1. Library Management System

1.1 Software Engineering Internship Assignment

Student Name: Nehan Pavindya Vidanaarchchi

Date: November 30, 2025

Company: Expernetic LLC

2. Table of Contents

3. Introduction

This report provides a comprehensive overview of the design, development, and implementation of the Library Management System, a full-stack web application created as part of the Software Engineering Internship assignment at Expernetic LLC. The document details the project's architecture, technology stack, challenges encountered, and key learnings, serving as a formal record of the work completed.

3.2 Purpose of the System

The primary purpose of the Library Management System is to provide a simple yet robust digital platform for managing a collection of books. The system is designed to support fundamental administrative tasks, including adding new books to the inventory, viewing the complete list of available books, updating the details of existing books, and removing books from the collection. It serves as a foundational application demonstrating the core principles of full-stack development, from database interaction to user interface design.

3.3 Assignment Objectives

The internship assignment was structured to achieve several key objectives:

- **Demonstrate Full-Stack Proficiency:** To build a complete, end-to-end application using a modern technology stack, showcasing competence in both backend and frontend development.
- **Implement RESTful Services:** To design and develop a clean, well-structured REST API that adheres to industry best practices for web services.
- **Master Data Persistence:** To effectively use an Object-Relational Mapper (ORM) for database schema management, migrations, and data

manipulation.

- **Develop a Modern Frontend:** To create a responsive and interactive single-page application (SPA) using a component-based framework and ensuring type safety.
- **Practice Professional Documentation:** To produce a high-quality technical report that clearly documents the project's architecture, design decisions, and implementation details.

3.4 Technology Stack Summary

The project leverages a carefully selected stack of modern technologies to ensure efficiency, scalability, and maintainability. The backend is powered by the **.NET 8 Web API** framework, providing a high-performance foundation for RESTful services. Data persistence is managed by **Entity Framework Core** with an **SQLite** database, chosen for its simplicity and ease of setup. The frontend is a dynamic single-page application built with **React** and **TypeScript**, which offers a robust, type-safe environment for building user interfaces.

4. Technologies Used

The selection of technologies for this project was guided by the requirements of building a modern, maintainable, and efficient full-stack application. The following table details the specific technologies, tools, and libraries used across the different tiers of the system.

Category	Technology/Tool	Description
Backend	.NET 8 Web API	The core framework for building the RESTful API. Chosen for its high performance, cross-platform capabilities, and robust feature set for web development.
Backend	C#	The primary programming language for the .NET backend. A modern, object-oriented, and type-safe language.
Backend	Entity Framework Core 8	A modern object-relational mapper (O/RM) for .NET. Used to abstract database interactions, manage the database schema via code-first migrations, and perform CRUD operations.
Database	SQLite	A self-contained, serverless, file-based SQL database engine. Selected for its lightweight nature and ease of setup, making it ideal for development and small-scale applications.
Frontend	React	A popular JavaScript library for building user interfaces. Its component-based architecture and

Category	Technology/Tool	Description
		virtual DOM facilitate the creation of interactive and reusable UI elements.
Frontend	TypeScript	A statically typed superset of JavaScript. Used to add type safety to the frontend codebase, reducing runtime errors and improving developer experience and code maintainability.
Frontend	React Router	A standard library for routing in React applications. Used to enable navigation between different pages (components) in the single-page application.
Styling	Custom CSS	Standard Cascading Style Sheets were used for styling. This approach was chosen to maintain simplicity and focus on the core application logic without introducing dependencies on large UI frameworks.
Tools	Visual Studio Code	The primary code editor for both frontend and backend development, offering excellent support for C#, .NET, TypeScript, and React.
Tools	Node.js / npm	The runtime environment for the React development server and the package manager for handling frontend dependencies.
Tools	Git	The distributed version control system used for source code management, tracking changes, and collaborating on the codebase.

Category	Technology/Tool	Description
Tools	Swagger (Swashbuckle)	A tool integrated into the .NET API to automatically generate interactive API documentation. It was crucial for testing and visualizing the API endpoints.
Tools	EF Core Migrations	A command-line tool used to generate and apply database schema changes based on the C# data models, ensuring the database schema stays in sync with the application code.

