Project:
Library Management System (LMS)

Team: NF2S

Team Member

- Nada Hossam
- Jakleen Eskander
- Fatma Tarek
- Sherehan Mohamed
- Shimaa Shafik

Supervision: Fatma Tolba

Content:

Project Overview

- **Project Title**: Library Management System (LMS)
- **Project Description:**

  The Library Management System (LMS) is a web-based application designed to streamline the management of library resources, book checkouts, member subscriptions, and penalties for late returns. Built using ASP.NET Core MVC, the system provides a comprehensive solution for librarians to manage book inventories, track member borrowing activities, and ensure that members are properly billed for overdue items. Librarians have the ability to register new members, record book borrow and return transactions, and track borrowing history. They can also calculate penalties for overdue books, ensuring that the library runs smoothly and efficiently. The system supports role-based authentication, enabling both librarians and members to access dedicated features. Members can log in, view available books, check their borrowing history, and track due dates and penalties.

- **Technologies Used:**
  - **ASP.NET Core MVC (.NET 8)**: Used to build the MVC architecture and handle requests, controllers, and views.
  - **Entity Framework Core**: Used as the ORM for managing data persistence and database interactions.
  - **SQL Server**: The relational database used to store the library's data, such as books, authors, members, and penalties.
  - **Bootstrap 5**: Used to design responsive and visually appealing user interfaces.
  - **jQuery and Ajax**: Used for interactive, client-side functionality like dynamic form submissions and auto-complete features.
  - **Identity Framework**: For handling user authentication and authorization, with role-based access for members and librarians.
  - **AutoMapper**: Simplifies the mapping of models to ViewModels, enhancing data transfer between the application's layers.
  - **Role-Based Authentication**: Supports two main roles (Librarian and Member), allowing role-specific functionality, including adding and managing users.

System Architecture

1. MVC Structure

The Library Management System (LMS) adheres to the Model-View-Controller (MVC) architectural pattern, which separates the application into three main components: Model, View, and Controller. This separation enhances the maintainability, testability, and scalability of the system.

a. Model

The **Model** layer represents the core data and business logic of the system. In LMS, the following entities are modeled using C# classes:

- **Book**: Represents books available in the library, with attributes like Title, ISBN, GenreId, AuthorId, AvailableCopies, etc.
- **Author**: Represents authors and stores information like Name and Biography.
- **Member**: Represents library members with details such as MemberNo, SubscribeDate, SubscribeEndOn, and the relation to the ApplicationUser class for authentication purposes.
- **Librarian**: Represents librarians who manage the system, containing fields such as LibrarianNo, HiredOn, and a relationship with the ApplicationUser.
- **Borrow**: Tracks borrowing records, including MemberId, BookId, BorrowedOn, DueDate, and ReturnedOn.
- **Penalitie**: Stores penalties for overdue books, linked to Borrow to calculate penalties based on DueDate and ReturnedOn.
- **ApplicationUser**: Extends the default IdentityUser to manage custom user attributes such as Name, NID, Address, and Phone.

These models are managed using **Entity Framework Core**, which handles database operations and defines relationships between the models.

b. View

The **View** layer consists of the user interfaces (UI) that allow users to interact with the system. In LMS, Razor Views are used to generate dynamic web pages based on the data provided by the controllers. The key views in the system include:

- **Book Management**: Provides interfaces for librarians to view, add, edit, and delete books in the library.

- **Member Management**: Allows librarians to register new members and view or update member information.
- **Checkout/Return System**: Provides views to handle the checkout and return of books, where librarians can record transactions and manage borrowing history.
- **Penalties and Borrowing History**: Displays the penalties incurred by members and shows borrowing history.
- **Authentication**: Views for login, registration, and role management for both librarians and members.

