation:

This model supports user authentication by storing hashed passwords and user roles, facilitating role-based access control.

## 2.5.CRUD Operations for Book Records

The BooksController provides RESTful endpoints for managing book records. Each endpoint handles a specific operation: create, read, update, or delete.

```
// GET: api/Books
[HttpGet]
public async Task<ActionResult<IEnumerable<Book>>> GetBooks()
{
    // Fetch all books from the database and return as a list
    return await _context.Books.ToListAsync();
}
```

**Explanation:** Retrieves all books from the database and returns them as a list.

- **Get Book by ID**

```
// GET: api/Books/{id}
[HttpGet("{id}")]
public async Task<ActionResult<Book>> GetBook(int id)
{
    // Find the book by ID in the database
    var book = await _context.Books.FindAsync(id);

    if (book == null)
    {
        // Return a 404 error if the book is not found
        return NotFound(new { message = $"Book with ID {id} not found." });
    }

    // Return the found book
    return book;
}
```

**Explanation:** Fetches a specific book using its ID and handles scenarios where the book is not found.

- **Add a New Book**

```
// POST: api/Books
[HttpPost]
public async Task<ActionResult<Book>> PostBook(Book book)
{
    // Validate the book model before adding it to the database
    if (!ModelState.IsValid)
    {
        // Return bad request if the model is not valid
        return BadRequest(ModelState);
    }

    // Add the new book to the database
    _context.Books.Add(book);
    await _context.SaveChangesAsync();

    // Return a 201 response with the location of the new book
    return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
}
```

**Explanation:** Validates and saves a new book record to the database, returning the newly created record.

- **Update Book**

```
// PUT: api/Books/{id}
[HttpPut("{id}")]
public async Task<ActionResult> PutBook(int id, Book book)
{
    // Check if the ID in the URL matches the ID in the request body
    if (id != book.Id)
    {
        return BadRequest(new { message = "Book ID in the URL does not match the ID in the payload." });
    }

    // Validate the book model before updating
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    // Check if the book exists in the database
    var existingBook = await _context.Books.FindAsync(id);
    if (existingBook == null)
    {
        return NotFound(new { message = $"Book with ID {id} not found." });
    }
}
```

**Explanation:** Updates an existing book's details after ensuring its presence in the database.

- **Delete Book**

```
// DELETE: api/Books/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteBook(int id)
{
    // Find the book by ID in the database
    var book = await _context.Books.FindAsync(id);
    if (book == null)
    {
        // Return a 404 error if the book is not found
        return NotFound(new { message = $"Book with ID {id} not found." });
    }

    // Remove the book from the database
    _context.Books.Remove(book);
    await _context.SaveChangesAsync();

    // Return a 204 No Content response to indicate the deletion was successful
    return NoContent();
}
```

**Explanation:** Removes a book from the database if it exists.

## 2.6. Authentication and Authorization

The AuthController handles user registration and login using JWT for secure authentication.

- **User Registration**

```
// POST: api/Auth/register
[HttpPost("register")]
public async Task<IActionResult> Register([FromBody] UserRegisterDTO userDTO)
{
    // Validate input fields: check if username, password, and confirm password are not empty
    if (string.IsNullOrEmpty(userDTO.Password) ||
        string.IsNullOrEmpty(userDTO.Username) ||
        string.IsNullOrEmpty(userDTO.ConfirmPassword))
    {
        return BadRequest(new { message = "Username, Password, and Confirm Password cannot be empty." });
    }

    // Check if the password and confirm password match
    if (userDTO.Password != userDTO.ConfirmPassword)
    {
        return BadRequest(new { message = "Password and Confirm Password do not match." });
    }

    // Check if the username already exists in the database
    if (await _context.Users.AnyAsync(u => u.Username == userDTO.Username))
    {
        return BadRequest(new { message = "Username already exists." });
    }

    // Use PasswordHasher to securely hash the password before storing it
    var passwordHasher = new PasswordHasher<User>();
    var hashedPassword = passwordHasher.HashPassword(new User(), userDTO.Password);
    // Map the DTO to a User model and save to the database
    var user = new User
    {
        Username = userDTO.Username,
        Password = hashedPassword,
        Role = userDTO.Role ?? "User" // Set default role as "User" if no role is provided
    };

    // Add the user to the context and save the changes
    _context.Users.Add(user);
    await _context.SaveChangesAsync();

    // Return a success message
    return Ok(new { message = "Registration successful." });
}
```

