Library Management System

By

Nirmada Edirisinghe

Internship Assignment

2025/12/09

GitHub Repo: https://github.com/nirmada001/LibraryApi.git

Table of Contents

Table of Figures:

1. Introduction

The Library Management System project was developed as a full-stack software engineering assignment. The objective of the project was to build a simple library management system using ASP.NET and React with Typescript (web technologies.). The system allows users to manage books by performing CRUD operations, Create, Read, Update, and Delete.

The backend of the project was developed using ASP.NET Core Web API, exposing the RESTful api endpoints to interact with SQLite database through Entity Framework Core. For the frontend development React with Typescript was used. Tailwind CSS was used for clean and responsive user interface. Axios call was used for all HTTP requests, and the client side navigation is handled through React Router.

The purpose of the project is to demonstrate full-stack development skills, database integration, API development, and UI design in a real-world scenario.

2. System Overview

The system follows client-server architecture, the frontend works as a standalone react application that runs in the user's browser. Frontend communicates with the backend through RESTful API calls using HTTP methods like GET, POST, PUT, and DELETE. Processing requests, validating data, interacting with the database, and returning responses are handled through the backend. When the backend is started, the API endpoints are exposed and it provides Swagger documentation to test the APIs. SQLite database (library. DB) stored the book records. It is a lightweight, file-based database ideal for small projects. The overall architecture of the project ensures clean separations. The frontend handles user interaction and presentation while the backend handles the logic and database operations.

3. Backend Implementation

Backend of the project was created using ASP.NET Core Web API. Controllers, data access classes, model definitions, and migration files are included in the structure. Book model created included 4 fields Id, Title, Author, and Description with Title and Author marked as required through annotations. Validation in backend enforced through model binding and ModelState checks. Database operations are handle through Entity Framework Core with LibraryContext class extending DbContext.

BooksController is implemented to handle all the CRUD operations. GET, POST, PUT, and DELETE methods are implemented in the books controller. Each endpoint also returns appropriate HTTP responses like 200 OK, 400 BadRequest, 404 NotFound etc..CORS was configured to allow the react frontend to access the APIs. Swaggers was used for testing the API endpoints before integrating with frontend.