Bootstrap 5 is used in the views to create responsive, user-friendly designs, while jQuery and Ajax enhance interactivity, allowing dynamic form submissions and data validation.

c. Controller

The **Controller** acts as the intermediary between the model and the view. It handles HTTP requests from users, interacts with the model to retrieve or update data, and returns the appropriate view. Key controllers in LMS include:

- **BooksController**: Manages actions related to books, such as listing available books, adding new books, and editing or deleting existing entries.
- **MembersController**: Handles member-related operations, including registering new members and managing their data.
- **CheckoutController**: Manages book checkouts, returns, and tracks borrowing history.
- **AccountController**: Manages user authentication (login, registration) and role-based access control for librarians and members.

Each controller uses services (e.g., BookService, MemberService) to interact with the database via the models, ensuring clean separation between the business logic and presentation layers.

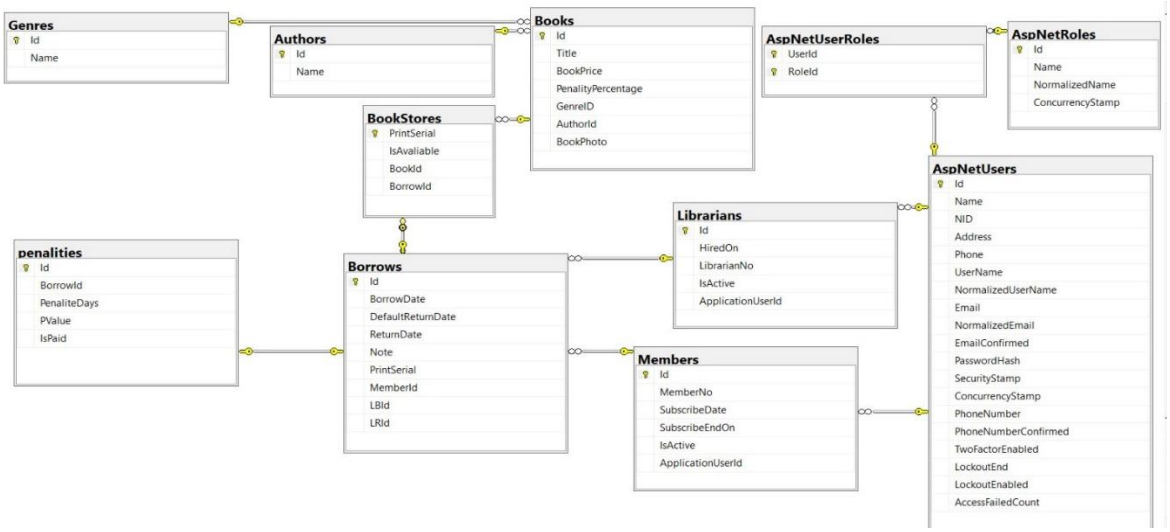
2. Database Design

The **Library Management System** employs a relational database schema to manage its data. The database is designed with several interrelated tables, corresponding to the models defined in the application. Key tables include:

- **Books**: Stores information about each book in the library, including title, genre, author, and availability.
- **Authors**: Holds information about authors and links to the books they have written.
- **Members**: Contains data about library members, including their subscription details.
- **Librarians**: Stores information about librarians, including their employment details.

- **Borrows**: Tracks borrow transactions, including borrow dates, return dates, and related penalties.
- **Penalties**: Stores penalties for overdue book returns, calculated based on the DueDate and ReturnedOn.

Entity Relationships:



- **One-to-Many**: A Book can belong to one Author, but an Author can have many Books.
- **Many-to-Many**: Members can borrow multiple books, and each book can be borrowed by multiple members over time. This relationship is handled via the Borrow table.
- **One-to-One**: Each Member and Librarian has a one-to-one relationship with the ApplicationUser class for authentication purposes.
- **Penalties**: Linked to the Borrow table to manage penalties based on late returns.

If an Entity Relationship Diagram (ERD) is required for visual representation, it would show the relationships between Books, Authors, Members, Librarians, Borrows, and Penalties, with the relevant foreign keys connecting them.

