Library Management System — Development Report

Project: Library Management System
Stack: ASP.NET Core 9 Web API (backend) + React 18 + TypeScript + Vite (frontend)
Date: February 2026

---

1. Executive Summary

This report summarizes the development of a full-stack **Library Management System** built with a C# .NET Web API backend and a React + TypeScript frontend. The system provides authenticated CRUD operations for book records, user registration and login via JWT, and a responsive web interface with search, filtering, pagination, CSV export, and light/dark theme support. The project includes automated CI (GitHub Actions), unit and integration tests for both backend and frontend, and a clear separation between domain, API, and UI layers.

---

2. Development Process

2.1 Approach

Development followed a **layered, API-first** approach:

- **Backend first:** Domain models (`LibraryManagement.Core`), database context and migrations, REST API with JWT authentication, then controller and service logic.
- **Frontend integration:** React app consuming the API with typed services, protected routes, and consistent error handling.
- **Quality:** Unit tests (password hashing, auth helpers, theme), API integration tests (Books CRUD with in-memory DB and test auth), and frontend component tests. CI runs on every push and pull request to `main` / `master` .

2.2 Project Structure

```
Library_Management_System/
├── backend/
│   ├── LibraryManagement.Api/       # Web API (controllers, DTOs, DbContext, middleware, service
│   ├── LibraryManagement.Core/      # Domain entities (Book, User)
│   ├── LibraryManagement.Tests/     # xUnit API and unit tests
│   └── LibraryManagement.sln
├── frontend/
│   ├── src/
│   │   ├── components/              # AppLayout, ProtectedRoute
│   │   ├── pages/                   # Book list, Create/Edit book, Auth, Profile, Settings, plac
│   │   ├── services/                # authService, bookService
│   │   ├── types/                   # auth, book
│   │   ├── theme.ts / theme.css     # Light/dark/system theme
│   │   └── test/                    # Vitest setup
│   ├── index.html, main.tsx, App.tsx
│   └── package.json, vite.config.ts
├── .github/workflows/ci.yml         # Backend + Frontend build and test
└── README.md
```

2.3 Tools and Technologies

Layer	Technology
Backend	.NET 9, ASP.NET Core Web API, Entity Framework Core 9, SQLite, JWT (Bearer), Swagger/OpenAPI
Frontend	React 18, TypeScript, Vite 5, React Router 6, CSS (no UI framework)
Auth	JWT (backend), PBKDF2 password hashing (HMACSHA256, 100k iterations), localStorage (frontend)
Testing	xUnit, Microsoft.AspNetCore.Mvc.Testing, EF Core InMemory (backend); Vitest, Testing Library (frontend)
CI	GitHub Actions (checkout, .NET 9, Node 20, restore/build/test for both projects)

---

3. Backend Implementation

3.1 Architecture

- **LibraryManagement.Core:** Domain entities only (`Book` , `User`) — no dependencies on API or EF.
- **LibraryManagement.Api:** Controllers, request/response DTOs, `LibraryDbContext` , migrations, middleware, and services (e.g. `PasswordHasher`). References Core.
- **LibraryManagement.Tests:** Uses `WebApplicationFactory` , in-memory database, and a test auth handler to run API tests without a real JWT.

3.2 API Design

- **Base URL:** `/api` (e.g. `http://localhost:5170/api`).
- **Auth:** `AuthController` at `/api/auth` — `POST register` , `POST login` ; both return `{ token, email }` . No refresh token; expiry configurable via `Jwt:ExpiryMinutes` (default 60).
- **Books:** `BooksController` at `/api/books` — full REST: `GET` (list), `GET {id}` , `POST` , `PUT {id}` , `DELETE {id}` . List supports optional query parameters: `search` (title/author/description), `author` (filter by author).
- All book endpoints require `Authorization: Bearer <token>` ; auth endpoints are `[AllowAnonymous]` .

3.3 Data Layer

DbContext: `LibraryDbContext` in `Data/LibraryDbContext.cs` — `DbSet<Book>` , `DbSet<User>` ; fluent configuration for keys, string lengths, and unique index on `User.Email` . **Migrations:** EF Core migrations under `Migrations/` ; applied automatically on startup (except when environment is `Testing`). Fallback SQL ensures `Books` and `Users` tables exist if migration history is out of sync. **Database:** SQLite; file path from `appsettings.json` (`Database:FileName` , default `library.db`).

3.4 Security and Validation

Passwords: `PasswordHasher` (PBKDF2-HMACSHA256, 100,000 iterations, 16-byte salt, 32-byte hash). Verification uses fixed-time comparison. **JWT:** Issuer, audience, and signing key from configuration; validation includes lifetime. **Validation:** Request DTOs use `[Required]` , `[StringLength]` , `[EmailAddress]` ; controllers return `400 Bad Request` with JSON `{ error }` .

