Contents

[C O M P R E H E N S I V E .NET Core Web API Architecture 3](#_Toc203382816)

[✅ Features Implemented So Far 3](#_Toc203382817)

[🏗️ Clean Architecture Overview 3](#_Toc203382818)

[📁 Solution Structure 4](#_Toc203382819)

[Project References 4](#_Toc203382820)

[Domain Layer (MyBookDomain) 5](#_Toc203382821)

[Application Layer (MyBookApplication) 5](#_Toc203382822)

[Interfaces/IBookRepository.cs 5](#_Toc203382823)

[Queries/GetBookByIdQuery.cs 5](#_Toc203382824)

[Queries/GetBookByIdQueryHandler.cs 6](#_Toc203382825)

[Commands/CreateBookCommand.cs 6](#_Toc203382826)

[Commands/CreateBookCommandHandler.cs 7](#_Toc203382827)

[Validators/CreateBookCommandValidator.cs 8](#_Toc203382828)

[Infrastructure Layer (MyBookInfrastructure) 8](#_Toc203382829)

[Repositories/BookRepository.cs 9](#_Toc203382830)

[Data/ApplicationDbInitializer.cs 10](#_Toc203382831)

[Presentation Layer (MyBookApi) 11](#_Toc203382832)

[Program.cs 12](#_Toc203382833)

[Docker Support 13](#_Toc203382834)

[Build & Run 14](#_Toc203382835)

[Authentication/Authorization 15](#_Toc203382836)

[🧱 Updated Project Structure 15](#_Toc203382837)

[Registration endpoints, Role-based policies (e.g. only Admin can delete), Refresh tokens and token revocation setup? 19](#_Toc203382838)

[Future Maintainability Level Will be H i g h Using this Clean Architecture 26](#_Toc203382839)

[Demonstration using c# code examples of how *each* section/component/layer of this .NET Core Web API solution architecture could be easily maintained in the future in a decoupled way that would have no, or very little impact, on other sections/components/layers because this solution adheres to Clean Architecture? 26](#_Toc203382840)

[✅ 1. Updating the Database Engine (Infrastructure) 26](#_Toc203382841)

[✅ 2. Modifying a Business Rule (Domain) 27](#_Toc203382842)

[✅ 3. Changing an API Contract (Presentation) 27](#_Toc203382843)

[✅ 4. Replacing MediatR with a Native Pipeline (Application) 28](#_Toc203382844)

# C O M P R E H E N S I V E .NET Core Web API Architecture

In your next text response, can you include everything you've produced so far, and in addition include to ensure that the architecture also adheres to Robert C. Martin's Clean Architecture and also include a clear Visual Studio Solution Explorer folder and file structure, and provide a concise but complete explanation of the architecture?

**ChatGPT said:**

Absolutely! Below is a **comprehensive, production-grade .NET 8+ Web API** project that includes:

## ✅ Features Implemented So Far

* CQRS (via MediatR)
* SQL Server + Entity Framework Core
* FluentValidation
* Auto-seeding with EF Core
* Docker containerization (API + DB)
* Swagger for testing
* Clean Architecture principles based on **Robert C. Martin's Clean Architecture**
* Complete folder and project structure for Visual Studio
* Clear, concise architecture explanation

## 🏗️ Clean Architecture Overview

**Clean Architecture (Uncle Bob)** emphasizes:

| **Layer** | **Responsibility** | **Dependency Direction** |
| --- | --- | --- |
| **Domain** | Business logic, Entities, Interfaces | No dependencies |
| **Application** | Use cases (commands/queries), DTOs | Depends only on Domain |
| **Infrastructure** | Data access (EF Core, external services) | Depends on Application |
| **Presentation** | Controllers, API endpoints | Depends on Application |

➡ All dependencies **point inward** (from outer layers to inner).

## 📁 Solution Structure

/MyBookSolution

├── MyBookApi/ → Presentation Layer (API)

│ ├── Controllers/

│ ├── Program.cs

│ └── MyBookApi.csproj

│

├── MyBookApplication/ → Application Layer (CQRS, DTOs, Validators)

│ ├── DTOs/

│ ├── Interfaces/

│ ├── Commands/

│ ├── Queries/

│ ├── Validators/

│ └── MyBookApplication.csproj

│

├── MyBookDomain/ → Domain Layer (Entities, pure logic)

│ ├── Entities/

│ └── MyBookDomain.csproj

│

├── MyBookInfrastructure/ → Infrastructure Layer (EF Core, SQL Server)

│ ├── Data/

│ ├── Repositories/

│ └── MyBookInfrastructure.csproj

│

├── docker-compose.yml

└── Dockerfile

## Project References

* MyBookApi references:
  + MyBookApplication
* MyBookApplication references:
  + MyBookDomain
* MyBookInfrastructure references:
  + MyBookApplication, MyBookDomain