Installation and Setup

1. Prerequisites

Before setting up and running the **Library Management System (LMS)**, ensure that the following tools, frameworks, and technologies are installed on your machine:

- **.NET 8 SDK**: Required to build and run the ASP.NET Core MVC project.
- **SQL Server**: The database system where all library data is stored.
- **Visual Studio 2022 or higher**: Recommended for development and debugging.

    - Ensure that the following workloads are installed:
        - ASP.NET and web development
        - .NET Core cross-platform development
- **Entity Framework Core Tools**: Required to manage database migrations and updates.
- **Git**: To clone the project repository from version control.

2. Step-by-Step Installation Guide

Follow these steps to install and set up the **Library Management System** on your local environment:

Step 1: Clone the Repository

1. Open a terminal or command prompt.
2. Navigate to the folder where you want to clone the project.
3. Run the following command to clone the repository

```
Git clone <repository-url>
```

Replace <repository-url> with the actual URL of the project repository.

Step 2: Restore the NuGet Packages

1. Once the repository is cloned, navigate into the project folder (e.g., LMSProject).
2. Open the solution (LMSProjectAUTH.sln) in Visual Studio.
3. In Visual Studio, go to **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.
4. Click **Restore** to download all the required NuGet packages for the project.

Alternatively, from the terminal, you can run the following command:

```
Dotnet restore
```

Step 3: Set Up the Database

1. **Database Configuration**:

- Open the appsettings.json or appsettings.Development.json file.
- Update the connection string to match your local SQL Server instance. It should look something like this:

```
"ConnectionStrings": { "DefaultConnection":
"Server=your_server_name;Database=LibraryManagementDB;Trusted_Connection=True;
MultipleActiveResultSets=true" }
```

Replace your_server_name with your actual SQL Server instance name

Apply Migrations:

- Open a terminal in the project's root folder where the .csproj file is located.
- Run the following command to apply migrations and create the database schema:

```
Update-database
```

1. This will create all the necessary tables and relationships as defined in the project's Entity Framework models.
2. **Seed Initial Data** (Optional):
    - The project may contain a ContextSeed class to seed roles and initial data into the database.
    - This data will automatically be seeded when you first run the application (if the logic has been configured in Program.cs).

Step 4: Run the Application

1. **Run via Visual Studio**:
    - Open the solution (LMSProjectAUTH.sln) in Visual Studio.
    - Set the project as the startup project (if not already set).
    - Click on the **Run** button (or press F5) to start the application in debug mode.
    - Visual Studio will launch the application in your default web browser.
2. **Run via Command Line**:
    - Navigate to the project root folder containing the .csproj file.

- Use the following command to build and run the application

```
dotnet run
```

After the build process is complete, the application will start, and you can access it via the provided URL (typically http://localhost:5000).

Usage Instructions

1. How to Run the Project

After completing the installation and setup steps, follow these instructions to run and use the **Library Management System (LMS)**:

Running the Project

1. Open the project solution (LMSProjectAUTH.sln) in Visual Studio.
2. Ensure that the database is properly set up and migrations have been applied (as explained in the **Installation and Setup** section).
3. Press F5 or click the **Run** button in Visual Studio to launch the application in your default web browser.
4. Alternatively, open a terminal in the project folder and run the following command to start the application from the command line:

bash

Copy code

dotnet run

5. Once the application is running, access it via the URL http://localhost:5000 or the specified port in Visual Studio.

One-Time Admin Registration

1. Upon first run, the application will take you to a one-time admin registration page.
2. Enter the required information for the admin user and provide the secret key from the Secert key.txt file.
3. After admin registration, you will be directed to the login page.