- **User Login**

```
[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] UserLoginDTO userDTO)
{
    // Validate input: check if username and password are provided
    if (string.IsNullOrEmpty(userDTO.Password) || string.IsNullOrEmpty(userDTO.Username))
    {
        return BadRequest(new { message = "Username and Password cannot be empty." });
    }

    // Retrieve the user from the database by username
    var existingUser = await _context.Users.FirstOrDefaultAsync(u => u.Username == userDTO.Username);
    if (existingUser == null)
    {
        return Unauthorized(new { message = "Invalid username or password." });
    }

    // Use PasswordHasher to verify the entered password against the stored hashed password
    var passwordHasher = new PasswordHasher<User>();
    var verificationResult = passwordHasher.VerifyHashedPassword(existingUser, existingUser.Password, userDTO.Password);

    if (verificationResult != PasswordVerificationResult.Success)
    {
        return Unauthorized(new { message = "Invalid username or password." });
    }

    // Create JWT claims based on the user data
    var claims = new[]
    {
        new Claim(ClaimTypes.Name, existingUser.Username),
        new Claim(ClaimTypes.Role, existingUser.Role ?? "User") // Default role to "User" if not provided
    };

    // Generate JWT token
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"] ?? string.Empty));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    // Create a new JWT token with the claims, expiration, and signing credentials
    var token = new JwtSecurityToken(
        issuer: _configuration["Jwt:Issuer"],
        audience: _configuration["Jwt:Audience"],
        claims: claims,
        expires: DateTime.Now.AddHours(1), // Token expires in 1 hour
        signingCredentials: creds);

    // Return the JWT token as a response
    return Ok(new { token = new JwtSecurityTokenHandler().WriteToken(token) });
}
```

**Explanation:**

- **Registration** hashes the user's password before storing it, ensuring secure handling of sensitive information.
- **Login** verifies credentials and generates a JWT token for the authenticated user.



## 2.7.Middleware Configuration

The backend configuration, located in Program.cs, sets up essential services and middleware.

```
// Add services to the container.
builder.Services.AddControllers();

// Configure SQLite Database
builder.Services.AddDbContext<LibraryContext>(options =>
    options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));

// Add JWT authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        var key = builder.Configuration["Jwt:Key"];
        if (string.IsNullOrEmpty(key))
        {
            throw new Exception("JWT Key is not configured in appsettings.json");
        }

        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key))
        };
    });
```

### Explanation:

- Configures SQLite as the database.
- Adds JWT authentication for secure access to protected API endpoints.

### 3. Frontend Development

The frontend of the Library Management System was built using **React** and **TypeScript**, with **Material-UI (MUI)** for styling and a focus on creating a responsive, user-friendly interface. It provides seamless integration with the backend RESTful APIs to enable book management and user authentication features.

This section details the technologies used, component structure, routing, state management, CRUD operations, error handling, and other essential aspects of the implementation.

#### 3.1. Technologies and Tools

1. **React**: Used for building dynamic and reusable user interface components.
2. **TypeScript**: Enforced type safety, reducing runtime errors and improving code maintainability.
3. **Material-UI (MUI)**: Provided responsive and pre-styled components, speeding up development and ensuring a consistent design language.
4. **Axios**: Simplified API communication for CRUD operations and authentication.
5. **React Router**: Handled routing and navigation between public and private pages.
6. **Context API**: Managed global authentication state across the application.



## 3.2.Component Structure

The application follows a modular component structure to promote scalability and maintainability. Below is an overview of the key components and their roles:

### 1. Core Components:

- **NavBar:** A responsive navigation bar that changes based on the user's authentication state.
- **BookList:** Displays a list of books in a card-based layout, with options to edit or delete records.
- **AddBook:** Provides a form for adding new books, with real-time validation and error feedback.
- **EditBook:** Allows editing existing book records fetched from the backend.

### 2. Authentication Components:

- **Login:** A login form that validates user credentials and sets authentication tokens.
- **Register:** A registration form for creating new accounts with validation for matching passwords.

### 3. Utility Components:

- **PrivateRoute:** Ensures only authenticated users can access protected routes (e.g., BookList, AddBook).
- **AuthContext:** Provides global authentication state and actions like login and logout.

