Library Management System - Report

1. Executive Summary

Project Overview

The Library Management System is a full-stack web application designed to provide robust, maintainable, and scalable management of a library's book collection. The system enables users to perform comprehensive CRUD (Create, Read, Update, Delete) operations on book records through a modern, responsive web interface.

Key Achievements

- Clean Architecture Backend: Implemented a RESTful API using ASP.NET Core with clear separation of concerns through Controller-Service-Data layers.
- Component-Based Frontend: Developed a responsive Single Page Application (SPA) using React, TypeScript, and Vite.
- End-to-End Type Safety: Achieved data integrity by mirroring backend DTOs as TypeScript interfaces.
- Mirror Validation Strategy: Implemented dual-layer validation (frontend and backend) for robust data integrity.
- Production-Ready Code: Emphasized clean code principles, dependency injection, and testability throughout.

Technologies Used

Layer	Technologies
Backend	C#, ASP.NET Core 8, Entity Framework Core, SQLite, Swagger
Frontend	React 18, TypeScript, Vite, React Router DOM
Tools	Visual Studio 2022, Node.js, npm

2. System Architecture Overview

2.1 Architecture Principles

1. Separation of Concerns: Each layer has distinct responsibilities.
2. Dependency Inversion: High-level modules don't depend on low-level modules.
3. Interface-Based Design: Programming to interfaces rather than implementations.
4. Testability: Architecture designed for easy unit and integration testing.

3. Backend Implementation (ASP.NET Core)

3.1 Project Structure

```
LibraryManagementAPI/
│
├── Controllers/
│   └── BooksController.cs           # HTTP endpoint handlers
│
├── Services/
│   └── BookService.cs                # Business logic implementation
│
├── Interfaces/
│   └── IBookService.cs               # Service contract
│
├── Data/
│   └── LibraryContext.cs             # EF Core DbContext
│
├── Models/
│   └── Book.cs                        # Database entity
│
├── DTOs/
│   ├── CreateBookDTO.cs               # POST request body
│   ├── UpdateBookDTO.cs                # PUT request body
│   └── BookResponseDTO.cs              # GET response body
│
├── Program.cs                          # DI and middleware configuration
├── appsettings.json                    # Configuration settings
└── LibraryManagementAPI.csproj         # Project dependencies
```

3.2 API Endpoints Summary

Method	Endpoint	Request Body	Response	Status Codes
GET	/api/books	None	BookResponseDTO[]	200 OK
GET	/api/books/{id}	None	BookResponseDTO	200 OK, 404 Not Found
POST	/api/books	CreateBookDTO	BookResponseDTO	201 Created, 400 Bad Request
PUT	/api/books/{id}	UpdateBookDTO	BookResponseDTO	200 OK, 404 Not Found, 400 Bad Request
DELETE	/api/books/{id}	None	string message	200 OK, 404 Not Found

4. Frontend Implementation (React + TypeScript + Vite)

```
library-frontend/
│
├── index.html                        # Vite entry point
├── vite.config.ts                     # Vite configuration
├── tsconfig.json                       # TypeScript configuration
│
├── src/
│   ├── main.tsx                        # React entry point
│   ├── App.tsx                          # Main component with routing
│   ├── App.css                           # Component styles
│   ├── index.css                         # Global styles
│   │
│   ├── components/                        # Reusable UI components
│   │   ├── BookForm.tsx                    # Shared form (create/edit)
│   │   ├── BookList.tsx                     # Book listing component
│   │   └── BookDetail.tsx                    # Book details display
│   │
│   ├── pages/                              # Page components
│   │   ├── Home.tsx                          # Books table view
│   │   ├── CreateBook.tsx                     # New book form
│   │   ├── EditBook.tsx                        # Edit book form
│   │   └── ViewBook.tsx                        # Book details page
│   │
│   ├── services/                            # API communication
│   │   └── bookService.ts                     # Book API wrapper
│   │
│   └── types/                               # TypeScript definitions
│       └── Book.ts                            # Book interfaces
```

5. Validation Strategy

5.1 Validation Rules Comparison

Field	Frontend Rule	Backend Rule	UI Feedback
Title	Required, ≤200 chars	[Required], [MaxLength(200)]	Error message
Author	Required, ≤100 chars	[Required], [MaxLength(100)]	Error message
Description	≤1000 chars	[MaxLength(1000)]	Error message

6. Development Environment and Configuration

6.1 Prerequisites

Backend Requirements:

- .NET 8.0 SDK
- Visual Studio 2022 (or VS Code with C# extensions)
- SQLite (included via Microsoft.EntityFrameworkCore.Sqlite)

Frontend Requirements:

- Node.js 18+ and npm
- Modern web browser (Chrome, Firefox, Edge)

6.2 Running the Application

Start Backend (Visual Studio):

1. Open LibraryManagementAPI.sln
2. Press F5 to run with debugging
3. Swagger UI opens at https://localhost:7058/swagger

Start Frontend (Terminal):

```
cd library-frontend
npm install
npm run dev
# Frontend runs at http://localhost:5173
```

7. Key Insights and Reflections

7.1 Architecture Lessons

Interface-Based Design Triumphs

- Implementing IBookService demonstrated the power of programming to interfaces
- Enabled loose coupling between Controller and Service layers
- Would make unit testing significantly easier with mocking

7.2 Full-Stack Development Insights

Type Safety Across the Stack

- Using TypeScript to mirror C# DTOs eliminated countless "undefined" errors
- Shared understanding of data shapes between teams
- Made refactoring safer and more predictable

Mirror Validation Strategy

- Frontend validation provides instant feedback (good UX)
- Backend validation ensures security (defense in depth)
- Both layers use same rules, creating a safety net

8. Conclusion

8.1 Project Summary

The Library Management System successfully demonstrates a professional, production-ready approach to full-stack web development. By combining Clean Architecture principles on the backend with Component-Based Architecture on the frontend, I've created a system that is:

- **Maintainable**: Clear separation of concerns makes updates and bug fixes straightforward
- **Scalable**: Architecture can accommodate additional features without refactoring
- **Testable**: Interface-based design enables comprehensive unit testing
- **User-Friendly**: Responsive interface with immediate validation feedback

8.2 Future Enhancements

While the current implementation meets all requirements, potential future improvements include:

- Authentication & Authorization: User login and role-based access
- Pagination: For large book collections
- Search & Filter: Advanced book discovery features
- Unit Tests: Comprehensive testing suite for both frontend and backend
- Dockerization: Containerized deployment

- Genre Categories: Additional metadata for books

8.3 Final Thoughts

This project represents a solid foundation for any data-driven web application. The architectural patterns demonstrated here—Clean Architecture, Service Layer, DTO Pattern, Component-Based UI—are industry best practices that translate directly to larger, more complex systems. The emphasis on clean code, separation of concerns, and type safety ensures that this system can evolve and scale while remaining maintainable.

The successful integration between a well-structured .NET backend and a responsive React frontend proves that full-stack development doesn't have to be chaotic. With proper planning and adherence to architectural principles, we can build systems that are both powerful and pleasant to work with.