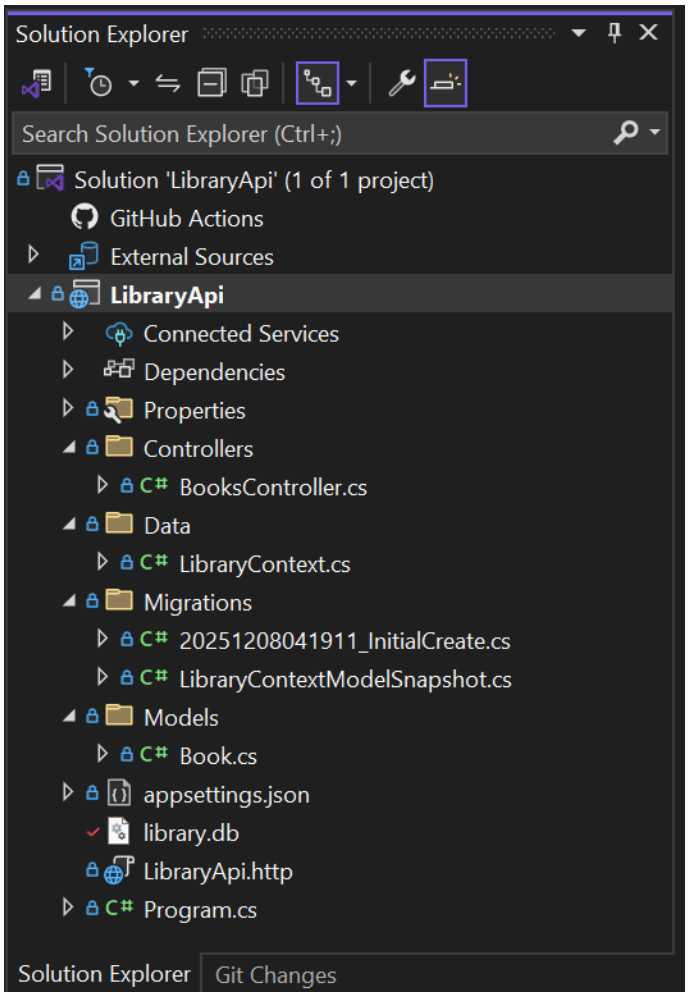

Figure 1: Backend folder Structure

4. Database Design

SQLite is used as the database due to it's simplicity and suitability for small applications. Here, the database consist of a single table named Books. The table contains columns for Id (Primary key, Auto-increment), Title (Required), Author (Required), and Description (Text). Entity Framework Core migrations was used to generate the database and table structure automatically. Commands dotnet ef migrations add InitialCreate and dotnet ef database update was run during the process. Since the SQLite is file based, the entire database exists in library.db file located inside the backend project folder.

5. API Documentation.

The API follows a RESTful design. Bellow are the APIs used in the project.

- GET/api/books – API to retrieve all books from the databse.
- GET/api/books/{id} – API to retrieve a single book based on its Id.
- POST/api/books – API to create a new book. Validates required fields and checks for duplicate titles.
- PUT/api/books/{id} – API to update book details.
- DELETE/api/books/{id} – API to delete a book if it exists.

All the endpoints are designed to return proper HTTP responses.

