1. Introduction

Introduce the project and its purpose:

The Library Management System is designed to allow users to manage books in a library, offering CRUD (Create, Read, Update, Delete) functionality.
Technologies used:
Backend: .NET 8, C#, SQLite, Entity Framework Core
Frontend: React, TypeScript, TailwindCSS
Mention the development timeline (e.g., "Completed within 7 days").

2. Project Overview

Provide a summary of the system:

Motivation: Explain why such a system is important (e.g., efficient book management).
Goals:
Build a functional backend API for book management.
Design a responsive web interface for users to interact with the system.
Scope: CRUD operations for books; optional user authentication (if included).

3. Backend Implementation

Detail the development of the backend:

Technology Stack:
Language and Frameworks: C#, .NET 8
Database: SQLite
ORM: Entity Framework Core
Architecture:
Controller-Repository-Service Pattern (Optional if you implemented this structure).
`BooksController`: Handles HTTP requests.
`AppDbContext`: Manages communication with the SQLite database.
Endpoints:

Method	Route	Description
GET	`/api/books`	Get all books
GET	`/api/books/{id}`	Get a single book
POST	`/api/books`	Add a new book
PUT	`/api/books/{id}`	Update an existing book
DELETE	`/api/books/{id}`	Delete a book

Error Handling:
Validation for missing or invalid data using ModelState.
Returning proper HTTP status codes (e.g., `404 Not Found`, `400 Bad Request`).

Database Schema: The `Book` entity with the fields:
`Id`: Primary key
`Title`: Book title
`Author`: Author's name
`Description`: Short description of the book
Include code snippets for major components:
C#

```
namespace LibraryAPI.Models
{
    public class Book
    {
        public int Id { get; set; } // Primary Key

        [Required, StringLength(100)]
        public string Title { get; set; } // Title cannot exceed 100 characters

        [Required, StringLength(100)]
        public string Author { get; set; } // Author cannot exceed 100 characters

        [StringLength(500)]
        public string Description { get; set; } // Optional, max 500 characters
    }
}
```

4. Frontend Implementation

Detail the frontend design and development:
Technology Stack:
React with TypeScript
TailwindCSS for the design
Axios for API calls
Architecture:
Component-based structure with reusable components:
`ModalForm`: Reusable modal for both adding and editing books.
`BookList`: The main page that displays book records in a table.
State Management:
React's `useState` and `useEffect` hooks to manage local state.
Features:
Add Book: Opens a modal to add a book (sends `POST` request to backend API).
View Books: Retrieves and displays book records in a table.
Edit Book: Opens a modal pre-filled with the book's details and sends a `PUT` request.
Delete Book: Deletes a book from the list with a `DELETE` request.
Design:
The interface is fully responsive and styled using TailwindCSS.
Implemented validation for form inputs in the modal.
Include frontend snippets where necessary, like the `ModalForm.tsx` or `BookList.tsx`.

5. Challenges Faced

Be honest about the problems you encountered and how you solved them. Examples:
Challenge: Cascading Renders in React
Resolution: Used `useCallback` to stabilize functions and reduce unnecessary renders.
Challenge: SQLite Migrations
Resolution: Studied `dotnet ef` commands and ensured proper migration setup.

Challenge: Managing Cross-Origin Resource Sharing (CORS) Errors
Resolution: Enabled CORS middleware in `Program.cs` to allow frontend-backend communication.

6. Key Insights

Reflect on what you learned during the project:
How did you improve your skills in integrating and debugging a full-stack project?
What were the design or performance trade-offs you considered?
Which aspects of .NET, React, or TypeScript were most important to mastering this build?

7. Conclusion

Summarize the project:
Completion Status:
The application implements all CRUD operations and aligns with the project scope.
The frontend offers a clean, responsive UI with proper error handling and validation.
Future Enhancements:
Add user authentication.
Deploy the backend and frontend to a live server for public use.