LIBRARY MANAGEMENT SYSTEM

SE Assignment - Expernetic

SEPTEMBER 12, 2025

HASHIR AHAMED

1. Introduction

This project is a simple Library Management System built as part of the Software Engineering Internship assignment. The application allows users to:

- Create new book records
- View a list of books
- Update existing book details
- Delete books

The system is implemented as:

- A C# .NET Web API backend with an SQLite database (using Entity Framework Core)
- A React + TypeScript frontend styled with Tailwind CSS
- JWT-based authentication, which I implemented to secure book management actions

The goal of the project is to demonstrate my ability to design, implement, and integrate a full-stack CRUD application using modern tools and best practices.

2. System Architecture

The solution follows a classic client–server architecture.

2.1 Backend

- **Framework:** ASP.NET Core Web API
- **Language:** C#
- **Database:** SQLite
- **ORM:** Entity Framework Core
- **Auth:** JWT (JSON Web Token) authentication and authorization
- **Tools:** Swagger for API exploration and testing

The backend exposes RESTful endpoints under /api/Book and /api/User. It is responsible for:

- Persisting books in the SQLite database
- Performing validation and error handling

- Authenticating users and issuing JWT tokens
- Authorizing access to protected endpoints

2.2 Frontend

- **Framework:** React (with Vite)
- **Language:** TypeScript
- **Styling:** Tailwind CSS
- **HTTP client:** Axios
- **Routing:** react-router-dom

The frontend runs as a single page application (SPA) that communicates with the backend via HTTP. It handles:

- Displaying and searching the list of books
- Showing a modal for adding and editing books
- Login and registration of users

2.3 Data Model

The main entity in the system is the **Book**:

- Id (int, primary key)
- Title (string, required)
- Author (string, required)
- Description (string, required)
- Units (int, required – number of available copies)

For authentication, there is also a **User** entity (stored in the database) that contains:

- Id
- Name
- Username
- Password (hashed)
- Role (e.g. "Admin")

3. Backend Implementation

3.1 Project Structure

Main backend files:

- Program.cs – application startup, services, middleware
- Data/Db.cs – EF Core DbContext
- Models/Book.cs – Book entity with validation attributes
- Models/User.cs and Models/UserDto.cs – User and DTO for auth endpoints
- Controllers/BookController.cs – CRUD endpoints for books
- Controllers/UserController.cs – registration, login, user listing
- appsettings.json – configuration including SQLite connection string

3.2 BookController – CRUD Endpoints

The BookController exposes RESTful endpoints under api/Book and includes error handling with try/catch blocks. Key actions:

- **GET /api/Book** – returns all books
- **GET /api/Book/{id}** – returns a book by ID
- **POST /api/Book** – creates a new book
- **PUT /api/Book/{id}** – updates a book
- **DELETE /api/Book/{id}** – deletes a

3.5 UserController – Authentication

UserController handles registration and login:

- **POST /api/User/register**
    - Checks if username already exists
    - Hashes the password using SHA-256

    - Creates a new user with role "Admin"
- **POST /api/User/login**
    - Verifies username and password
    - If valid, generates a JWT token and returns it as plain text

4. Frontend Implementation

4.1 Project Setup and Structure

The frontend was created using **Vite + React + TypeScript** and styled with **Tailwind CSS**.

Key files:

- src/App.tsx – top-level component, handles auth routing and logout
- src/pages/LoginPage.tsx – login/register UI
- src/pages/Books.tsx – main Book Management page
- src/components/AddForm.tsx – reusable modal for adding/editing books
- src/services/api.ts – book API helper functions
- src/services/loginApi.ts – auth API and Axios instance
- src/types/book.ts – TypeScript Book type
- .env – contains VITE_BASE_URL pointing to the backend API

4.2 Book Management UI

The BookList page:

- Fetches books on mount using getBooks()
- Stores them in state and supports live **search** by title or author
- Displays each book in a styled list item with:
    - First letter avatar
    - Title

    - “Edit” and “Delete” buttons
    - A chevron to expand and show author, description, and units

The **Add/Edit** modal is implemented in AddForm.tsx:

- Controlled by showAddModal and editingBook in BookList
- If initialBook is provided → modal is in **Edit** mode
- If initialBook is null → modal is in **Add** mode
- On submit, it calls onSubmit with either:
    - A new book (no id) → calls createBook
    - An existing book with id → calls updateBook

The **Delete** button:

- Asks for confirmation (confirm("Are you sure..."))
- Calls deleteBook(id)
- Updates local state to remove the book from the list

5. Error Handling and Validation

- Backend endpoints are wrapped in try/catch blocks and return appropriate HTTP status codes:
    - 400 BadRequest for invalid input or mismatched IDs
    - 404 NotFound when an entity doesn’t exist
    - 500 InternalServerError for unexpected exceptions
- Frontend catches Axios errors and:
    - Displays friendly messages on the login page
    - Logs errors in the console for debugging
- Model validation attributes on the Book class ensure that incomplete or invalid data is rejected by the server.