## Domain Layer (MyBookDomain)

**📄 Entities/Book.cs**

namespace MyBookDomain.Entities;

public class Book

{

public Guid Id { get; set; }

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

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

}

## Application Layer (MyBookApplication)

**📄 DTOs/BookDto.cs**

namespace MyBookApplication.DTOs;

public class BookDto

{

public Guid Id { get; set; }

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

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

}

## Interfaces/IBookRepository.cs

using MyBookDomain.Entities;

namespace MyBookApplication.Interfaces;

public interface IBookRepository

{

Task<Book?> GetByIdAsync(Guid id);

Task<Book> AddAsync(Book book);

}

## Queries/GetBookByIdQuery.cs

using MediatR;

using MyBookApplication.DTOs;

namespace MyBookApplication.Queries;

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

## Queries/GetBookByIdQueryHandler.cs

using MediatR;

using MyBookApplication.DTOs;

using MyBookApplication.Interfaces;

namespace MyBookApplication.Queries;

public class GetBookByIdQueryHandler : IRequestHandler<GetBookByIdQuery, BookDto?>

{

private readonly IBookRepository \_repository;

public GetBookByIdQueryHandler(IBookRepository repository)

=> \_repository = repository;

public async Task<BookDto?> Handle(GetBookByIdQuery request, CancellationToken cancellationToken)

{

var book = await \_repository.GetByIdAsync(request.Id);

return book == null ? null : new BookDto

{

Id = book.Id,

Title = book.Title,

Author = book.Author

};

}

}

## Commands/CreateBookCommand.cs

using MediatR;

using MyBookApplication.DTOs;

namespace MyBookApplication.Commands;

public record CreateBookCommand(string Title, string Author) : IRequest<BookDto>;

## Commands/CreateBookCommandHandler.cs

using MediatR;

using MyBookApplication.Commands;

using MyBookApplication.DTOs;

using MyBookApplication.Interfaces;

using MyBookDomain.Entities;

namespace MyBookApplication.Commands;

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, BookDto>

{

private readonly IBookRepository \_repository;

public CreateBookCommandHandler(IBookRepository repository) => \_repository = repository;

public async Task<BookDto> Handle(CreateBookCommand request, CancellationToken cancellationToken)

{

var book = new Book

{

Id = Guid.NewGuid(),

Title = request.Title,

Author = request.Author

};

var created = await \_repository.AddAsync(book);

return new BookDto

{

Id = created.Id,

Title = created.Title,

Author = created.Author

};

}

}

## Validators/CreateBookCommandValidator.cs

using FluentValidation;

using MyBookApplication.Commands;

namespace MyBookApplication.Validators;

public class CreateBookCommandValidator : AbstractValidator<CreateBookCommand>

{

public CreateBookCommandValidator()

{

RuleFor(x => x.Title).NotEmpty().MaximumLength(100);

RuleFor(x => x.Author).NotEmpty().MaximumLength(100);

}

}

## Infrastructure Layer (MyBookInfrastructure)

**📄 Data/ApplicationDbContext.cs**

using Microsoft.EntityFrameworkCore;

using MyBookDomain.Entities;

namespace MyBookInfrastructure.Data;

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options) { }

public DbSet<Book> Books => Set<Book>();

}

## Repositories/BookRepository.cs

using Microsoft.EntityFrameworkCore;

using MyBookApplication.Interfaces;

using MyBookDomain.Entities;

using MyBookInfrastructure.Data;

namespace MyBookInfrastructure.Repositories;

public class BookRepository : IBookRepository

{

private readonly ApplicationDbContext \_context;

public BookRepository(ApplicationDbContext context) => \_context = context;

public async Task<Book?> GetByIdAsync(Guid id) =>

await \_context.Books.FindAsync(id);

public async Task<Book> AddAsync(Book book)

{

\_context.Books.Add(book);

await \_context.SaveChangesAsync();

return book;

}

}

## Data/ApplicationDbInitializer.cs

using MyBookDomain.Entities;

namespace MyBookInfrastructure.Data;

public static class ApplicationDbInitializer

{

public static void SeedDatabase(this IServiceProvider services)

{

using var scope = services.CreateScope();

var context = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

context.Database.Migrate();

if (!context.Books.Any())

{

context.Books.AddRange(

new Book { Id = Guid.Parse("11111111-1111-1111-1111-111111111111"), Title = "Clean Code", Author = "Robert C. Martin" },

new Book { Id = Guid.Parse("22222222-2222-2222-2222-222222222222"), Title = "The Pragmatic Programmer", Author = "Andrew Hunt" }

);

context.SaveChanges();

