Library Management System

- **Project Repository**
- **Live Demo**

1.Overview

The Library Management System is a full-stack web application developed to efficiently manage and organize a digital book catalog. The system provides complete CRUD (Create, Read, Update, Delete) functionality, user authentication, category-based navigation, and a clean, professional user interface. Built using React with TypeScript for the frontend and ASP.NET Core Web API for the backend, the application follows industry-standard practices such as RESTful API architecture, secure password handling, and responsive UI/UX design. This project demonstrates practical experience in full-stack development, database integration, and modern software engineering workflows

2. Development Process

2.1 Backend Implementation (ASP.NET Core)

The backend was developed using ASP.NET Core following a clean and modular architecture.
Entity Framework Core with SQLite is used for database access and persistence.
Two main entities are implemented:

Books

Users.

The system exposes RESTful APIs through controllers:

- BooksController – Handles all book CRUD operations (create, retrieve, update, delete)
- AuthController – Handles user registration and login

Security is implemented using SHA-256 password hashing to ensure that passwords are never stored in plain text. Cross-Origin Resource Sharing (CORS) is enabled to allow communication between the frontend and backend.

2.2 Frontend Implementation (React + TypeScript)

The frontend is built using React with TypeScript to provide a modern, type-safe, and responsive user experience. Axios is used for API communication and React Router DOM is used for client-side routing.

Main features include:

- Login and Register pages with a professional UI
- Books dashboard for viewing all books
- Add, Edit, and Delete book forms
- Category-based book browsing
- Protected routes to restrict access for unauthenticated users

Reusable components such as BookList ,BookForm and Button are used to keep the code organized. Application state is managed using React hooks, and user login sessions are stored in localStorage.

2.3 Development Workflow

Git was used for version control with meaningful commit messages. Development was carried out in phases:

- Project setup
- Database and API implementation
- Frontend authentication pages
- Book CRUD functionality

3. Additional Features Implemented

Beyond the core CRUD functionality, several enhancements were implemented to improve usability and overall system quality.

Category-Based Navigation

- ❖ The system includes predefined book categories displayed as clickable cards on the home page. Selecting a category filters books accordingly, while a “View All Books” option allows access to the complete library. This enables intuitive and organized browsing.

Enhanced Book Detail View

- ❖ Each book has a dedicated detail page showing full information, including description and category. Dynamic routing is used to access books by ID, with proper loading and error handling.

Professional Authentication UI

- ❖ Login and registration pages feature a library-themed background, frosted-glass card design, and responsive layout, providing a modern and professional user experience.

Smart Description Truncation

- ❖ Book descriptions are shortened on list views to maintain a clean interface, with users able to view full details on the book detail page.

Real-Time UI Updates

- ❖ All CRUD operations reflect immediately without page reloads. Newly added, edited, or deleted books update dynamically using React state management.

Sample Data Seeding

- ❖ The database is populated with sample books across multiple categories to demonstrate realistic usage and presentation.

Responsive Design

- ❖ The application is fully responsive and adapts to mobile, tablet, and desktop screen sizes using modern CSS layouts.

Error Handling & User Feedback

- ❖ User-friendly error messages, loading indicators, and form validation improve reliability and usability on both frontend and backend.

Deployment-Ready Configuration

- ❖ The application is configured to support deployment with environment variables and production-ready settings.

4. Deployment

4.1 Deployment Architecture

The system follows a microservices architecture with separate hosting:

- Backend API: Hosted on Railway
- Frontend Application: Hosted on Vercel
- Database: SQLite with persistent storage on Railway
- Communication: Secure HTTPS

This setup allows independent scaling and efficient resource management.

4.2 Backend Deployment

- Platform: Railway
- Automatic GitHub integration
- Environment variable configuration
- CORS enabled for frontend access
- Automatic database table creation on startup

4.3 Frontend Deployment

- Platform: Vercel
- Built using Vite (TypeScript)
- Connected to Railway backend API

4.4 Database

- Technology: SQLite
- File Location: /app/data/library.db
- Data persists using Railway volume mounting

5. Challenges Faced & Solutions

5.1 Database Initialization in Cloud

Challenge: Tables (Books, Users) not created on Railway; migrations showed "up to date."
Solution: Replaced migrations with raw SQL CREATE TABLE IF NOT EXISTS on startup and seeded data.
Outcome: Tables reliably create on every deployment; pre-loaded test data available.