### Example Component Structure (from AddBook.tsx):

```
const AddBook: React.FC = () => {
  const [title, setTitle] = useState("");
  const [author, setAuthor] = useState("");
  const [description, setDescription] = useState("");
  const [error, setError] = useState<string | null>(null);
  const [open, setOpen] = useState(true);
  const navigate = useNavigate();

  useEffect(() => {
    setOpen(true); // Ensure dialog is open on mount
  }, []);

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault(); // Prevent default form behavior
    setError("");

    if (!title || !author || !description) {
      setError("Please fill in all fields"); // Input validation
      return;
    }

    try {
      await addBook({ title, author, description }); // Add book via service
      navigate("/"); // Navigate to book list
    } catch (err) {
      setError("Failed to add book"); // Handle errors
    }
  };

  const handleClose = () => {
    navigate("/"); // Close dialog and navigate back
  };
};
```

**Why This Matters:** Each component is focused on a specific functionality, making the codebase easier to understand and extend.

### 3.3.Routing

Routing was implemented using **React Router**, allowing navigation between public and private routes. The PrivateRoute component ensures restricted access to authenticated users.

#### Example of Private Route:

```
<Route
  path="/add-book"
  element={
    <ProtectedRoute>
      {" "}
      {/* Ensure only authenticated users can access */}
      <NavBar />
      <AddBook /> {/* Add new book page */}
    </ProtectedRoute>
  }
/>
```

#### Explanation:

1. Public routes like /login and /register are accessible to all users.
2. Private routes such as /add-book and /edit-book/:id require authentication, handled by the PrivateRoute component.

### 3.4.State Management

State management is achieved through:

1. **Local State:** Managed with React's `useState` and `useEffect` hooks for individual components.
2. **Global State:** Authentication state is managed globally using the **Context API**, ensuring consistency across all components.

**Example from AuthContext.tsx:**

```
const login = () => {  
  setIsAuthenticated(true); // User is now authenticated  
  localStorage.setItem("auth_token", "dummy_token"); // Store the dummy token in  
};
```

**Why This Matters:** Centralized state management avoids redundant logic and ensures a consistent user experience.

### 3.5.CRUD Operations

CRUD operations are the core functionality of the application. Each operation is tightly integrated with the backend API.

#### 1. Create:

- Implemented in `AddBook.tsx` with input validation and a call to the `addBook` service.
- Example:

```
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault(); // Prevent default form behavior
  setError("");

  if (!title || !author || !description) {
    setError("Please fill in all fields"); // Input validation
    return;
  }

  try {
    await addBook({ title, author, description }); // Add book via service
    navigate("/"); // Navigate to book list
  } catch (err) {
    setError("Failed to add book"); // Handle errors
  }
};
```

#### 2. Read:

- Handled in `BookList.tsx` by fetching all book records from the backend and displaying them using MUI cards.

- Example:

```
useEffect(() => {  
  // Fetch books when the component is mounted  
  const fetchBooks = async () => {  
    try {  
      const data = await getBooks(); // Call API to get books  
      setBooks(data); // Update state with fetched books  
    } catch (error) {  
      console.error("Failed to fetch books:", error); // Log fetch errors  
    }  
  };  
  fetchBooks();  
}, []);
```

### 3. Update:

- Implemented in `EditBook.tsx` with pre-filled form fields fetched from the backend for the selected book.
- Example:

```
const handleSubmit = async (e: React.FormEvent) => {  
  e.preventDefault(); // Prevent default form behavior  
  setError("");  
  
  if (!title || !author || !description) {  
    setError("Please fill in all fields"); // Validate inputs  
    return;  
  }  
  
  try {  
    await updateBook(Number(id), {  
      id: Number(id),  
      title,  
      author,  
      description,  
    }); // Update book  
    setOpen(false); // Close dialog on success  
    navigate("/"); // Navigate back to book list  
  } catch (err) {  
    setError("Failed to update book"); // Handle update errors  
  }  
};
```

#### 4. Delete:

- Allows users to delete books directly from the `BookList` component.
- Example:

```
const handleDelete = async (id: number) => {  
  try {  
    await deleteBook(id); // Call API to delete a book  
    setBooks(books.filter((book: any) => book.id !== id)); // Update state to remove deleted book  
  } catch (error) {  
    console.error(`Failed to delete book with ID ${id}:`, error); // Log delete errors  
  }  
};
```

### 3.6.Error Handling

Error handling is integrated into various parts of the application to ensure robust user feedback and graceful failure management.

