Thilina Chandrasekara

Library CRUD Project – Development Report

1. Introduction

This report summarizes the development of a small full-stack Library CRUD application

Context & Learning Journey. I had not built a .NET project before. I completed this work by actively learning, reading documentation, and searching for solutions while building. I kept the scope to focus on the fundamentals that clean CRUD, readable code, and a reliable local run setup.

2. Scope & Features

- Create, Read, Update, Delete (CRUD) books: title, author, description.
- Persistence using SQLite via Entity Framework Core.
- Minimal React UI with a list page and a single form used for both create and edit.
- TypeScript types for the Book shape in the frontend.
- Local development on fixed ports (Frontend 5173, Backend 5228).

3. Architecture Overview

This is the architecture:

• Frontend (React + TypeScript + Vite) calls the API using Axios.
• Backend (ASP.NET Core .NET 9 Web API) exposes REST endpoints under /api/Books.
• Data is stored in a local SQLite file (library.db) through EF Core.

4. Backend Implementation

• Frameworks: .NET 9 Web API, EF Core 9, SQLite provider.

• Model: Book { Id, Title, Author, Description } with simple validation (Title and Author required).

• DbContext: AppDbContext with DbSet<Book> to map the Books table.

• Controller: BooksController with endpoints: GET all, GET by id, POST, PUT, DELETE;

• Configuration: Connection string in appsettings.json; dependency injection wires EF Core in Program.cs.

5. Frontend Implementation

• Stack: Vite + React + TypeScript for a fast and type-safe DX.

• API client: src/api/books.ts centralizes Axios calls (baseURL points to http://localhost:5228/api).

• Pages:

- BooksList.tsx: loads and lists books; supports Delete with an optimistic UI update.

- BookForm.tsx: used for both create and edit; shows basic client validation and error messages.

6. Development Process

1) Designed the minimal data model (Book).

2) Created the Web API project, added EF Core + SQLite packages, and configured the connection string.

3) Built AppDbContext and the BooksController with async CRUD.

4) Ran EF Core migrations to create the database (library.db).

5) Scaffolded the React app, installed axios and react-router-dom.

6) Implemented a small Axios client and the two simple pages (list + form).

7) Verified end-to-end CRUD locally on fixed ports.

Since I was new to .NET, I learned as I progressed—reading docs and searching for solutions to issues like EF CLI setup, migrations, and port/CORS alignment.

7. How to Run Locally (Two Terminals)

Backend (Terminal 1):

```
  cd D:\library-ms\backend

  dotnet run
```

Frontend (Terminal 2):

```
  cd D:\library-ms\frontend

  npm run dev
```

Frontend runs at http://localhost:5173 and calls the API at http://localhost:5228/api.

8. Challenges & How I Solved Them

• EF CLI not found (dotnet-ef): installed it as a local tool using a tool manifest; ran migrations afterward.

• Build failed during migrations: fixed missing packages and Program.cs wiring; ensured net9.0 and EF9 versions.

• Network Error from frontend: caused by port mismatch or CORS; standardized on HTTP ports (5173 UI, 5228 API) and aligned baseURL.

9. Testing

• Manual API verification: GET/POST/PUT/DELETE tested via the frontend UI

• UI checks: Created a book, edited it, deleted it, and refreshed to confirm persistence in the SQLite DB.

10. Key Insights & Reflection

• Proper HTTP status codes matter for predictable UI behavior.

• Fixing ports and avoiding unnecessary complexity makes local dev and demos reliable.

• Even without prior .NET experience, a focused, incremental approach works well—build the smallest viable slice, verify, then extend.

11. Conclusion

This project is intentionally small but complete. It shows a clean CRUD flow across a React frontend and an ASP.NET Core backend with EF Core and SQLite. I built it while learning .NET, documenting the steps and decisions , and so it's easy to run on any machine with two simple commands.