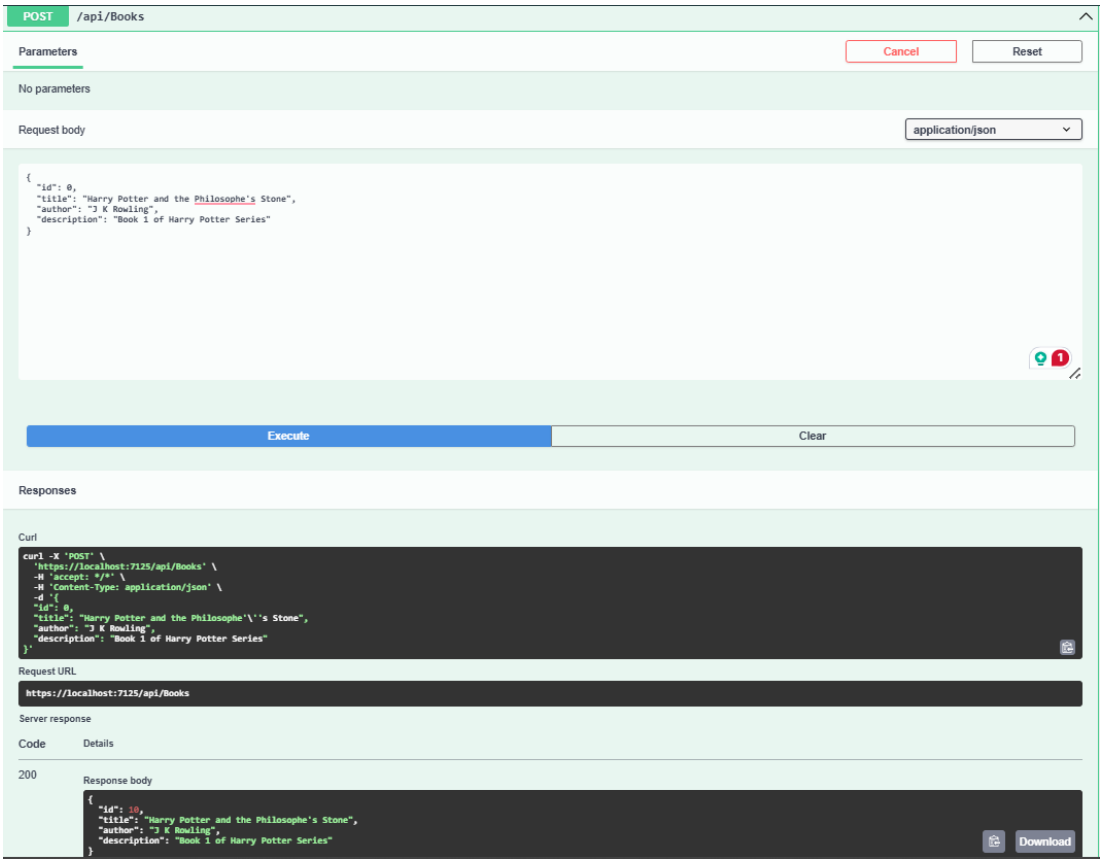

Figure 2: POST API Swagger UI

6. Frontend Implementation.

The frontend of the application was developed using React and Typescript. Vite was used as the build tool to provide a fast development. The project uses a folder structure that separates pages, types, and API helper functions. Axios was used to communicate with the backend, and helper functions for making the API requests were included in the booksapi.ts file. getBooks, getBooksById, createBook, updateBook, and deleteBook functions are included in the file. Page navigation is ensured by using React Router. The application consist four main pages: Home Page, Add Book Page, Edit Book Page, and Book List page, all these files are implemented sperately for better maintainability. React hooks like useEffect, useState, useNaviagate are used for state management. Finally the styling of the pages was done using Tailwind CSS.

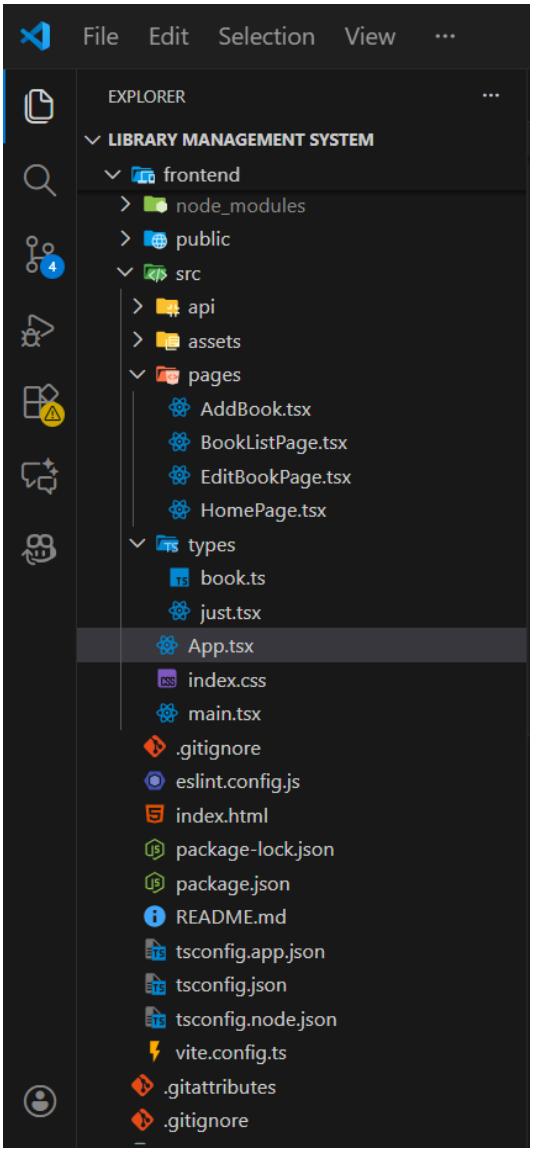

Figure 3: Frontend Folder Structure

7. Routing Implementation.

To ensure the frontend as a complete system Routing is used. React Router is configured in the App.tsx file to provide clean navigation across different pages of the system.

Routes included in the library management system.

- / - Home page showing main navigation option.
- /books – Display list of books retrieved from the backend.
- /add-book – Display a form to add a new book.
- /edit-book/:id – Displays an edit form pre-filled with an existing book data.

The router uses <Router>, <Routes>, and <Route> components. <Link> and useNavigate hook is used for the navigation within the application.