5.2 CORS Issues

Challenge: Frontend (Vercel) blocked from calling backend (Railway) due to CORS.
Solution: Configured backend CORS middleware to allow all origins, methods, and headers.
Outcome: Frontend communicates seamlessly with backend APIs.

5.3 HTTPS Redirect Conflicts

Challenge: Infinite redirect loops in production.
Solution: Enabled HTTPS redirect only in local development.
Outcome: App works via HTTPS in production without loops; local dev still enforces SSL.

5.4 Type Safety in API Communication

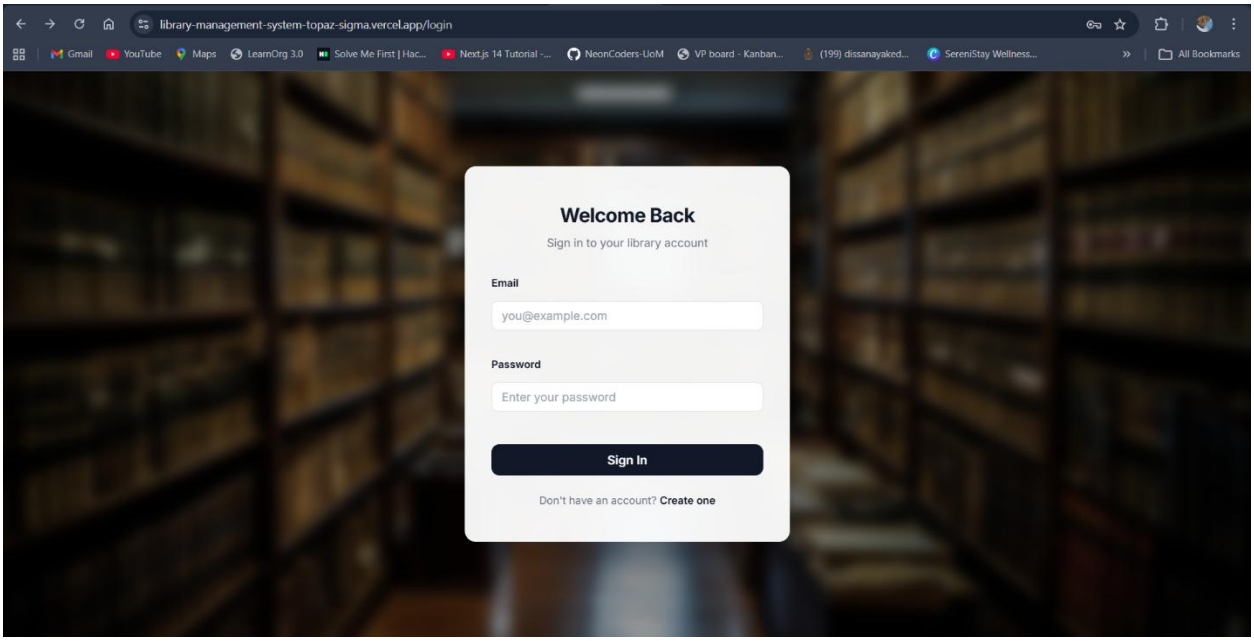
Challenge: API responses lacked types, causing runtime errors; no IDE autocomplete.
Solution: Created TypeScript interfaces for backend models and typed all API calls.
Outcome: Compile-time error detection, IDE autocomplete, fewer runtime issues.

5.5 State Synchronization After CRUD

Challenge: UI didn't update after CRUD operations; required manual refresh.
Solution: Implemented refresh trigger via state toggling between parent and child components.
Outcome: UI updates instantly after CRUD operations; smooth user experience.