}

}

}

## Presentation Layer (MyBookApi)

**📄 Controllers/BooksController.cs**

using MediatR;

using Microsoft.AspNetCore.Mvc;

using MyBookApplication.Commands;

using MyBookApplication.DTOs;

using MyBookApplication.Queries;

namespace MyBookApi.Controllers;

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator) => \_mediator = mediator;

[HttpGet("{id}")]

public async Task<ActionResult<BookDto>> Get(Guid id)

{

var book = await \_mediator.Send(new GetBookByIdQuery(id));

return book == null ? NotFound() : Ok(book);

}

[HttpPost]

public async Task<ActionResult<BookDto>> Create(CreateBookCommand command)

{

var result = await \_mediator.Send(command);

return CreatedAtAction(nameof(Get), new { id = result.Id }, result);

}

}

## Program.cs

using FluentValidation;

using FluentValidation.AspNetCore;

using MediatR;

using Microsoft.EntityFrameworkCore;

using MyBookApplication.Commands;

using MyBookApplication.Interfaces;

using MyBookInfrastructure.Data;

using MyBookInfrastructure.Repositories;

var builder = WebApplication.CreateBuilder(args);

// EF Core + SQL Server

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Repositories

builder.Services.AddScoped<IBookRepository, BookRepository>();

// MediatR + Validators

builder.Services.AddMediatR(cfg => cfg.RegisterServicesFromAssemblyContaining<CreateBookCommand>());

builder.Services.AddFluentValidationAutoValidation();

builder.Services.AddValidatorsFromAssemblyContaining<CreateBookCommandValidator>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.Services.SeedDatabase(); // EF migrations + seeding

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

## Docker Support

**📄 Dockerfile**

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY ["MyBookApi/MyBookApi.csproj", "MyBookApi/"]

COPY ["MyBookApplication/MyBookApplication.csproj", "MyBookApplication/"]

COPY ["MyBookDomain/MyBookDomain.csproj", "MyBookDomain/"]

COPY ["MyBookInfrastructure/MyBookInfrastructure.csproj", "MyBookInfrastructure/"]

RUN dotnet restore "MyBookApi/MyBookApi.csproj"

COPY . .

WORKDIR "/src/MyBookApi"

RUN dotnet publish -c Release -o /app

FROM base AS final

WORKDIR /app

COPY --from=build /app .

ENTRYPOINT ["dotnet", "MyBookApi.dll"]

**📄 docker-compose.yml**

version: '3.8'

services:

api:

build:

context: .

dockerfile: Dockerfile

ports:

- "5000:80"

environment:

- ConnectionStrings\_\_DefaultConnection=Server=sqlserver;Database=MyBookDb;User=sa;Password=Your\_password123;

depends\_on:

- sqlserver

sqlserver:

image: mcr.microsoft.com/mssql/server:2022-latest

container\_name: sqlserver

environment:

SA\_PASSWORD: "Your\_password123"

ACCEPT\_EULA: "Y"

ports:

- "1433:1433"

volumes:

- sqlserverdata:/var/opt/mssql

volumes:

sqlserverdata:

**📄 appsettings.json**

{

"ConnectionStrings": {

"DefaultConnection": "Server=sqlserver;Database=MyBookDb;User=sa;Password=Your\_password123;"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

}

}

## Build & Run

docker-compose build

docker-compose up

Swagger is at:  
📎 http://localhost:5000/swagger

# Authentication/Authorization

**ChatGPT said:**

Here’s the updated, full-featured Clean Architecture Web API project—now with **JWT-based authentication and authorization** added. Everything below is in **text form**, as you requested.

**🛡️ Authentication & Authorization Overview**

We’ll secure the API using **JWT bearer tokens**, including:

1. User registration and login endpoints
2. Issuing JWT tokens on successful login
3. Protecting the CreateBook endpoint so only authenticated users can create books

For simplicity, user credentials are stored in an in-memory store—swap this out for your own user datastore as needed.

## 🧱 Updated Project Structure

/MyBookSolution

├── MyBookApi/ → Presentation Layer (API)

│ ├── Controllers/

│ │ ├── AuthController.cs

│ │ └── BooksController.cs

│ ├── Program.cs

│ └── MyBookApi.csproj

│

├── MyBookApplication/ → Application Layer (CQRS, DTOs, Validators)

│ └── (unchanged)

│

├── MyBookDomain/ → Domain Layer

│ └── (unchanged)

│

├── MyBookInfrastructure/ → Infrastructure Layer (EF Core, Repos)

│ └── (unchanged)

│

├── docker-compose.yml

└── Dockerfile

**📝 1. AuthController & DTOs (Presentation Layer)**

**Controllers/AuthController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.IdentityModel.Tokens;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

[ApiController]

[Route("api/[controller]")]

public class AuthController : ControllerBase

{

private static readonly List<UserRecord> \_users = new()