3.5 Middleware and Error Handling

ExceptionHandlingMiddleware runs first. Maps `KeyNotFoundException` → 404, `ArgumentException` → 400, others → 500 (in Development, 500 can include exception message). Pipeline order: Exception handling → Swagger → CORS → Authentication → HTTPS redirection → Authorization → MapControllers.

3.6 Configuration

`appsettings.json` : Database file name, JWT (Key, Issuer, Audience, ExpiryMinutes), logging. CORS allows `http://localhost:5173` and `http://localhost:3000` . API runs at `http://localhost:5170` .

---

4. Frontend Implementation

4.1 Architecture

SPA with React Router; no global store — React local state and `localStorage` for auth and theme. API client: `authService` , `bookService` using `fetch` ; base URL from `VITE_API_URL` . Book requests send `Authorization: Bearer <token>` ; on 401 the app clears auth and redirects to `/login` .

4.2 Routing and Layout

Public: `/login` , `/register` . **Protected:** All other routes in `ProtectedRoute + AppLayout` . `ProtectedRoute` checks `isAuthenticated()` ; redirects to `/login` with `state.from` . **AppLayout:** Sidebar (LibManager, Dashboard, Books, Categories, Members, Transactions, Profile, Settings, user email, Log out) and `<Outlet />` . Index → Book list; `books/new` , `books/:id/edit` ; `categories` , `members` , `transactions` (placeholders); `profile` , `settings` . Catch-all → `/` .

4.3 Pages and Features

Page	Purpose
BookListPage	Dashboard: list with search/author filter (debounced), pagination (10/page), delete confirmation, CSV export, Recent Activity sidebar.
CreateBookPage	Add book form and list of existing books with update/delete.
EditBookPage	Load book by id, edit form, redirect to list.
LoginPage	Email/password; store token/email; redirect to / or state.from.
RegisterPage	Email, password, confirm (min 6); register then login.
ProfilePage	Email and "Librarian" role (read-only).
SettingsPage	Theme (light/dark/system) via theme.ts; placeholder Email notifications.
CategoriesPage, MembersPage, TransactionsPage	Placeholder "coming soon".

4.4 Theming

`theme.ts` : getStoredTheme, getResolvedTheme, setTheme, applyTheme (`data-theme` on documentElement); initTheme() and system preference listener. `theme.css` : overrides for `html[data-theme="dark"]` .

4.5 Build and Dev

Vite dev server port 5173; proxy `/api` to backend. Scripts: dev, build, test, test:run, lint, preview.

---

5. Challenges Faced

1. **Database and migrations:** Fallback SQL in Program.cs for missing tables; skipping migrations when Environment is Testing; in-memory provider for tests.
2. **Authentication in tests:** Test auth handler with fixed claims so protected endpoints can be tested without real JWT.

3. **Frontend–API contract:** Keeping TypeScript types aligned with API DTOs; consistent `{ error }` shape simplified handling.
4. **Session handling:** 401 → clear auth and redirect to login in both handleResponse and deleteBook.
5. **CI and environment:** Backend tests can fail if API is running (file lock); README advises stopping API; CI uses fresh runner.

---

6. Additional Features

- **Search and filtering:** Backend `search` and `author` params; frontend debounced search and collapsible filter.
- **Pagination:** Client-side, 10 per page.
- **Delete confirmation:** Two-step delete.
- **CSV export:** From dashboard.
- **Theme switching:** Light, dark, system with persistence.
- **Placeholder pages:** Categories, Members, Transactions.
- **Swagger UI:** /swagger.
- **Structured errors:** JSON `{ error }`; frontend displays them.
- **CI/CD:** GitHub Actions for backend and frontend (lint, build, test).

---

7. Key Insights and Reflection

1. **Separation of concerns:** Core domain without API/EF references improved clarity and testability.
2. **API-first and DTOs:** Dedicated request/response models improved validation, versioning, and Swagger.
3. **Testing strategy:** In-memory DB and test auth enabled full HTTP integration tests; frontend unit tests for auth and theme.
4. **User experience:** Debounced search, delete confirmation, CSV export, theme persistence.
5. **Operational robustness:** Migrate on startup and fallback SQL for missing tables.
6. **Security:** PBKDF2, fixed-time comparison, JWT validation, explicit AllowAnonymous/Authorize.

---

8. Conclusion

The Library Management System delivers a full-stack application with a clear backend (ASP.NET Core 9, EF Core, SQLite, JWT) and frontend (React 18, TypeScript, Vite). The development process emphasized layered architecture, consistent API design, secure authentication, and testability. Challenges around migrations, test auth, and session handling were addressed with targeted solutions. Additional features (search, filtering, pagination, CSV export, theming, CI/CD) enhance developer and end-user experience. The project provides a solid base for extending into categories, members, and transactions.

End of Report