Great — below is a complete, ready-to-use implementation for the **Library Management API** (ASP.NET Core, .NET 6) plus an **NUnit** test project that uses **Moq** to unit-test the BookController. I built the API using a simple in-memory repository so you can run it immediately; you can later swap in an EF Core DbContext repository with the scaffolding/commands you already have.

I'll give you:

* Project structure & important files
* Full source for each file (model, repository interface, in-memory repo, controller, and Program)
* NUnit test project with Moq and tests covering positive & negative cases
* Commands to create/run/build/test

No emojis, clean and practical.

**Project structure (suggested)**

dotnetapp/

├─ Controllers/

│ └─ BooksController.cs

├─ Models/

│ └─ Book.cs

├─ Repositories/

│ ├─ IBookRepository.cs

│ └─ InMemoryBookRepository.cs

├─ Program.cs

├─ dotnetapp.csproj

tests/

└─ MyTestNUnitProject/

├─ MyTestNUnitProject.csproj

└─ BooksControllerTests.cs

**API Code**

**Models/Book.cs**

using System;

namespace dotnetapp.Models

{

public class Book

{

public int BookId { get; set; }

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

public string ISBN { get; set; } = string.Empty;

public DateTime PublishedDate { get; set; }

}

}

**Repositories/IBookRepository.cs**

using System.Collections.Generic;

using dotnetapp.Models;

namespace dotnetapp.Repositories

{

public interface IBookRepository

{

IEnumerable<Book> GetBooks();

Book? GetBook(int bookId);

Book CreateBook(Book book);

Book? UpdateBook(Book book);

bool DeleteBook(int bookId);

}

}

**Repositories/InMemoryBookRepository.cs**

using System.Collections.Concurrent;

using System.Collections.Generic;

using System.Linq;

using dotnetapp.Models;

namespace dotnetapp.Repositories

{

public class InMemoryBookRepository : IBookRepository

{

private readonly ConcurrentDictionary<int, Book> \_store = new();

private int \_currentId = 0;

public InMemoryBookRepository()

{

// seed sample data (optional)

CreateBook(new Book { Title = "Clean Code", Author = "Robert C. Martin", ISBN = "9780132350884", PublishedDate = new System.DateTime(2008,8,1) });

CreateBook(new Book { Title = "Domain-Driven Design", Author = "Eric Evans", ISBN = "9780321125217", PublishedDate = new System.DateTime(2003,8,30) });

}

public IEnumerable<Book> GetBooks() => \_store.Values.OrderBy(b => b.BookId);

public Book? GetBook(int bookId) => \_store.TryGetValue(bookId, out var book) ? book : null;

public Book CreateBook(Book book)

{

var id = System.Threading.Interlocked.Increment(ref \_currentId);

book.BookId = id;

\_store.TryAdd(id, book);

return book;

}

public Book? UpdateBook(Book book)

{

if (!\_store.ContainsKey(book.BookId)) return null;

\_store[book.BookId] = book;

return book;

}

public bool DeleteBook(int bookId)

{

return \_store.TryRemove(bookId, out \_);

}

}

}

**Controllers/BooksController.cs**

using Microsoft.AspNetCore.Mvc;

using dotnetapp.Models;

using dotnetapp.Repositories;

using System.Collections.Generic;

namespace dotnetapp.Controllers

{

[ApiController]

[Route("api/books")]

public class BooksController : ControllerBase

{

private readonly IBookRepository \_repo;

public BooksController(IBookRepository repo)

{

\_repo = repo;

}

// GET /api/books

[HttpGet]

public ActionResult<IEnumerable<Book>> GetBooks()

{

var books = \_repo.GetBooks();

return Ok(books);

}

// GET /api/books/{id}

[HttpGet("{id:int}")]

public ActionResult<Book> GetBook(int id)

{

var book = \_repo.GetBook(id);

if (book == null) return NotFound();

return Ok(book);

}

// POST /api/books

[HttpPost]

public ActionResult<Book> CreateBook([FromBody] Book book)