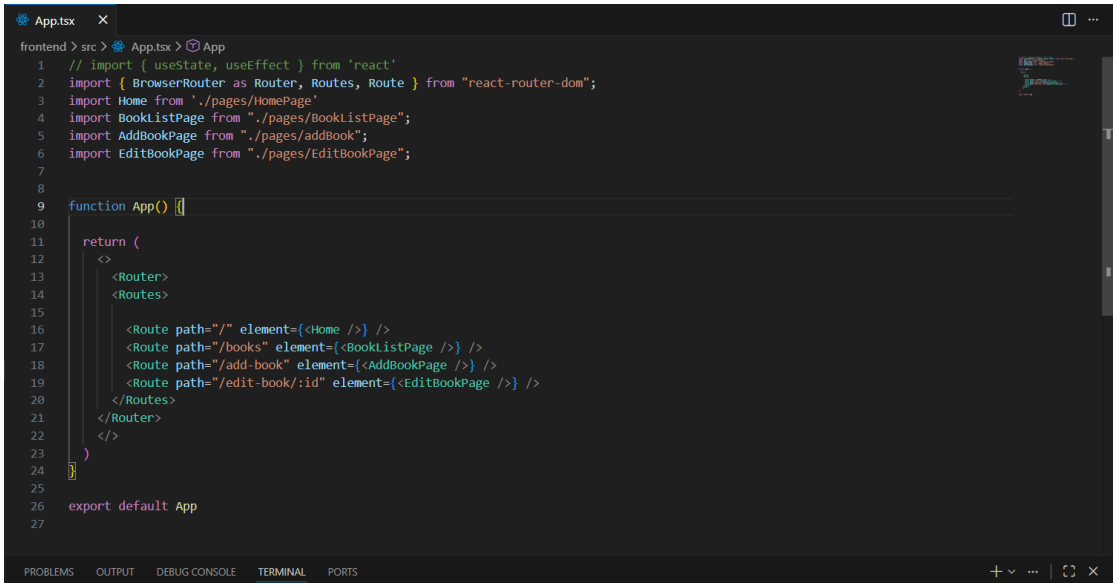

Figure 4: Routing implementation in App.tsx file

8. User Interface and Tailwind Styling.

The UI design focuses on clarity, simplicity and ease of use. Tailwind CSS was used because it allows styling directly inside tsx files using utility classes like bg-gray-500, border, rounded-lg etc…The pages contained structured card layouts, forms, and lists easy to read. Buttons include color-coded styles. Error and success messages are displayed using alert like styling along with dismiss buttons.