2. Screenshots

Here are some key features with screenshots (add these screenshots to your documentation):

a. Login Page

- The login page allows users (members and librarians) to log in with their credentials.

b. Admin Dashboard

- After logging in as a librarian, you are redirected to the dashboard, where you can manage members, books, borrowing, and penalties.

c. CRUD Operations for Books

- **Create Book**: Librarians can add a new book to the system, including details like title, genre, author, and available copies.
- **Read (Book List)**: View all available books in the library with options to edit or delete.
- **Update Book**: Modify details of an existing book.
- **Delete Book**: Remove a book from the library's inventory.

d. CRUD Operations for Members

- **Create Member**: Librarians can register new members, including entering personal details and subscription information.
- **Member List**: View all registered members, with options to edit or delete.
- **Update Member**: Edit member details, such as subscription renewal.

e. Checkout and Return

- **Checkout Book**: Librarians can record the borrowing of a book by a member, including due dates.
- **Return Book**: Record the return of a book and calculate any penalties for late returns.

---

3. User Roles

The **Library Management System** supports two primary user roles, each with specific permissions and access to different features:

a. Admin (Librarian)

- **Role Description**: The librarian is the system administrator with full access to all features.
- **Accessible Features**:

- Manage Books (Add, Edit, Delete)
- Manage Members (Register, Edit, Delete)
- Record Book Checkout and Return
- Track Borrowing History
- Calculate and Manage Penalties for Overdue Books
- View and Manage the Dashboard
- Administer Users (add new librarians, etc.)

b. Member

- **Role Description**: Members are the end users of the library system who borrow books and view their borrowing status.
- **Accessible Features**:
  - View Available Books
  - Check Borrowing History
  - View Penalties (if applicable)
  - Update personal profile information

Each role's permissions are enforced via role-based access control (RBAC), ensuring that only authorized users can access specific parts of the system.

Core Features

1. Authentication and Authorization

The **Library Management System (LMS)** implements user authentication and authorization using **ASP.NET Core Identity**. This framework provides secure user management, including registration, login, password hashing, and role-based access control.

- **Authentication**:
  - Users are authenticated by entering their username and password. The authentication mechanism is powered by ASP.NET Core Identity, ensuring secure handling of credentials.
  - Upon successful authentication, users are issued an identity token that grants them access to the system based on their role.
- **Authorization**:
  - The system supports **three roles**: **Librarian**, **Admin**, and **Member**.

- **Admin**:
    - Can add, update, and delete books, authors, and genres.
    - Can register new librarians.
    - Has the ability to stop member subscriptions.
- **Librarian**:
    - Can register new members.
    - Manages the borrowing and returning of books.
    - Can handle penalties for late returns.
    - Can search for books and check availability status.
- **Member**:
    - Can view their borrowing history.
    - Can update and manage personal information.
    - Can view penalty history and any active penalties.

Role-based access control (RBAC) ensures that each role has specific permissions tailored to their responsibilities within the system, providing a secure and organized experience.

---

2. CRUD Operations

The **LMS** provides full **CRUD (Create, Read, Update, Delete)** operations for its core entities, based on the user's role:

- **Books, Authors, and Genres**:
    - **Admin Only**: Admins have exclusive rights to add, update, and delete books, authors, and genres.
    - Librarians and members can only view book details and availability.
- **Members**:
    - **Librarians**: Can register new members and manage their information.
    - **Admin**: Has the ability to stop a member’s subscription.
    - **Members**: Can manage their personal information.
- **Borrow/Return**:
    - **Librarians**: Manage the checkout and return process, ensuring accurate tracking of due dates and penalties for overdue books.

---

3. Search and Filtering

The **LMS** includes robust search and filtering functionality, especially for librarians:

- **Books**: Librarians can search for books by title, author, or genre and check for availability.
- **Checkout Records**: Librarians can filter borrowing records by member No.

---

4. Reports

Currently, the **Library Management System** does not include detailed reporting features. However, this can be expanded in future versions to provide:

- **Borrowing Reports**: Track which books are most borrowed and which members are the most active.
- **Overdue Reports**: Generate reports of overdue books and penalties incurred.
- **Membership Reports**: Analyze member activity, including subscription periods and borrowing habits.