6. Security and Authorization

Security aspects:

- **JWT Authentication**:
  - Users log in via /api/User/login
  - On success, the API returns a JWT token
  - Token includes username and role claims
- **Authorization**:
  - [Authorize] is applied at the BookController level
  - GET endpoints are explicitly marked [AllowAnonymous] (public)
  - POST, PUT, and DELETE require [Authorize(Roles = "Admin")], meaning only authenticated admins can modify books
- **CORS**:
  - Configured to allow only the frontend origin (http://localhost:5173) to call the API

For a production system, password hashing and token management would be further strengthened (e.g., using ASP.NET Identity, salted hashes, refresh tokens, etc.), but for this assignment the implementation is intentionally lightweight and easy to explain

7. Challenges and Key Learnings

During the development of this assignment, I faced several practical challenges that helped me improve both my backend and frontend skills:

- **Adapting to the .NET development workflow**

Even though I had basic knowledge of C#, I had not built a full project using ASP.NET Core before. Understanding project structure, controller interactions, dependency injection, and EF Core setup took time to get familiar with. Learning shortcuts, function naming conventions, and how the framework handles requests was a valuable experience.

-
-

- **Authentication and Authorization**

Implementing JWT authentication required a deeper understanding of how tokens are generated, validated, and passed between the backend and frontend. Setting up role-based authorization, protecting endpoints, and configuring middleware in the correct order took experimentation and debugging.

- **Modal UI behavior**

My modal initially caused a blocked UI because it stayed active even after closing actions. Debugging the visibility state logic taught me good practices for controlled UI components.

Key Learnings

This project helped me gain hands-on knowledge of:

- Structuring real-world .NET APIs using best practices
- JWT security workflow (login → token → protected endpoints)
- Database design and migration processes in EF Core
- Effective API consumption and state management in React
- Thinking about UI/UX behavior and edge cases
- Full-stack debugging and error tracing

8. Instructions for Running the Application

8.1 Prerequisites

- **.NET SDK** (compatible with the project's target framework)
- **Node.js and npm**
- **Git** (optional, for cloning the repository)
- A code editor such as **Visual Studio** or **VS Code**

8.2 Clone the Repository

git clone https://github.com/HashirAhamed/library-management-system.git

Assume the project structure:

- backend/LibraryApi – ASP.NET Core Web API
- frontend – React + TypeScript app

8.3 Configure the Backend

1. Open the backend folder:
2. cd backend/LibraryApi
3. Check the **connection string** in appsettings.json:

   ```
   "ConnectionStrings": {
     "DefaultConnection": "Data Source=library.db"
   }
   ```

This will create a library.db SQLite file in the project folder.

4. Apply Entity Framework migrations (if not already applied):
5. dotnet ef database update
6. dotnet tool install --global dotnet-ef
7. Run the backend:
8. dotnet run

The API should start on URLs similar to:

   - https://localhost:7114
   - http://localhost:3000 (Kestrel HTTP port)

9. Test Swagger:
   - Open a browser and go to: https://localhost:7114/swagger

8.4 Configure the Frontend

1. Open a new terminal and go to the frontend folder:
2. cd frontend
3. Install dependencies:
    - npm install
4. Create a .env file in the frontend folder (if not already present) and set:
    - VITE_BASE_URL=https://localhost:7114/api

This must match the backend URL and port where the API is running.

5. Start the frontend development server:
    - npm run dev
6. Open the application in a browser:
    - Vite usually runs on: http://localhost:5173

8.5 Using the Application

1. **Register Admin User**
    - On first launch, you will see the **Login / Register** screen.
    - Click **"Register here"**, fill in name, username, and password, and submit.
    - The backend will create a user with role "Admin".
2. **Login**
    - Enter the username and password.
    - On success:
        - The frontend stores the JWT in localStorage under authToken.
        - isAuthenticated becomes true and the app navigates to the **Book Management** page.
3. **Book Management**
    - You will see a list of books (if any exist).

    - Use the **search bar** to filter by title or author.
    - Click **“Add Book”** to open the modal and create a new record.
    - Click **“Edit”** on a book to update its details using the same modal.
    - Click **“Delete”** to remove a book (after confirmation).

4. **Logout**
    - Click the **Logout** button in the top-right corner.
    - This clears the token from localStorage and returns you to the login screen.

10. Conclusion

This assignment allowed me to design and build a full-stack Library Management System using C# .NET, SQLite, React, and TypeScript. I implemented complete CRUD functionality for books, working JWT-based authentication, and a responsive, user-friendly interface.

The project demonstrates:

- Clean separation between backend and frontend
- Proper use of RESTful APIs and HTTP status codes
- Database integration via Entity Framework Core
- Basic but functional authentication and authorization
- State management, routing, and API integration on the frontend

These components together form a complete, functional solution that can be run locally and extended further if needed.