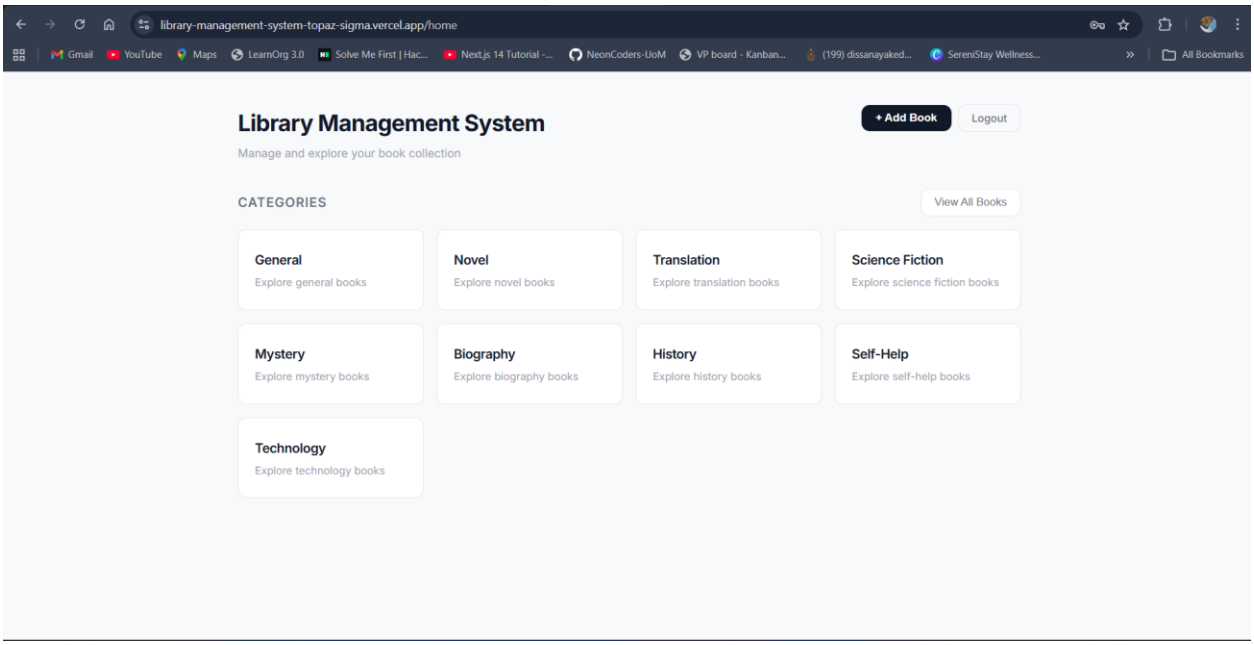
6. Figures

6.1 Login Page



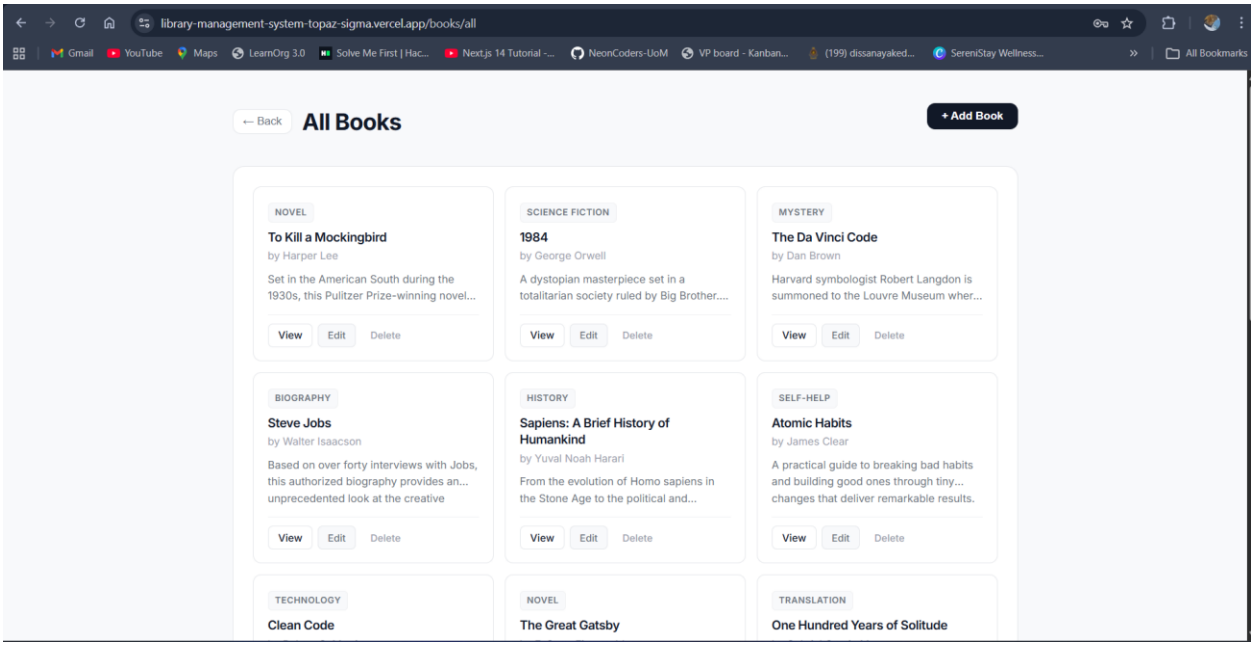
Elegant authentication interface with library-themed background and frosted glass card effect