#### 1. Input Validation:

- Ensures users provide all required fields in forms.
- Example:

```
if (!title || !author || !description) {  
  setError("Please fill in all fields"); // Validate inputs  
  return;  
}
```

## 2. Fetching Errors:

- Caught using `try...catch` blocks to prevent crashes and log errors for debugging.
- Example:

```
const fetchBooks = async () => {
  try {
    const data = await getBooks(); // Call API to get books
    setBooks(data); // Update state with fetched books
  } catch (error) {
    console.error("Failed to fetch books:", error); // Log fetch errors
  }
};
fetchBooks();
}, []);
```

## 3.7.Styling

Material-UI (MUI) was used extensively for creating a visually appealing and responsive design. Additional CSS was applied for custom components like dialog boxes.

### Example of MUI Styling in `BookList.tsx`:

```
<Button
  variant="contained"
  sx={{
    backgroundColor: "black",
    color: "white",
    "&:hover": {
      backgroundColor: "#333",
    },
  }}
  component={Link}
  to="/add-book"
>
  Add Book
</Button>
```

**Why This Matters:** MUI ensures that the app adheres to modern design standards while remaining highly customizable.



## 4. Challenges and Solutions

The development of the Library Management System's frontend and backend presented several challenges. This section outlines key challenges encountered during the project and the solutions implemented to overcome them.

### 4.1. Managing Authentication State Across Components

#### **Challenge:**

Handling user authentication and ensuring restricted access to certain pages required a global state management solution. Additionally, managing state changes after logging in or out needed to be reflected across all components seamlessly.

#### **Solution:**

The **Context API** was implemented to manage authentication state globally. This approach allowed centralized management of user login and logout actions. For seamless updates across components, `useEffect` hooks were used to listen for changes in the authentication state and re-render dependent components..

## 5. Ensuring Secure Communication Between Frontend and Backend

#### **Challenge:**

Ensuring secure and consistent communication between the frontend and backend required implementing token-based authentication and protecting sensitive routes from unauthorized access.

#### **Solution:**

**JWT (JSON Web Token)** authentication was used to secure communication. The frontend sent the token in the `Authorization` header for every request to protected backend APIs. The token was validated on the backend before processing the request.

## 6. Responsive Design

### **Challenge:**

Ensuring that the application was fully responsive across different devices and screen sizes required careful planning and implementation.

### **Solution:**

**Material-UI (MUI)** was used to create responsive components. Its grid system and style utilities allowed for easy customization and consistency. Custom CSS was applied where additional styling was needed.

## 7. Error Handling for User Input and API Failures

### **Challenge:**

Handling errors effectively, such as form validation errors or API failures, was necessary to provide a smooth user experience.

### **Solution:**

Input validation was implemented in form components (`AddBook.tsx`, `EditBook.tsx`, etc.) to catch errors before making API calls. For API failures, **Axios** interceptors were used to standardize error messages across the application.

## 8. Securing Protected Routes

### **Challenge:**

Restricting access to certain pages (e.g., Book List, Add/Edit Book) to authenticated users was critical for maintaining security.

### **Solution:**

The `PrivateRoute` component was implemented to wrap protected routes. It checks the user's authentication state and redirects unauthenticated users to the login page.

## 9. Debugging API Calls and Network Issues

### **Challenge:**

Debugging issues related to API calls, such as incorrect payloads or network failures, slowed down the development process.

### **Solution:**

Console logging and Axios interceptors were used to track and debug API calls. Additionally, `try...catch` blocks were added in all API-related functions to capture and log errors for further analysis.