1. Home page:

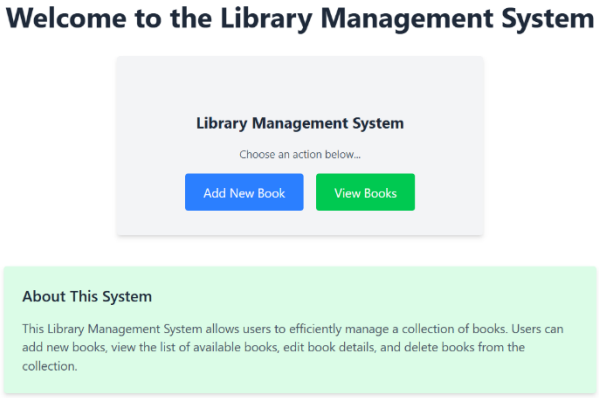

Figure 5: Home Page UI

2. Book List Page:

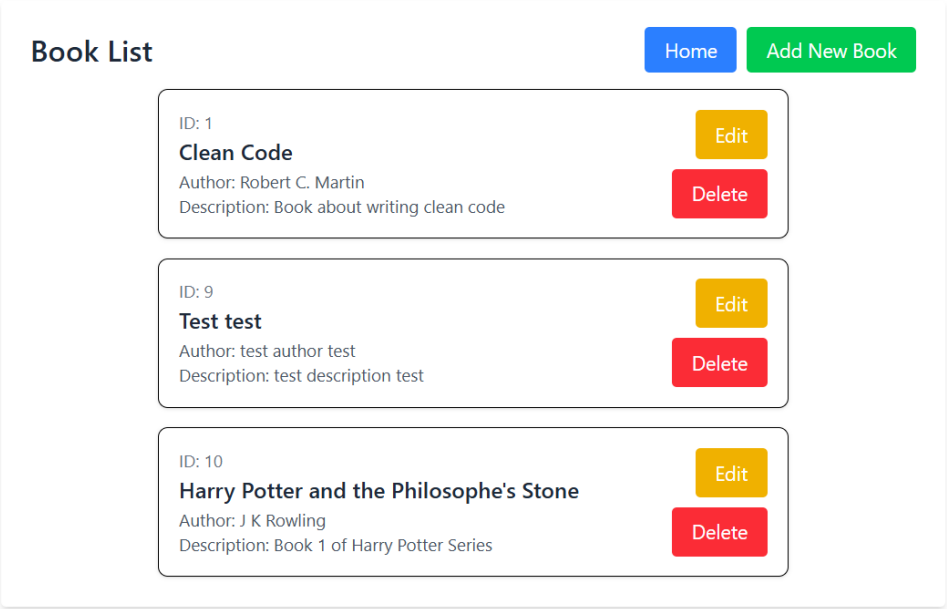

Figure 6: Book List Page UI

3. Add Book Page:

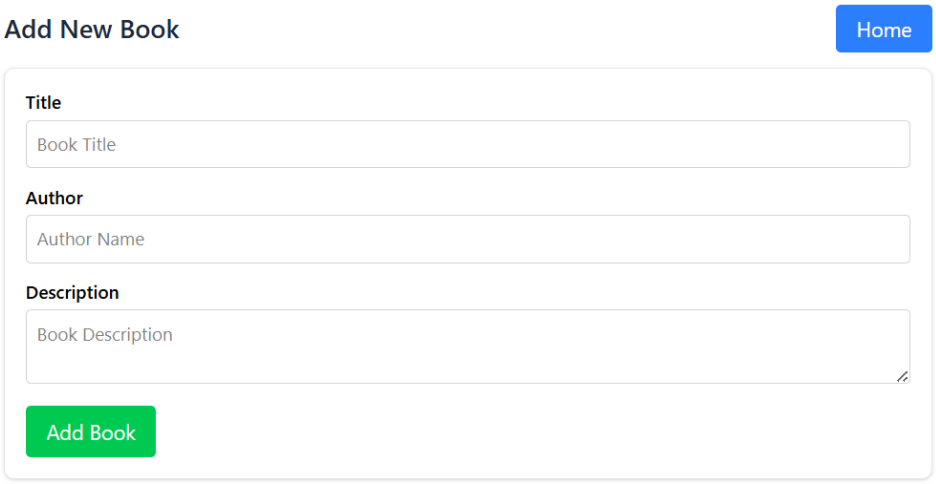

Figure 7: Add Book Page UI

4. Edit Book Page:

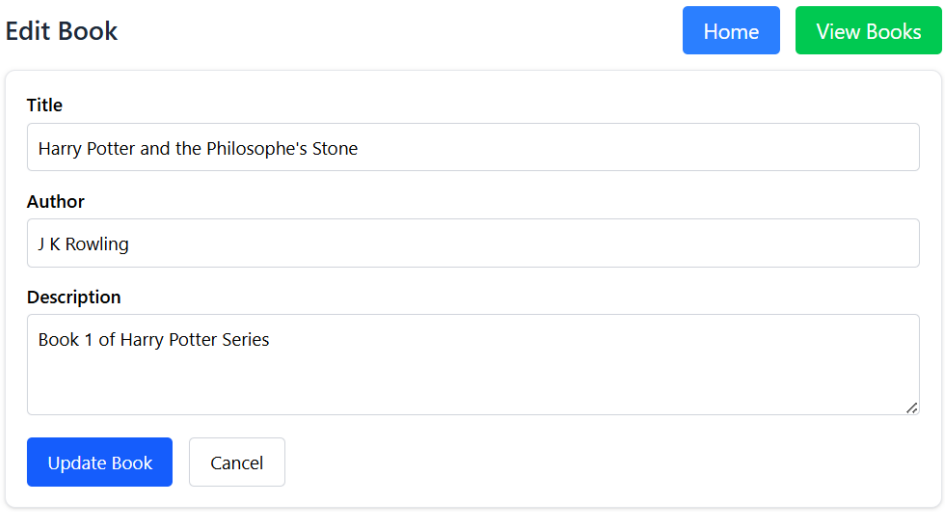

Figure 8: Edit Book Page UI

5. Delete book:

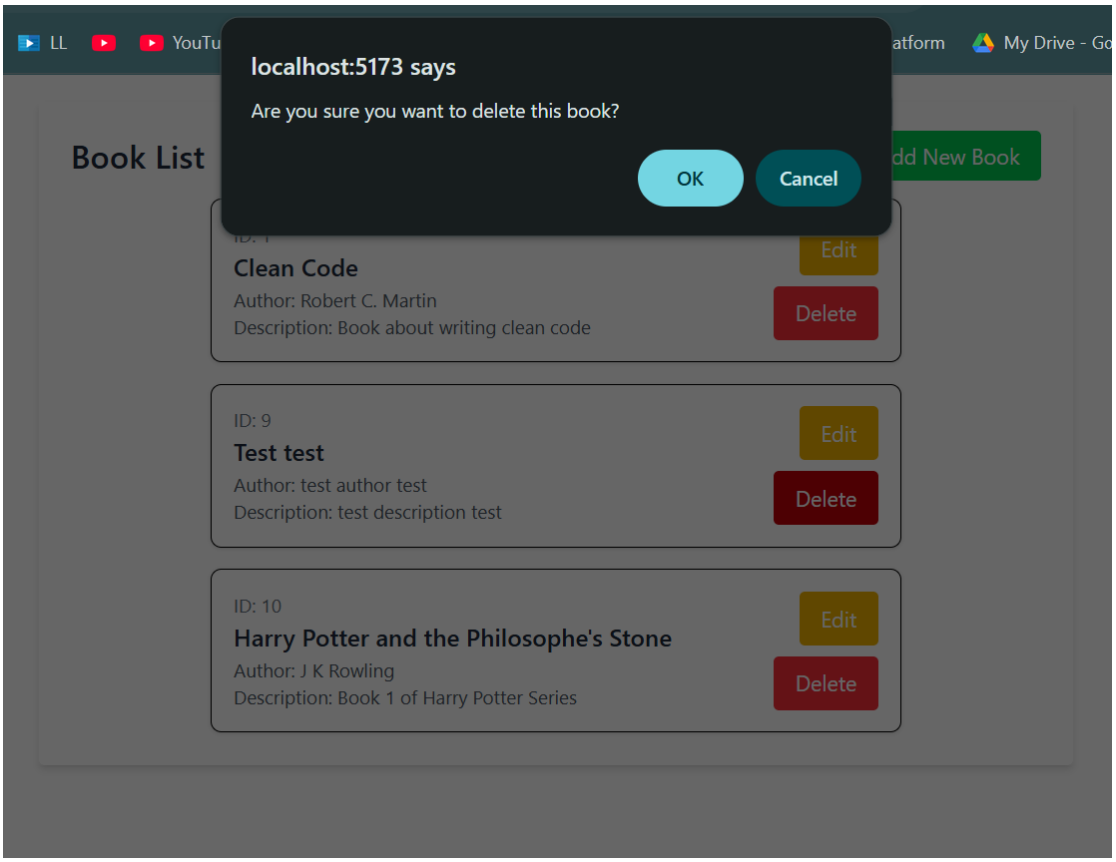

Figure 9: Book Deletion Confirmation

9. Testing and Verification

During system implementation both frontend and backend were tested thoroughly. The backend API testing was mainly done using Swagger UI, which allowed sending requests for all CRUD operations. All responses were confirmed to return appropriate responses. SQLite database was verified to correctly store and update book records.

Frontend testing including verifying proper navigation, form validation, Axios API integration, and UI updates. The delete action incudes a confirmation popup for safety. The system works end to end, ensuring that changes made via the frontend reflect correctly in the database.

10. Challenges Faced and Solutions.

During the implementation I faced several challenges. During the database implementation I faced the challenge of understanding about SQLite database, because I have not worked with it before. But I made this an opportunity to learn and faced the challenge well by figuring out about SQLite and why It is suitable for a simple project like this. Afterwards I found it a little bit difficult to configure SQLite through EF core migration, but was able to resolve by properly trying the commands.

Another difficulty I faced was when I connected the frontend and backend. Even though the API URLs were properly coded, it was shown in the console that the frontend URL was blocked by the backend. Then I figured out that this was a CORS policy issue, and solved it by enabling CORS in Program.cs file allowing the frontend origin. Another issue was when developing the frontend, since I am not a lot familiar with Typescript it was a little bit confusing, but was able to quickly figure it out and solve the issues.
Those are some of the challenges I faced and how I solved them to successfully complete the Library management system.

11. Conclusion

The project successfully demonstrates the complete implementation of a full-stack CRUD application. The backend provide a reliable and well-structured REST API with proper validation and database management using SQLite and Entity Framework core.

The frontend offers a simple, clean, and responsive user interface built using React, Typescript, and Tailwind CSS. The use of React Router enhances the user experience providing smooth navigation between pages. The requirements in the assignment have been met, and the system is fully functional and maintainable. This project highlights my skills in both frontend and backend development, API design, database management, and usage of modern web technologies.