6.2 Home Dashboard



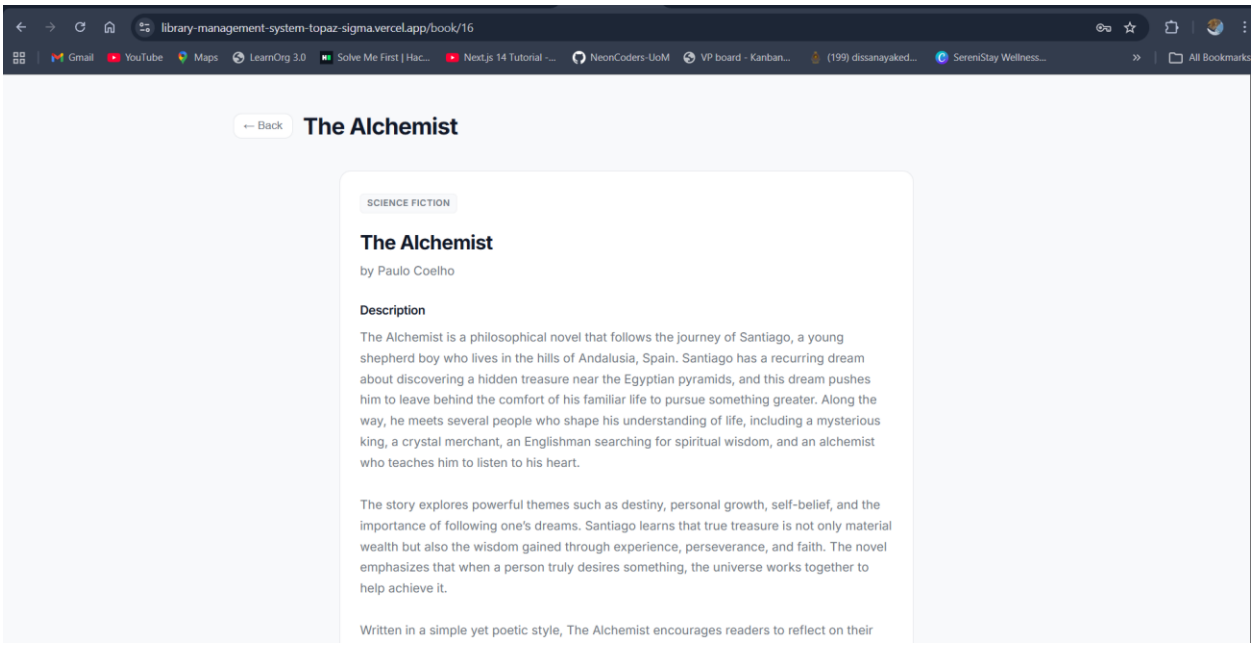
Category-based navigation with visual cards for intuitive book browsing

