# Expernetic Library Management System

Full-Stack Internship – Project Report

**Reezan Saleem**

**June 2025**

# Table of Contents

# 1. Project Overview

The Library Management System was developed as part of the Expernetic Software Engineering Full-Stack Internship assignment. The goal of the project was to design and implement a full-stack web application that allows users to efficiently manage a collection of books through a user-friendly interface and robust backend API.

The application provides full CRUD functionality—allowing users to create, read, update, and delete book records—with a focus on usability, data integrity, and secure access.

The tech stack includes:

- A .NET 9 backend using Entity Framework Core with a SQLite database, providing a lightweight yet powerful server-side API.
- A modern React frontend developed in TypeScript, offering a responsive, component-driven user interface with strong typing and seamless integration with the backend.

This project demonstrates the integration of frontend and backend technologies to deliver a cohesive application with real-world functionality.

# 2. Development Process

## 2.1. Backend Implementation

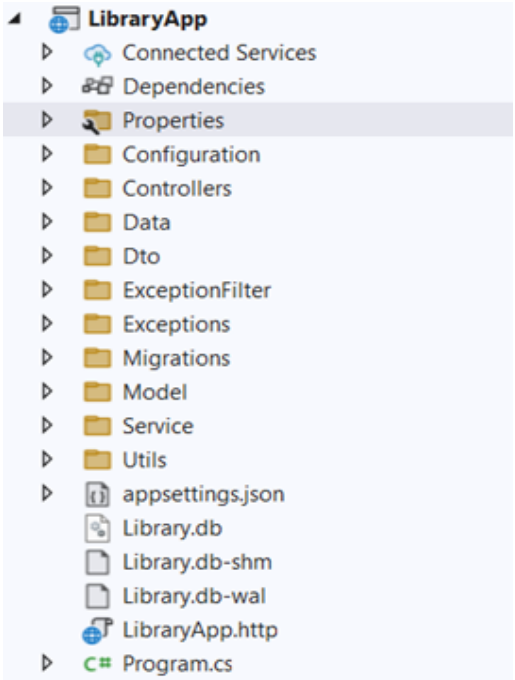
The backend for the Library Management System was built using .NET 9 Web API with Entity Framework Core and follows a code-first approach for database design using SQLite.

**Architecture and Folder Structure**

The project follows a modular architecture to maintain separation of concerns and improve maintainability. The folder structure includes:

- Controllers – Handles HTTP requests and routes them to the appropriate services.
- Service – Contains business logic for book management and user authentication.
- Model – Defines data models such as Book and User, with validation using data annotations like [Required] and [StringLength].
- Dto – Used for request/response payload shaping.
- Data – Sets up the DbContext for Entity Framework.
- Migrations – Maintains schema version history for the SQLite database.
- Configuration, Utils, Exceptions, ExceptionFilter – For settings, helper logic, and consistent error handling.

The backend folder structure is illustrated below,



**API Design**

The API supports all major CRUD operations:

- GET – Fetch book(s)
- POST – Create a new book
- PATCH – Partially update a book (preferred for updating selected fields)
- DELETE – Remove a book

Additionally, a mass edit endpoint was implemented to allow updates to multiple books in a single request—complementing the existing single-book update functionality and significantly enhancing productivity for bulk operations.

**Authentication & JWT Token Generation**

User authentication was implemented using JWT (JSON Web Tokens) to secure protected API endpoints and authorize access to library resources.

1. Key Details of the Authentication Flow:
   - The AuthController handles user registration and login.
   - Upon successful login, a JWT is generated and returned to the client for use in subsequent requests.
   - All book-related endpoints are secured using [Authorize] and require a valid token.
2. Token Generation Strategy:

   The token is generated using a custom TokenService class, which encapsulates the logic for secure token creation. Key aspects of the strategy include:

   - Symmetric key signing is used via HMAC SHA256, chosen for its simplicity, performance, and suitability for single-server scenarios.
   - The username is included as a claim (ClaimTypes.Name) and forms part of the token payload to identify the authenticated user.
   - The token has a configurable expiry (set to 1 hour), meaning it becomes invalid one hour after issuance to reduce the risk of misuse.
   - Token generation is implemented using the System.IdentityModel.Tokens.Jwt and Microsoft.IdentityModel.Tokens packages, which provide cryptographic utilities, signing algorithms, and token serialization support.