---

5. Validation and Error Handling

The **LMS** implements client-side and server-side validation to ensure that all data entered is valid.

- **Input Validation**:
    - **Client-Side**: HTML5 and jQuery handle basic validation for forms.
    - **Server-Side**: Model validation attributes in C# are used to enforce proper data entry.
- **Error Handling**:
    - The system uses custom middleware to handle global exceptions, displaying user-friendly error messages.
    - Errors are logged using ASP.NET Core's built-in logging framework.

Design Patterns and Best Practices

1. SOLID Principles

The **Library Management System (LMS)** adheres to the **SOLID** principles, ensuring that the application is modular, maintainable, and scalable.

- **Single Responsibility Principle (SRP)**:
    - Each class in the system has one specific responsibility. For example, BooksController manages book-related logic, while the BookRepository handles data access for books.
- **Open/Closed Principle (OCP)**:
    - The system is designed to be extensible, allowing new functionality to be added without modifying existing code. For instance, new services can be added or existing services extended without changing their core logic.
- **Liskov Substitution Principle (LSP)**:
    - Subclasses can replace their parent classes without affecting functionality. This principle is followed by using interfaces such as IRepository<> and allowing different implementations without altering higher-level code.
- **Interface Segregation Principle (ISP)**:
    - Specific interfaces are created for each service or repository (e.g., IBookRepository, IUnitOfWork) to avoid having large, generalized interfaces.
- **Dependency Inversion Principle (DIP)**:
    - The application depends on abstractions (interfaces) rather than concrete implementations. This allows for easy swapping of services and repositories in the future.

---

2. Dependency Injection

Dependency Injection (DI) is implemented throughout the system to ensure loose coupling between components and to make the system testable.

- **Service Registration in Program.cs**: In the Program.cs file, services, repositories, and other components are registered using dependency injection:

These services are then injected into controllers, ensuring that the application adheres to the Dependency Inversion Principle and remains flexible and testable.

---

4. Repository Pattern

```
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
builder.Services.AddScoped<IAccountService, AccountService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<AuthorService>();
builder.Services.AddScoped<GenreService>();
builder.Services.AddScoped<ReturnBookService>();
builder.Services.AddScoped<PenaltyRepository>();
builder.Services.AddScoped<BookSearchService>();
builder.Services.AddScoped<BookRepository>();
```

The **Repository Pattern** is used to abstract the data access layer from the business logic layer.

- **Repository Registration**:

```
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped<BookRepository>();
```

- **Example of a Repository**: Each repository interacts directly with the DbContext, isolating database operations from business logic. This helps in maintaining the **Single Responsibility Principle** by separating concerns between data access and other services.
- **Usage in Services**: The repository interfaces are injected into the services, allowing the services to interact with the database without needing to know the specifics of how the data is retrieved.

---

4. Unit of Work Pattern

The **Unit of Work Pattern** ensures that all database operations are grouped into a single transaction, ensuring data consistency.

- **Unit of Work Registration**:

```
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
```

- **Example of a Unit of Work**: The UnitOfWork class is responsible for coordinating repository operations. At the end of a business transaction, UnitOfWork.Commit() is called to persist changes to the database.

5. Authentication and Authorization

Authentication and authorization are handled using **ASP.NET Core Identity**, with additional cookie settings for user sessions.