6.3 All Books View



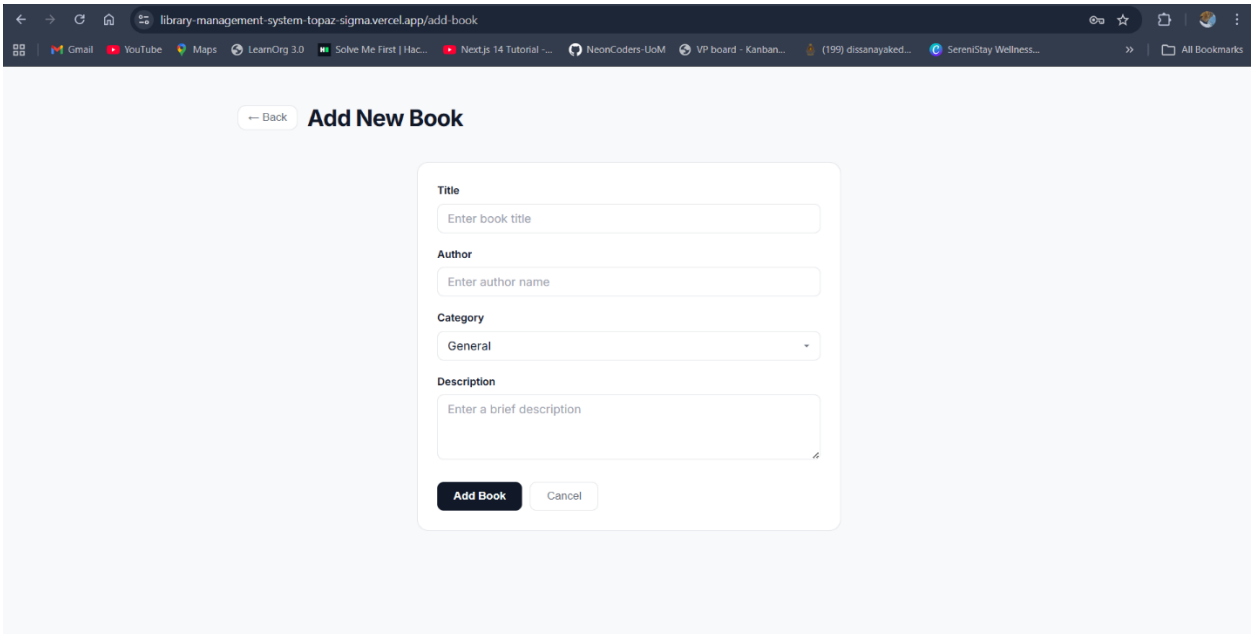
Complete book catalog with category badges, edit, and delete options

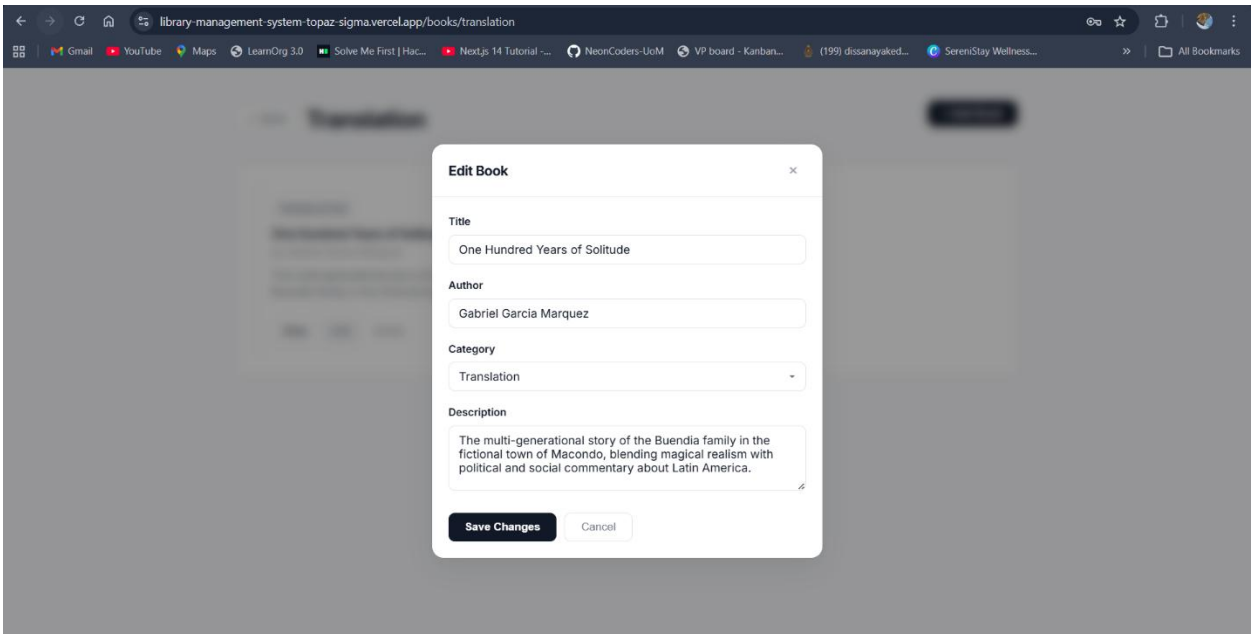
6.4 Book Detail Page



Full book information display with clean, readable layout

5.5 Add/Edit Book Form





7. Conclusion

The Library Management System successfully fulfills the assignment requirements by implementing a full-stack CRUD application with user authentication and a professional user interface. The project demonstrates practical knowledge of frontend and backend development, API integration, and database management.