{

if (book == null) return BadRequest();

var created = \_repo.CreateBook(book);

return CreatedAtAction(nameof(GetBook), new { id = created.BookId }, created);

}

// PUT /api/books/{id}

[HttpPut("{id:int}")]

public ActionResult<Book> UpdateBook(int id, [FromBody] Book book)

{

if (book == null || id != book.BookId) return BadRequest();

var updated = \_repo.UpdateBook(book);

if (updated == null) return NotFound();

return Ok(updated);

}

// DELETE /api/books/{id}

[HttpDelete("{id:int}")]

public ActionResult DeleteBook(int id)

{

var deleted = \_repo.DeleteBook(id);

if (!deleted) return NotFound();

return NoContent();

}

}

}

**Program.cs (minimal hosting, .NET 6)**

using dotnetapp.Repositories;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddSingleton<IBookRepository, InMemoryBookRepository>(); // replaceable with EF-backed repo

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseRouting();

app.UseAuthorization();

app.MapControllers();

// If you want to run on port 8080 using launchSettings.json, update Profiles there.

// But you can also pass --urls "http://0.0.0.0:8080" when running for testing locally.

app.Run();

**Test Project (NUnit + Moq)**

Create the test project and add references:

# inside workspace root

dotnet new nunit -n MyTestNUnitProject

dotnet sln add dotnetapp/dotnetapp.csproj

dotnet sln add MyTestNUnitProject/MyTestNUnitProject.csproj

# add reference from tests to app

dotnet add MyTestNUnitProject/MyTestNUnitProject.csproj reference dotnetapp/dotnetapp.csproj

# add Moq

dotnet add MyTestNUnitProject/MyTestNUnitProject.csproj package Moq --version 4.18.4

**tests/MyTestNUnitProject/BooksControllerTests.cs**

using NUnit.Framework;

using Moq;

using dotnetapp.Controllers;

using dotnetapp.Repositories;

using dotnetapp.Models;

using Microsoft.AspNetCore.Mvc;

using System;

using System.Collections.Generic;

namespace MyTestNUnitProject

{

public class BooksControllerTests

{

private Mock<IBookRepository> \_repoMock = null!;

private BooksController \_controller = null!;

[SetUp]

public void Setup()

{

\_repoMock = new Mock<IBookRepository>();

\_controller = new BooksController(\_repoMock.Object);

}

[Test]

public void GetBooks\_ReturnsOkWithList()

{

var sample = new List<Book>

{

new Book { BookId = 1, Title = "A", Author = "X", ISBN="111", PublishedDate = DateTime.Today },

new Book { BookId = 2, Title = "B", Author = "Y", ISBN="222", PublishedDate = DateTime.Today }

};

\_repoMock.Setup(r => r.GetBooks()).Returns(sample);

var result = \_controller.GetBooks();

Assert.IsInstanceOf<OkObjectResult>(result.Result);

var ok = result.Result as OkObjectResult;

Assert.IsNotNull(ok);

Assert.AreEqual(sample, ok!.Value);

}

[Test]

public void GetBook\_Found\_ReturnsOk()

{

var book = new Book { BookId = 1, Title = "A", Author = "X" };

\_repoMock.Setup(r => r.GetBook(1)).Returns(book);

var result = \_controller.GetBook(1);

Assert.IsInstanceOf<OkObjectResult>(result.Result);

var ok = result.Result as OkObjectResult;

Assert.IsNotNull(ok);

Assert.AreEqual(book, ok!.Value);

}

[Test]

public void GetBook\_NotFound\_ReturnsNotFound()

{

\_repoMock.Setup(r => r.GetBook(42)).Returns((Book?)null);

var result = \_controller.GetBook(42);

Assert.IsInstanceOf<NotFoundResult>(result.Result);

}

[Test]

public void CreateBook\_Valid\_ReturnsCreatedAtAction()

{

var input = new Book { Title = "New", Author = "Author" };

var created = new Book { BookId = 5, Title = "New", Author = "Author" };