The resulting token is a compact, URL-safe string that is returned to the frontend and used as a Bearer token in the Authorization header for protected API requests to access the book controller.

### Model Design with Validation

Models like Book are decorated with validation attributes to enforce data integrity. Example:

```
namespace LibraryApp.Model
{
    39 references
    public class Book
    {
        10 references
        public int Id { get; set; }

        [Required(ErrorMessage = "Title is required.")]
        [StringLength(100, MinimumLength = 2, ErrorMessage = "Title must be between 2 and 100 characters.")]
        10 references
        public string Title { get; set; } = string.Empty;

        [Required(ErrorMessage = "Author is required.")]
        [StringLength(200, MinimumLength = 2, ErrorMessage = "Author must be between 2 and 200 characters.")]
        10 references
        public string Author { get; set; } = string.Empty;

        [Required(ErrorMessage = "Description is required.")]
        9 references
        public string Description { get; set; } = string.Empty;
        4 references
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

        5 references
        public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;

        4 references
        public Book() { }

        2 references
        public Book(BookDto bookDto)
        {
```

These constraints are reflected in the SQLite schema through the Entity Framework's code-first migration system.

### Exception Handling

Custom exceptions (e.g., NotFoundException, BadRequestException) are defined to represent specific error scenarios within the system. A global exception filter (GlobalExceptionFilter) is implemented to centrally handle these and other unhandled exceptions, ensuring they are consistently translated into meaningful HTTP responses for the client.

## 2.2. Frontend Implementation Process

**Frontend Implementation**

The frontend of the Library Management System was developed using **React** with **TypeScript**, chosen for its strong typing capabilities, component-driven architecture, and compatibility with modern web tooling. The user interface is designed to be intuitive, responsive, and user-friendly.

1. **Technology Stack**
   - React – Used to build modular, reusable, and interactive UI components for a seamless user experience.
   - TypeScript – Provides static typing to catch errors early and improve code clarity and maintainability.
   - RSuite (React Suite) – A UI component library used for consistent styling and professionally designed elements such as tables, buttons, inputs, and toasts.
   - React Router – Handles navigation between routes such as login, register, and the main book management interface.
   - Axios – Handles API requests to the backend for operations related to user authentication and book data (create, read, update, delete).
   - Context API – Manages global state across the application for both book data and authenticated user sessions, avoiding prop drilling and enabling centralized updates.

2. **Key Features & Functionality**

   - **Book CRUD Operations**
     Users can create, view, update, and delete book records directly from the UI:
     - **Create**: Form input to add a new book with validation for title, author, and description.
     - **Read**: Books are listed in a responsive table with expandable rows.
     - **Update**: Inline editing is enabled in the table via editable cells with validation.
     - **Delete**: Records can be deleted with a confirmation prompt to prevent accidental removal.

- **Mass Edit Functionality**
  The application includes a Mass Edit feature for efficient bulk updates. Users can select multiple book records using checkboxes and edit fields directly within the table. After making changes, clicking the Mass Edit button applies the updates to all selected entries at once. This feature significantly reduces the effort required to update records individually and enhances the user experience during large-scale edits.

- **Inline Editing**

  For displaying book records, the application uses editable tables provided by RSuite. Each record can be edited inline within the table itself, eliminating the need to navigate to a separate page or open a modal popup. This design choice enhances usability by providing a single-page, uninterrupted workflow for viewing, editing, and deleting records.