5. Backend Architecture

The backend of the Library Management System is a RESTful API built using .NET 8. It follows a clean, layered architecture designed for separation of concerns, maintainability, and testability. The architecture is centered around the standard ASP.NET Core Web API patterns.

The architecture is composed of three primary layers. The **Presentation Layer**, consisting of the API Controllers (e.g., `BooksController`), serves as the entry point for all incoming HTTP requests. It is responsible for request validation and routing. The **Data Access Layer**, implemented using Entity Framework Core and the `ApplicationDbContext`, handles all interactions with the SQLite database, abstracting the data persistence logic. For this project's scope, the business logic is contained directly within the controllers, which mediate between the presentation and data access layers to fulfill client requests.

5.5 REST API Design Philosophy

The API adheres to REST principles by using standard HTTP methods for operations and resource-based URLs.

- **Resources:** The primary resource is the \`Book\`. All operations are performed on this resource or a collection of it.
- **URLs:** The base URL for the resource is \`/api/Books\`. A specific book is identified by its ID, e.g., \`/api/Books/{id}\`.
- **HTTP Methods:** Standard methods are mapped to CRUD (Create, Read, Update, Delete) operations:
    - \`GET\`: Retrieve a list of books or a single book.
    - \`POST\`: Create a new book.
    - \`PUT\`: Update an existing book.
    - \`DELETE\`: Remove a book.
- **Statelessness:** Each request from a client contains all the information needed to understand and process the request. The server does not store any client context between requests.

5.6 Core Components

5.6.1 Controllers

The `BooksController` is the entry point for all HTTP requests related to books. It is decorated with the `[ApiController]` attribute, which enables standard API behaviors like automatic model validation and problem details for error responses. The `[Route("api/[controller]")]` attribute defines the base route for all actions within the controller. Each action method is mapped to an HTTP verb and handles the business logic for that request, often by interacting with the `DbContext`.

5.6.2 Models

The `Book` model is a Plain Old C# Object (POCO) that represents the data structure of a book. This class is used by Entity Framework Core to define the database table schema and by the API controllers to define the shape of request and response data.

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public string Description { get; set; }
}
```

5.6.3 DbContext

The `ApplicationDbContext` class inherits from `Microsoft.EntityFrameworkCore.DbContext`. It acts as the bridge between the domain models (like `Book`) and the database. It contains a `DbSet` property, which represents the collection of all books in the database and allows for querying and saving data. The database connection string (pointing to the SQLite file) is configured in `Program.cs` and injected into the `DbContext`.

5.6.4 Program.cs Configuration

In .NET 8, the `Program.cs` file is the central place for service registration and middleware configuration. Key configurations for this project include:

- **CORS (Cross-Origin Resource Sharing):** A policy is configured using `builder.Services.AddCors()` and `app.UseCors()` to allow the React frontend (running on `http://localhost:3000`) to make requests to the backend API (running on a different port). This is critical for development.
- **Database Context:** The `ApplicationDbContext` is registered with the dependency injection container using `builder.Services.AddDbContext()`, specifying the use of SQLite.
- **Swagger/OpenAPI:** Services for Swagger generation (`AddSwaggerGen`) and the Swagger UI (`UseSwagger`, `UseSwaggerUI`) are configured to provide interactive API documentation, which is invaluable for testing and development.

5.7 Folder Structure

The backend project maintains a conventional folder structure for clarity and organization:

5.8 Rationale for SQLite

SQLite was chosen as the database for this project due to its simplicity and suitability for the assignment's scope. As a serverless, file-based database, it requires no separate database server installation or configuration. This significantly simplifies the development setup and makes the project self-contained. Its seamless integration with Entity Framework Core further streamlines development, making it an excellent choice for prototyping, local development, and small-scale applications where the overhead of a full-fledged database server like SQL Server or PostgreSQL is unnecessary.

6. Frontend Architecture

The frontend is a Single-Page Application (SPA) built with React and TypeScript. The architecture is designed around the principles of componentization, clear state management, and separation of concerns to create a maintainable and scalable user interface.

6.9 Component-Based Structure

The application follows React's core philosophy of building UIs from small, reusable pieces called components. The UI is broken down into a hierarchy of components, where parent components manage state and pass data down to child components as props. This approach promotes reusability and makes the UI easier to reason about and test.

6.10 Core Pages and Components

The application is structured into several main "page" components, each corresponding to a specific view or route.

- **`BookList.tsx`**: This is the main page of the application. It fetches and displays a list of all books from the API. Each book in the list has controls for editing or deleting it. This component also includes a link to navigate to the "Add Book" page. It manages the state of the book list using the `useState` and `useEffect` hooks.
- **`AddBook.tsx`**: This page contains a form for creating a new book. Upon submission, it sends a `POST` request to the backend API and navigates the user back to the book list, which then shows the newly added book.
- **`EditBook.tsx`**: This page contains a form for updating an existing book. It is pre-populated with the data of the book being edited, which is fetched from the API based on the book's ID from the URL parameter. On submission, it sends a `PUT` request to the API.

6.11 Routing Architecture

Client-side routing is managed by the `react-router-dom` library. In the main `App.tsx` component, `BrowserRouter`, `Routes`, and `Route` are used to define the navigation structure. This allows the application to have distinct URLs for different views without requiring a full page reload from the server.