## 10.Validation of Backend Responses

### **Challenge:**

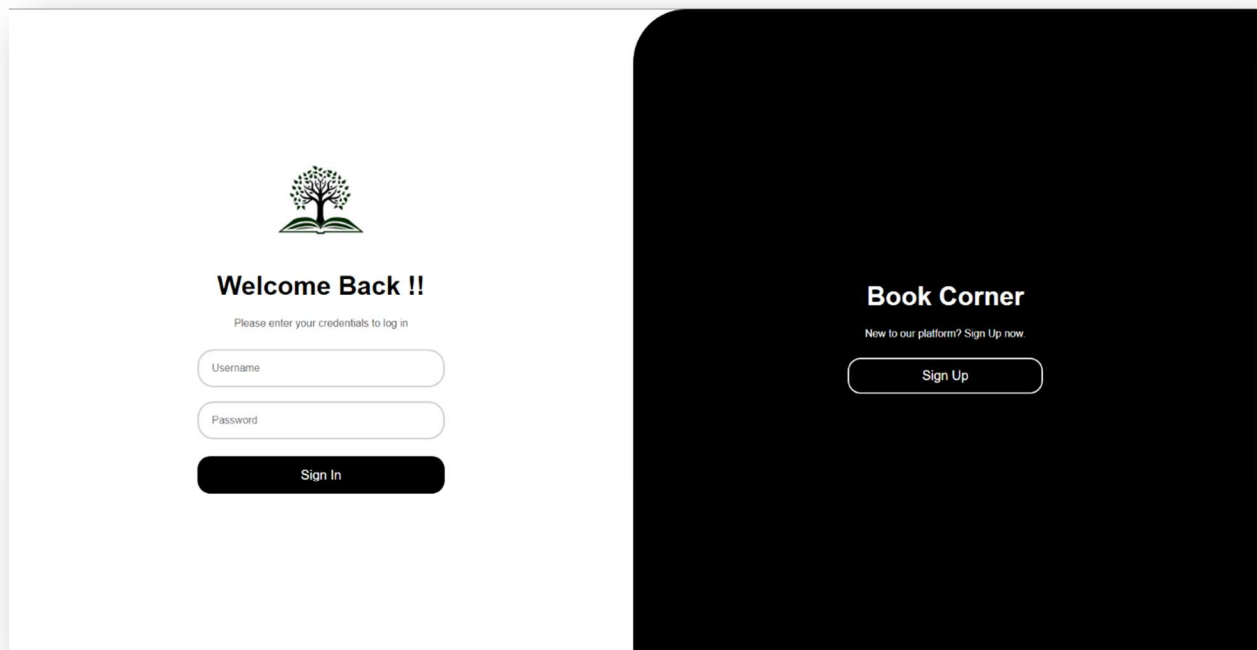
Ensuring the frontend handled unexpected backend responses (e.g., missing fields, validation errors) without breaking the UI was crucial.

### **Solution:**


Responses from the backend were validated before updating the UI. For instance, the `AddBook` component checked the response status and displayed appropriate messages for errors.

## 11. User Interface Designs

### 11.1. Login Page



The image shows a login page design with a white background on the left and a black background on the right. The white section features a logo of a tree with roots, the text "Welcome Back !!", a prompt "Please enter your credentials to log in", and input fields for "Username" and "Password". A black "Sign In" button is at the bottom. The black section features the text "Book Corner", a prompt "Now to our platform? Sign Up now.", and a white "Sign Up" button.