- **User Authentication**
  A lightweight login and registration system is implemented with the following features:

  - Credential Validation: User credentials are validated securely on the backend.
  - JWT-Based Authentication: On successful login, a JWT token is issued and securely stored in the client's localstorage.
  - Centralized Auth Management: The application uses a centralized AuthContext to manage authentication state. This context handles token storage, updates, and removal during login, registration, and logout events, ensuring consistent access control across the app.
  - Secure API Access: The JWT token is automatically attached to all protected API requests via the `Authorization` header using a custom interceptor or header injection.

- **Error Handling & Toasts**
  Alert toasts and inline feedback messages are shown for success/failure cases:

  - Book added/edited/deleted confirmations.
  - User Login/Logout.
  - Validation errors on form input for book creation, update and mass edit.
  - Authentication errors (e.g., invalid credentials)

- **Responsive & Accessible Design**
    - The layout is mobile-friendly, supporting smaller screens without breaking UI components. This  is done through media queries for it.
    - Buttons and controls are clearly labeled and styled for visibility.
- **Security**

  The application implements multiple layers of security to ensure that only authorized users can access protected resources, and that all user inputs are validated to prevent misuse or injection attacks such as Cross-Site Scripting (XSS).

  Route Protection & Token Handling

  To prevent unauthorized access, the application uses a private route pattern within the React frontend:

    - Protected Routes: The Book Management page is accessible only if a valid JWT token is present. This route is wrapped inside a protected component in App.tsx. If the token is missing, the user is automatically redirected to the login page ("/").
    - Token Expiry Handling: Even if a token exists in localStorage, the application verifies its validity. Tokens are configured to expire after 1 hour. If the token is expired, the user is logged out automatically to prevent session misuse.
    - When fetching book data via BooksContext, if the backend responds with a 401 Unauthorized status (e.g., due to token expiration), the app triggers an automatic logout to enforce session integrity.

  Input Validation & XSS Prevention

  Extensive validation is implemented on the frontend to protect against malformed or malicious input and ensure clean, consistent data entry:

    - Field Requirements: Author, Title, and Description fields are required and cannot be left empty.
    - Format Restrictions: The Author field must not contain numbers or special characters, reducing the chance of invalid data or script injection.
    - Mass Edit Safeguards: The mass edit feature ensures that at least one change is made across selected rows before allowing a batch update.
    - Save Before Update: Newly added records must be explicitly saved before they can be selected for editing or mass updates.

- Accidental Deletion Prevention: Deletion actions are explicitly confirmed by the user to avoid unintentional data loss.

These client-side validations not only enhance usability but also serve as a crucial first line of defense against XSS attacks and other input-related vulnerabilities.

### 3. Component Structure

The application is built using modular and reusable components to ensure maintainability and scalability.

| COMPONENT | PURPOSE |
|---|---|
| BOOKTABLE | Displays all books on editable table |
| ACTIONCELL | Provides edit/save/delete controls within each table row |
| EDITABLECELL | Convert table cells into input fields during edit mode |
| BOOKCONTEXT | Manages global state for book data and handles API fetch logic |
| LOGINSIGNUP | Handles user authentication (login and registration) |
| AUTHCONTEXT | Handle authentication state of the user |
| ALERTTOAST | Displays of contextual or global alert messages |

#### Global State Management with Context API

The application uses the **Context API** to manage the global state across three core areas:

- **Books:** BookContext manages book data. Upon successful login, it fetches book records from the backend API. This design ensures the UI reflects the loading states immediately improving responsiveness even if database operations take time.
- **Alerts (Toast):** A centralized AlertContext controls the visibility and content of alerts. An AlertToast component is mounted in App.tsx and listens for context updates. After being triggered (e.g., on book update or login), the toast appears for 3 seconds and then auto-hides, providing a consistent and user-friendly notification experience.
- **Authentication:** The AuthContext handles token storage and cleanup. It ensures authentication state is properly maintained or reset across the app, syncing with localStorage for persistence.

**Additional UX Features**

- **Confirmation Modal:** A confirmation dialog is displayed before permanently deleting a book. Since the system doesn't currently support undo or recovery, this modal acts as a safeguard against accidental deletions.

