Library Management System API

Overview

Students will develop a RESTful API for a library management system using ASP.NET Core Web API, Entity Framework Core, and MS SQL Server. This project will allow students to demonstrate their understanding of API design, database interactions, error handling, middleware implementation, and dependency injection.

Domain Models

Book

- Id (int): Primary key
- Title (string, required): Book title
- ISBN (string, required): International Standard Book Number
- PublicationYear (int): Year the book was published
- Description (string): Book description
- CoverImageUrl (string): URL to book cover image
- Quantity (int): Number of copies available
- AuthorId (int, foreign key): References the Author entity

Author

- Id (int): Primary key
- FirstName (string, required): Author's first name
- LastName (string, required): Author's last name
- Biography (string): Author's biography
- DateOfBirth (DateTime?): Author's birth date
- Books (ICollection<Book>): Collection of books by this author

BorrowRecord

- Id (int): Primary key
- BookId (int, foreign key): References the Book entity
- PatronId (int, foreign key): References the Patron entity
- BorrowDate (DateTime): Date when the book was borrowed

- DueDate (DateTime): Expected return date
- ReturnDate (DateTime?): Actual return date (null if not returned)
- Status (enum): Current status (Borrowed, Returned, Overdue)

Patron

- Id (int): Primary key
- FirstName (string, required): Patron's first name
- LastName (string, required): Patron's last name
- Email (string, required): Patron's email address
- MembershipDate (DateTime): Date when the patron joined the library
- BorrowRecords (ICollection<BorrowRecord>): Collection of borrow records

API Endpoints

Books API

- GET /api/books: Get all books with pagination support
- GET /api/books/{id}: Get a specific book by ID
- GET /api/books/search?title={title}&author={author}: Search books by title or author
- POST /api/books: Add a new book
- PUT /api/books/{id}: Update an existing book
- DELETE /api/books/{id}: Delete a book
- GET /api/books/{id}/availability: Check if a book is available for borrowing

Authors API

- GET /api/authors: Get all authors with pagination
- GET /api/authors/{id}: Get a specific author by ID
- GET /api/authors/{id}/books: Get all books by a specific author
- POST /api/authors: Add a new author
- PUT /api/authors/{id}: Update an existing author
- DELETE /api/authors/{id}: Delete an author

Patrons API

- GET /api/patrons: Get all patrons with pagination
- GET /api/patrons/{id}: Get a specific patron by ID
- GET /api/patrons/{id}/books: Get all books currently borrowed by a patron
- POST /api/patrons: Add a new patron
- PUT /api/patrons/{id}: Update an existing patron
- DELETE /api/patrons/{id}: Delete a patron

Borrow Records API

- GET /api/borrow-records: Get all borrow records with filtering options
- GET /api/borrow-records/{id}: Get a specific borrow record
- POST /api/borrow-records: Create a new borrow record (check out a book)
- PUT /api/borrow-records/{id}/return: Return a book
- GET /api/borrow-records/overdue: Get all overdue books

Technical Requirements

Architecture & Design

1. **Clean Architecture**: Implement a layered architecture with separate concerns:
    - API Controllers
    - Service Layer
    - Repository Layer
    - Data Access Layer
2. **DTOs**: Use Data Transfer Objects to separate domain models from API responses/requests
    - Implement mapping between entities and DTOs (using AutoMapper or manual mapping)
3. **Repository Pattern**: Implement repository interfaces and implementations
    - IBookRepository, IAuthorRepository, etc.
    - Concrete implementations using Entity Framework Core

ASP.NET Core Features

4. **Dependency Injection**:

- Register services in Program.cs or using extension methods
- Use constructor injection in controllers and services
- Create appropriate lifetime scopes (Transient, Scoped, Singleton) based on service type

5. **Middleware**:
   - Custom exception handling middleware to catch and process exceptions
   - Request/response logging middleware
   - API version middleware
6. **Configuration**:
   - Use appsettings.json for configuration
   - Implement environment-specific settings (appsettings.Development.json, etc.)
7. **Validation**:
   - Use DataAnnotations or FluentValidation for input validation
   - Implement custom validation logic where needed

Database

8. **Entity Framework Core**:
   - Code-First approach with migrations
   - Proper relationship configuration (one-to-many, many-to-many)
   - Configuring entities using Fluent API in separate configuration classes
9. **Database Seeding**:
   - Include seed data for initial testing
10. **Transactions**:
    - Use transactions for operations that modify multiple entities

API Design & Documentation

11. **REST Principles**:
    - Follow HTTP method semantics (GET, POST, PUT, DELETE)
    - Use appropriate HTTP status codes

- Implement HATEOAS for resource linking (optional, advanced)

12. **Swagger/OpenAPI**:
    - Implement Swagger documentation
    - Include examples and descriptions in Swagger UI
13. **Pagination & Filtering**:
    - Implement pagination for collection resources
    - Support filtering and sorting

Error Handling & Logging

14. **Global Exception Handling**:
    - Create standardized error responses
    - Log exceptions appropriately
    - Return user-friendly error messages
15. **Logging**:
    - Use ILogger for application logging
    - Configure Serilog or NLog as a logging provider
    - Log to file and/or console**Acceptance Criteria**

Functional Requirements

1. The API must allow complete management of books, authors, patrons, and borrow records
2. Users must be able to search for books by title, author, or ISBN
3. The system must track book availability and prevent borrowing unavailable books
4. Overdue books must be identifiable through a dedicated endpoint
5. The API must support creating, reading, updating, and deleting all entities

Technical Requirements

1. All API endpoints must return appropriate HTTP status codes
2. The API must include proper validation with meaningful error messages
3. Database operations must be protected with transactions where appropriate

4. The application must use dependency injection for all services
5. The code must follow clean architecture principles with clear separation of concerns
6. Custom middleware must handle exceptions globally
7. The API must include comprehensive Swagger documentation
8. All endpoints must handle edge cases gracefully (not found, validation errors, etc.)
9. Database migrations must be included for easy deployment
10. The API must include filtering and pagination for collection resources

Code Quality Requirements

1. Code must follow C# coding conventions
2. Solutions must include meaningful comments
3. Complex business logic must be covered by unit tests [Optional]
4. Use of design patterns where appropriate [Optional]

Bonus Features (Optional)

1. Implement basic authentication and authorization
2. Add rate limiting
3. **URL-friendly ID obfuscation**:
    1. Implement a system to convert numeric database IDs into URL-friendly string identifiers for all external-facing APIs
    2. Use a NuGet package such as HashIds, IdGen, or ShortGuid to generate these identifiers
    3. Ensure the conversion is reversible (can decode back to original ID)
    4. Create appropriate middleware or filters to handle the conversion automatically