**Welcome Back !!**

Please enter your credentials to log in

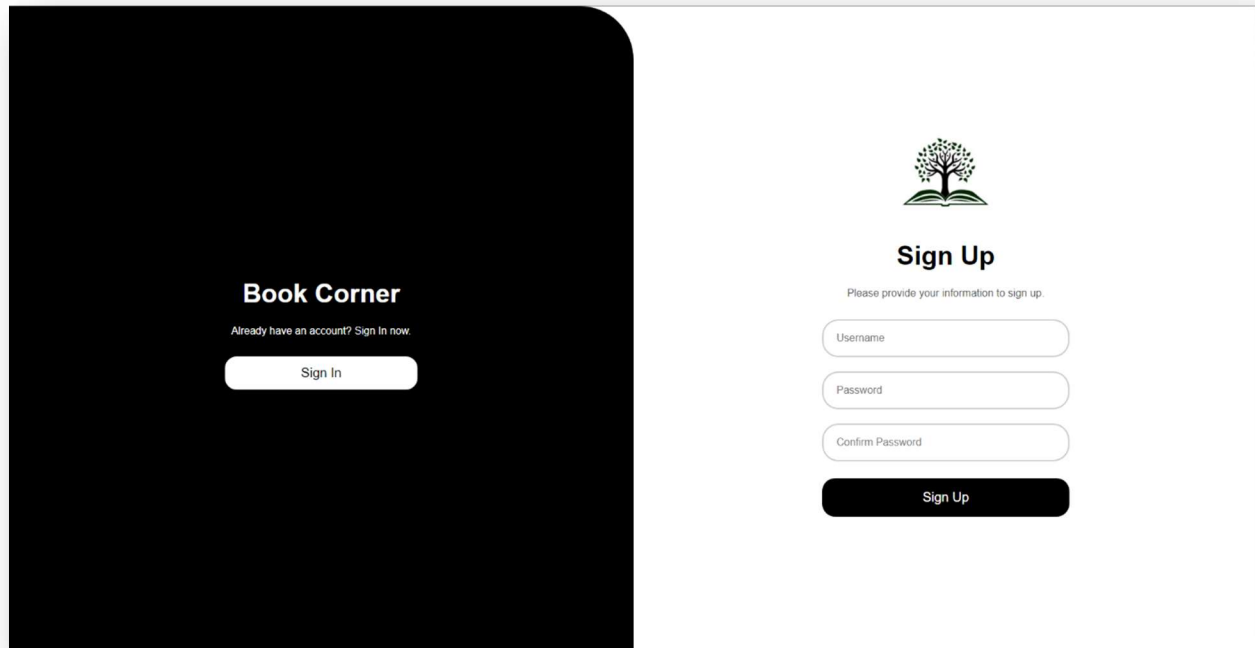
**Sign In**

**Book Corner**

Now to our platform? Sign Up now.


**Sign Up**

## 11.2. Register Page

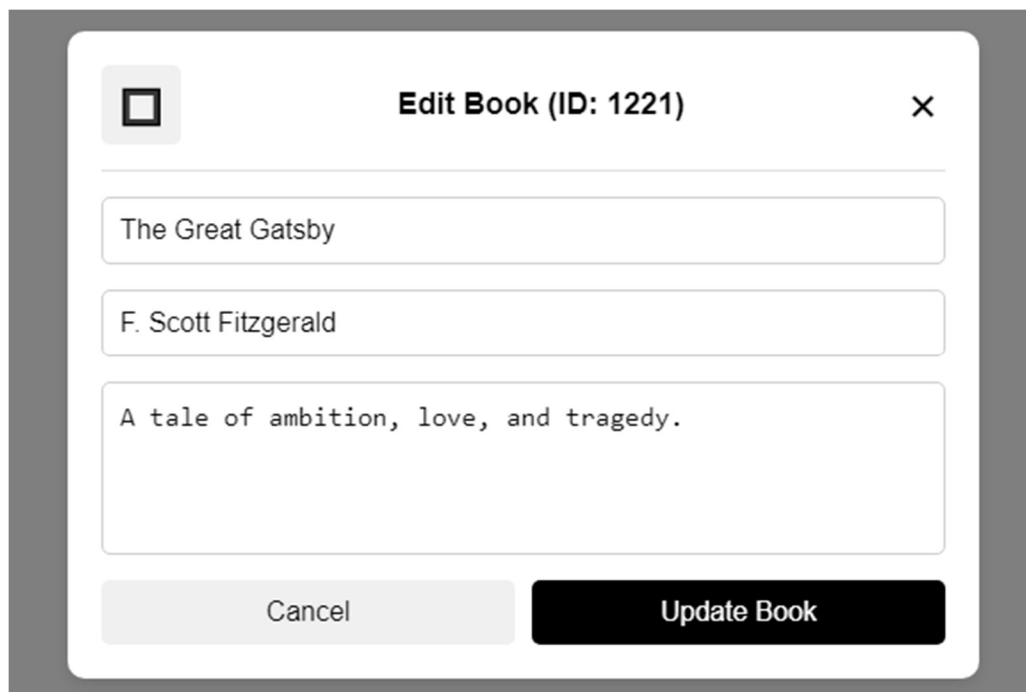


The Register Page is a split-screen layout. The left side is a solid black panel with the text "Book Corner" in white, followed by "Already have an account? Sign In now." and a white "Sign In" button. The right side is white and features a tree logo above the "Sign Up" heading. Below the heading is the instruction "Please provide your information to sign up." and three input fields for "Username", "Password", and "Confirm Password". A black "Sign Up" button is at the bottom.