- **Identity and Cookie Configuration**:

```
builder.Services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 6;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = false;
})
.AddEntityFrameworkStores<AppDBContext>()
.AddDefaultTokenProviders();

builder.Services.ConfigureApplicationCookie(options =>
{
    options.ExpireTimeSpan = TimeSpan.FromDays(2);
    options.SlidingExpiration = true;
    options.LoginPath = "/Account/Login";
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.Cookie.HttpOnly = true;
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
});
```

Challenges and Solutions

1. Challenges Faced

a. Duplicate Usernames During Account Creation

During the development of the user registration system, an issue was encountered with **duplicate usernames**. When multiple users attempted to register with the same username, the registration process failed due to a conflict with the unique constraints of the user accounts in the database.

b. Managing Role-Based Access Control (RBAC)

Implementing role-based access control (RBAC) to ensure that the different roles (Admin, Librarian, and Member) had access only to their respective features was a complex task. Ensuring that unauthorized users were correctly redirected and unable to access restricted pages required thorough testing.

c. Handling Complex Relationships in the Database

With several entities, such as **Books**, **Members**, **Librarians**, and **Penalties**, the relationships between these entities became complex. Mapping these relationships correctly with **Entity Framework Core** and ensuring data consistency while performing CRUD operations was a technical challenge.

d. Penalties and Return Date Calculations

Calculating penalties based on the due date of book returns involved complex logic, particularly when taking into account grace periods, exceptions for holidays, and different penalty rates for overdue days. Ensuring this logic worked smoothly required thorough testing and validation.

e. Efficient Search and Filtering of Books

Implementing an efficient search and filtering system for books, which allowed users to search by title, author, genre, and availability, while maintaining good performance with a large dataset, was a challenge. The system needed to return accurate results quickly without degrading the user experience.

2. Solutions Implemented

a. Handling Duplicate Usernames

To resolve the **duplicate username issue**, a validation check was added during the user registration process. Before creating a new user, the system checks if the username already exists in the database:

```
var userExists = await _userManager.FindByNameAsync(username);
if (userExists != null)
{
    ModelState.AddModelError("", "Username already exists");
}
else
{
    // Proceed with user creation
}
```

This ensures that duplicate usernames are handled gracefully, and users are informed to choose a unique username. This solution was straightforward and eliminated registration failures.

b. Implementing Role-Based Access Control (RBAC)

Role-based access control was implemented using **ASP.NET Core Identity**. To handle unauthorized access, **authorization policies** were defined, and **[Authorize]** attributes were used in controllers. Additionally, a custom middleware (CheckForAdminMiddleware) was added to check for admin-specific actions. Unauthorized users were redirected to the login or access denied pages.

```
app.UseMiddleware<CheckForAdminMiddleware>();
app.UseAuthorization();
```

This solution ensured that only authorized users could access the features they were permitted to use, while restricting access to others.

c. Mapping Complex Relationships in the Database

To manage complex entity relationships in the database, **Entity Framework Core** was used with **foreign key constraints** and **navigation properties**. One-to-many and many-to-many relationships were carefully modeled, and migrations were used to keep the schema up to date. The use of repositories also helped abstract the complexity of data access, making the business logic cleaner.

d. Penalties and Return Date Calculations

The logic for calculating penalties based on the return date was encapsulated in a **service class** (PenaltyService), which handled all business rules related to due dates, grace periods, and penalty rates. The service was thoroughly tested to ensure it worked for all scenarios, including edge cases where books were returned right on the due date or with significant delays. This approach centralized the penalty logic and made it easier to manage and modify in the future.

e. Optimizing Search and Filtering

To ensure efficient search and filtering, **LINQ queries** were optimized to handle large datasets, and **indexes** were added to the database for fields like Title, Author, and Genre. This significantly improved the performance of search operations. Additionally, **Ajax-based search** was implemented to provide real-time feedback to users as they typed search queries, ensuring a smoother user experience.

```
var books = _context.Books
    .Where(b => b.Title.Contains(searchTerm) || b.Author.Name.Contains(searchTerm))
    .ToList();
```

Future Enhancements

While the **Library Management System (LMS)** already offers robust functionality for managing books, members, and borrowing activities, there are several potential improvements and additional features that could further enhance the system's capabilities:

1. Real-Time Notifications

- **Feature**: Implement real-time notifications for overdue books and penalties using WebSockets or SignalR.
- **Description**: Librarians and members would receive real-time alerts when books are overdue or when penalties are updated, improving communication and helping members avoid additional penalties.