\_repoMock.Setup(r => r.CreateBook(It.IsAny<Book>())).Returns(created);

var actionResult = \_controller.CreateBook(input);

Assert.IsInstanceOf<CreatedAtActionResult>(actionResult.Result);

var createdResult = actionResult.Result as CreatedAtActionResult;

Assert.IsNotNull(createdResult);

Assert.AreEqual(created, createdResult!.Value);

}

[Test]

public void UpdateBook\_IdMismatch\_ReturnsBadRequest()

{

var book = new Book { BookId = 2, Title = "X" };

var result = \_controller.UpdateBook(3, book);

Assert.IsInstanceOf<BadRequestResult>(result.Result);

}

[Test]

public void UpdateBook\_NotFound\_ReturnsNotFound()

{

var book = new Book { BookId = 1, Title = "X" };

\_repoMock.Setup(r => r.UpdateBook(book)).Returns((Book?)null);

var result = \_controller.UpdateBook(1, book);

Assert.IsInstanceOf<NotFoundResult>(result.Result);

}

[Test]

public void UpdateBook\_Success\_ReturnsOk()

{

var book = new Book { BookId = 1, Title = "X" };

\_repoMock.Setup(r => r.UpdateBook(book)).Returns(book);

var result = \_controller.UpdateBook(1, book);

Assert.IsInstanceOf<OkObjectResult>(result.Result);

var ok = result.Result as OkObjectResult;

Assert.IsNotNull(ok);

Assert.AreEqual(book, ok!.Value);

}

[Test]

public void DeleteBook\_NotFound\_ReturnsNotFound()

{

\_repoMock.Setup(r => r.DeleteBook(10)).Returns(false);

var result = \_controller.DeleteBook(10);

Assert.IsInstanceOf<NotFoundResult>(result);

}

[Test]

public void DeleteBook\_Success\_ReturnsNoContent()

{

\_repoMock.Setup(r => r.DeleteBook(1)).Returns(true);

var result = \_controller.DeleteBook(1);

Assert.IsInstanceOf<NoContentResult>(result);

}

}

}

**Commands: create, build, run, test**

Assuming you're in a workspace directory:

# create the API project (if not already)

dotnet new webapi -n dotnetapp

# create solution (optional)

dotnet new sln -n dotnetappSolution

dotnet sln add dotnetapp/dotnetapp.csproj

# create the test project

dotnet new nunit -n MyTestNUnitProject

dotnet sln add MyTestNUnitProject/MyTestNUnitProject.csproj

# add project references

dotnet add MyTestNUnitProject/MyTestNUnitProject.csproj reference dotnetapp/dotnetapp.csproj

# add Moq to tests

dotnet add MyTestNUnitProject/MyTestNUnitProject.csproj package Moq

# build both

dotnet build

# run the API (pass urls to run on specific port)

cd dotnetapp

dotnet run --urls "http://0.0.0.0:8080"

# from workspace root: run tests

dotnet test

**Notes & next steps**

1. **EF Core swap**: replace InMemoryBookRepository with an EF implementation that uses DbContext. The IBookRepository interface stays same — only DI registration changes (e.g., AddScoped<IBookRepository, EfBookRepository>()).
2. **Validation**: add model validation attributes ([Required], [MaxLength]) to Book if you want stronger input checks.
3. **Integration tests**: you can add WebApplicationFactory-based integration tests later to verify HTTP responses end-to-end.
4. **Launch on port 8080**: as described in your instructions, update launchSettings.json or use the --urls option when running.
5. **Test coverage**: the provided unit tests cover common positive/negative controller scenarios. You can add tests for model validation, concurrency, and repository logic.

If you want, I can:

* Convert the repository to EF Core DbContext + migrations and provide the DbContext & EF repository implementation.
* Add request validation with DataAnnotations and tests that verify model validation errors.
* Provide a Postman collection or sample curl commands to exercise the endpoints.

Which of those (if any) would you like next?