```
// Example routing setup in App.tsx
<BrowserRouter>
  <Routes>
    <Route path="/" element={<BookList />} />
    <Route path="/add" element={<AddBook />} />
    <Route path="/edit/:id" element={<EditBook />} />
  </Routes>
</BrowserRouter>
```

6.12 TypeScript Integration

TypeScript is used throughout the frontend codebase to enforce type safety. This is particularly beneficial for defining the shape of data structures and component props. An `IBook` interface is defined to match the structure of the `Book` model from the backend, ensuring that data flowing from the API is handled consistently and preventing common type-related bugs.

```
export interface IBook {
    id: number;
    title: string;
    author: string;
    description: string;
}
```

6.13 API Service Layer

To promote separation of concerns and avoid scattering `fetch` calls throughout components, a dedicated API service layer was created (e.g., `src/services/apiService.ts`). This file centralizes all communication with the backend API. It exports functions for each endpoint (e.g., `getAllBooks`, `createBook`, `deleteBook`). Components call these service functions instead of directly making HTTP requests, making the code cleaner, easier to test, and simpler to update if the API changes.

7. Database Design

The database is the persistence layer of the application, responsible for storing and managing the book data. The design is intentionally simple for this project, focusing on a single entity and leveraging Entity Framework Core for schema management.

7.14 Rationale for SQLite

As mentioned previously, SQLite was the database of choice. Its primary advantages in this context are its zero-configuration nature and file-based storage. This eliminates the need for a separate database server, simplifying both the development environment and potential deployment. For an application with a single data model and low concurrency requirements, SQLite provides all the necessary functionality without the complexity of larger database systems.

7.15 EF Core Migrations

The database schema is managed using the "code-first" approach with EF Core Migrations. This workflow involves:

1. Defining the data model as a C# class (`Book.cs`).
2. Defining the database context (`ApplicationDbContext.cs`).
3. Running the command `dotnet ef migrations add ` (e.g., `InitialCreate`). This command inspects the models and generates C# code that describes the necessary schema changes.
4. Running the command `dotnet ef database update`. This command applies the pending migrations to the database, creating or updating the tables accordingly.

This process ensures that the database schema is always in sync with the application's data models and provides a version-controlled history of all schema changes.

7.16 Database Schema

The database contains a single table, `Books`, which directly maps to the `Book` model in the C# code.

Books Table Definition

Column Name	Data Type	Constraints	Description
Id	INTEGER	PRIMARY KEY, AUTOINCREMENT	The unique identifier for each book record.
Title	TEXT	NOT NULL	The title of the book.
Author	TEXT	NOT NULL	The author of the book.
Description	TEXT	NULLABLE	A brief description or summary of the book.