2. Mobile App Integration

- **Feature**: Develop a mobile application to complement the web-based system.
- **Description**: A mobile app for members and librarians would allow users to manage their accounts, check book availability, and perform other tasks on the go. This would improve user accessibility and engagement with the system.

3. Enhanced Reporting and Analytics

- **Feature**: Add comprehensive reporting and analytics features for admins and librarians.
- **Description**: Include detailed reports on book circulation, member activity, overdue books, and penalty payments. Visual dashboards with charts and graphs could help administrators make informed decisions and identify trends in library usage.

4. Automated Reminders for Members

- **Feature**: Implement automated email or SMS reminders for members about upcoming due dates or penalties.
- **Description**: Members would receive automated reminders a few days before their book is due, helping them return books on time and avoid penalties. Integration with an SMS or email service would be required for this enhancement.

5. Bulk Book Management

- **Feature**: Implement bulk operations for book management (e.g., adding, updating, or deleting multiple books at once).
- **Description**: Librarians would be able to perform bulk operations on books, making it easier to manage large inventories and update information for multiple books simultaneously.

6. Recommendation System

- **Feature**: Develop a book recommendation system for members based on their borrowing history.
- **Description**: Using machine learning or simple algorithms, the system could recommend books to members based on their past checkouts and reading preferences, providing a personalized experience and encouraging more frequent borrowing.

7. Improved Search and Filter Functionality

- **Feature**: Enhance the search functionality by adding advanced filtering options (e.g., publication year, language, or book rating).
- **Description**: This would allow users to perform more granular searches, helping them find specific books faster. Advanced filtering could also improve usability when managing large inventories.

8. Third-Party API Integration for Book Information

- **Feature**: Integrate with external APIs to automatically retrieve and update book information, such as book covers, descriptions, and publication details.
- **Description**: APIs like Google Books or OpenLibrary could be integrated to pull book metadata, reducing the need for manual entry and ensuring accurate book details.

9. Role-Based Access for Enhanced Admin Functions

- **Feature**: Expand the admin role to include more granular role-based access control.

- **Description**: Introduce sub-roles under the admin category, such as “Super Admin” and “Manager,” each with varying permissions for handling different administrative tasks, improving the overall security and manageability of the system.

10. Performance Optimizations

- **Feature**: Optimize the system’s performance for larger datasets and more concurrent users.
- **Description**: As the system scales, additional performance optimizations, such as database indexing, query caching, and asynchronous processing, would help improve response times and system stability under heavy usage.

11. Offline Mode for Mobile App

- **Feature**: Allow librarians to manage certain tasks offline (e.g., book checkouts and returns) via a mobile app.
- **Description**: In case of network connectivity issues, librarians could continue to manage books and members offline, with data syncing once connectivity is restored.

12. Multi-Language Support

- **Feature**: Add multi-language support to improve accessibility for users in different regions.
- **Description**: By supporting multiple languages, the system can cater to a broader audience, allowing users to switch between languages in the interface.

Conclusion

The **Library Management System (LMS)** effectively implements the core business logic required for managing library operations. The system handles complex workflows such as book management, member registration, borrow/return transactions, and penalty calculations. By using **Entity Framework Core**, the program ensures that relationships between books, members, and penalties are maintained accurately, supporting key business functions like tracking borrowing history and managing overdue penalties.

The business logic is modularized through service classes like BookService, MemberService, and PenaltyService, which encapsulate the rules for each domain. The **Repository Pattern** and **Unit of Work** are employed to streamline data access and ensure consistency across transactions. Role-based access control is enforced using **ASP.NET Core Identity**, ensuring that admins, librarians, and members have access to the appropriate functionality.

This structure allows the system to achieve its business goals efficiently, providing a scalable and maintainable solution for managing library resources.