**Book Corner**  
Already have an account? Sign In now.  
Sign In

  
**Sign Up**  
Please provide your information to sign up.  
Username  
Password  
Confirm Password  
Sign Up

## 11.3. Edit Book



The Edit Book modal is a white box with a gray border. It has a title bar with a close button (X) and the text "Edit Book (ID: 1221)". Below the title bar are three input fields: "The Great Gatsby", "F. Scott Fitzgerald", and "A tale of ambition, love, and tragedy.". At the bottom are two buttons: "Cancel" and "Update Book".

 **Edit Book (ID: 1221)** 


The Great Gatsby

F. Scott Fitzgerald

A tale of ambition, love, and tragedy.

Cancel Update Book

## 11.4. Add Book



Add New Book

×

Title

Author

Description

Cancel

Add Book


## 11.5. Home Page

Book Corner

LOGOUT

Books

ADD BOOK




ID: 1221

The Great Gatsby

Author: F. Scott Fitzgerald

A tale of ambition, love, and tragedy.

EDITDELETE




ID: 1228

Pride and Prejudice

Author: Jane Austen

Love confronts societal norms

EDITDELETE




ID: 1229

To Kill a Mockingbird

Author: Harper Lee

Justice clashes with prejudice.

EDITDELETE



ID: 1231

The Catcher in the Rye

Author: J.D. Salinger

Youth's struggle with identity

EDITDELETE

## 12. Key Learnings

The development of the Library Management System provided several valuable insights and learning opportunities. This section highlights the technical, problem-solving, and project management skills gained throughout the project.

### 12.1. Technical Skills

- **React with TypeScript:** Gained hands-on experience in building a dynamic and type-safe frontend using React and TypeScript. This improved understanding of state management, routing, and component-based architecture.
- **Material-UI (MUI):** Learned how to use Material-UI to create a responsive and visually appealing user interface. This included customizing components and utilizing MUI's grid system for responsive design.
- **ASP.NET Core:** Enhanced backend development skills by implementing RESTful APIs, integrating Entity Framework Core, and configuring middleware for authentication.
- **JWT Authentication:** Gained a practical understanding of implementing secure login mechanisms using JSON Web Tokens (JWT) for protected routes.
- **SQLite and Entity Framework Core:** Learned how to design and manage databases with SQLite and use Entity Framework Core for seamless database operations.

### 12.2. Problem-Solving

- **Error Handling:** Developed effective strategies for handling errors on both the frontend and backend. This included using Axios interceptors and `try...catch` blocks for debugging and error recovery.
- **API Integration:** Addressed challenges in connecting the frontend and backend by creating reusable service files and ensuring proper request-response handling.



- **State Management:** Gained insights into managing global and local states efficiently using Context API and React hooks (`useState`, `useEffect`).

### 12.3. Project Management

- **Time Management:** Successfully planned and executed the project within the given deadline by prioritizing tasks and breaking them into manageable parts.
- **Version Control:** Learned to use Git and GitHub for version control, ensuring safe and organized code collaboration.
- **Testing and Debugging:** Used tools like Postman and Swagger to validate API endpoints, enabling quicker identification and resolution of issues.

### 12.4. Personal Growth

- **Adaptability:** Adapted to learning new tools and frameworks, such as TypeScript and Material-UI, during the development process.
- **Attention to Detail:** Improved the ability to write clean, maintainable code by adhering to best practices and validating inputs thoroughly.

## 13. Conclusion

The development of the Library Management System successfully addressed the core objectives of creating a robust and user-friendly platform for managing book records. By integrating a secure backend with a responsive frontend, the system provides seamless functionality for CRUD operations and user authentication.

Despite challenges such as managing authentication, ensuring API integration, and handling responsiveness, these were effectively resolved through the use of modern tools like React, TypeScript, Material-UI, and ASP.NET Core. Testing via Postman and Swagger ensured the application's reliability and error handling capabilities.

## **Future Enhancements**

1. Implement automated testing with tools like Jest or Cypress for frontend validation.
2. Add features like search, sorting, and pagination for better user experience.
3. Deploy the system on a production environment, such as Azure or AWS, for wider access.
4. Enhance the UI with additional themes and accessibility features.

The project demonstrates the application of best practices in software development, creating a scalable and maintainable system that can be further extended to meet evolving needs.