**4. API Integration**

API requests are made using the axios library.

- The JWT token is included in the Authorization header for all protected endpoints.
- Responses are parsed and error messages are surfaced to the user using the alert system if necessary.

# 3. Steps to Run the Project

Follow the steps below to run both the backend API and the frontend application locally.

**1. Clone the repository**

git clone https://github.com/<your-username>/Expernetic-Library-Application.git

cd Expernetic-Library-Application

**2. Run the backend (.NET)**

1. Open the **Backend** folder (inside Expernetic-Library-Application) in Visual Studio 2022 or a compatible IDE.
2. Select the *HTTPS* launch profile and start the project ( **F5** ).
3. The server will start at **https://localhost:7137** as defined in *launchSettings.json*.
4. A pre-configured SQLite database file (Library.db) is already associated, so existing records will remain intact.

**3. Run the frontend (React)**

cd Expernetic-Library-Application/Frontend

npm install  # first–time only

npm start

The React app will open automatically in your browser

**4. Create a user account**

After the frontend loads, register a new user via the UI. Logging in enables access to the library's books and operations.

**5. Explore the API (Scalar)**

Additionally, with the backend running, open https://localhost:7137/Scalar/ in your browser. Scalar provides an interactive UI where you can explore and test all available API endpoints.

# 4. Challenges Faced

Although I have prior experience working with Java (Spring Boot) and Node.js, developing APIs with .NET Core was relatively new to me. This project required me to become familiar with C# syntax, the .NET Core Web API architecture, and tools like Entity Framework Core for database operations. I spent time reading the documentation and exploring online resources to understand best practices for building APIs in the .NET ecosystem.

On the frontend, while I have several years of experience with React and React Native, working with TypeScript in React was a new area. I had to adjust to TypeScript's strict typing system and understand how it integrates with React components, hooks, and context APIs. To bridge the gap, I referred to official TypeScript documentation and watched targeted video tutorials before beginning the frontend implementation.

Both of these learning curves enhanced my understanding of typed programming paradigms and framework-specific tooling, ultimately improving my ability to work across full-stack environments confidently.

# 5. Additional Features

Several optional but valuable features were implemented to enhance the overall functionality and user experience of the Library Management System.

1. Timestamp Tracking (CreatedAt & UpdatedAt)

   The backend automatically tracks both the creation date (CreatedAt) and the last modified date (UpdatedAt) for each book record. These timestamps are:

   - Stored in the database via the entity model.
   - Displayed in the frontend, providing users with visibility into when a record was created or last edited.

   This feature improves data transparency and auditability for all book-related actions.

2. Mass Edit Functionality

Beyond standard update capabilities, a mass edit feature was introduced to allow batch updating of multiple book records simultaneously. This is particularly useful for administrative tasks where the same change (e.g., correcting an author's name) must be applied to several entries at once.

3. Responsive User Validation with Feedback

The application incorporates extensive client-side validations, paired with alert toasts to guide user interaction. These alerts provide real-time, context-aware feedback for actions.

This combination of validation and immediate feedback ensures a smoother and more user-friendly experience while minimizing data entry errors.

# 6. Conclusion

The Expernetic Library Management System project served as a comprehensive exercise in full-stack application development, integrating both backend and frontend technologies to deliver a secure, functional, and user-friendly solution.

Through this project, I gained hands-on experience in:

- Developing and securing RESTful APIs using .NET 9, C#, and Entity Framework Core.
- Designing and implementing a modern, responsive frontend with React, TypeScript, and RSuite.
- Managing global application state with Context API and implementing JWT-based authentication for secure access control.
- Handling complex UI interactions like mass editing, inline editing, and real-time validation feedback.

While the project required overcoming a learning curve with both .NET and TypeScript, it significantly strengthened my understanding of full-stack engineering principles. The end result is a maintainable, scalable, and production-ready application that reflects real-world business requirements.

This internship assignment was not only a technical milestone but also a valuable learning experience in system architecture, secure coding practices, and user-centric design.