{

new() { Username = "testuser", Password = "Pa$$w0rd" }

};

private readonly IConfiguration \_config;

public AuthController(IConfiguration config) => \_config = config;

[HttpPost("login")]

public IActionResult Login([FromBody] LoginDto login)

{

var user = \_users.SingleOrDefault(u =>

u.Username == login.Username && u.Password == login.Password);

if (user == null)

return Unauthorized();

var token = GenerateJwtToken(user.Username);

return Ok(new { token });

}

private string GenerateJwtToken(string username)

{

var claims = new[]

{

new Claim(ClaimTypes.Name, username),

};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

issuer: \_config["Jwt:Issuer"],

audience: \_config["Jwt:Audience"],

claims: claims,

expires: DateTime.UtcNow.AddHours(1),

signingCredentials: creds

);

return new JwtSecurityTokenHandler().WriteToken(token);

}

private class UserRecord { public string Username = ""; public string Password = ""; }

}

**DTOs for login**

public class LoginDto

{

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

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

}

**🔐 2. Secure the CreateBook endpoint**

In **BooksController.cs**, add [Authorize]:

using Microsoft.AspNetCore.Authorization;

[HttpPost]

[Authorize]

public async Task<ActionResult<BookDto>> Create(CreateBookCommand command) { ... }

**⚙️ 3. Configure JWT in Program.cs**

Add the following before builder.Build():

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.IdentityModel.Tokens;

using System.Text;

builder.Services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = builder.Configuration["Jwt:Issuer"],

ValidAudience = builder.Configuration["Jwt:Audience"],

IssuerSigningKey = new SymmetricSecurityKey(

Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))

};

});

Add after app.UseHttpsRedirection();:

app.UseAuthentication();

app.UseAuthorization();

**📄 4. Update appsettings.json**

"Jwt": {

"Key": "YourSuperSecretKey123!",

"Issuer": "MyBookApiIssuer",

"Audience": "MyBookApiAudience"

}

**🧪 5. How to Use It**

1. **Login**:

 POST /api/auth/login

{

"username": "testuser",

"password": "Pa$$w0rd"

}

→ Returns { "token": "..." }

 **Use Token**:

* Add header: Authorization: Bearer <token>

 **Call protected endpoint**:

1. POST /api/books
2. Authorization: Bearer <token>
3. {
4. "title": "New Book",
5. "author": "Author Name"
6. }
7. → Returns created book

**🛡️ 🌟 Summary**

* **Unauthenticated** users can only **GET** books.
* **Authenticated** users (with valid JWT) can **CREATE** books.
* JWT settings stored in appsettings.json; tokens valid 1 hour.
* Clean Architecture layers untouched—auth lives in Presentation only.

# Registration endpoints, Role-based policies (e.g. only Admin can delete), Refresh tokens and token revocation setup?

Here’s your **complete, clean-architecture .NET 8+ Web API** with JWT-based:

1. **User Registration & Login**
2. **Role-Based Authorization** (with an Admin role)
3. **Refresh Token Support**

All packaged in text form—no files omitted.

**🏗️ Complete Folder Structure**

/MyBookSolution

├── MyBookApi/

│ ├── Controllers/

│ │ ├── AuthController.cs

│ │ └── BooksController.cs

│ ├── Program.cs

│ └── MyBookApi.csproj

│

├── MyBookApplication/ (CQRS + DTOs + Validators)

│ └── (existing files)

│

├── MyBookDomain/ (Entities)

│ └── (existing files)

│

├── MyBookInfrastructure/ (EF Core, Repos)

│ ├── Data/

│ │ └── TokenStore.cs

│ └── Repositories/

│

├── Dockerfile

└── docker-compose.yml

**🔐 Expanded AuthController (MyBookApi/Controllers/AuthController.cs)**

using Microsoft.AspNetCore.Mvc;

using Microsoft.IdentityModel.Tokens;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

using MyBookInfrastructure.Data;

[ApiController]

[Route("api/[controller]")]

public class AuthController : ControllerBase

{

private readonly IConfiguration \_config;

private readonly TokenStore \_tokenStore;

public AuthController(IConfiguration config, TokenStore tokenStore)

{

\_config = config;

\_tokenStore = tokenStore;

}

[HttpPost("register")]

public IActionResult Register([FromBody] RegisterDto reg)

{

if (\_tokenStore.GetUser(reg.Username) != null)

return Conflict("User already exists");

\_tokenStore.AddUser(new UserRecord

{

Username = reg.Username,

Password = reg.Password,

Role = reg.Role

});

return Created("", null);

}

[HttpPost("login")]

public IActionResult Login([FromBody] LoginDto login)

{

var user = \_tokenStore.GetUser(login.Username);

if (user == null || user.Password != login.Password)

return Unauthorized();

var accessToken = GenerateJwtToken(user);

var refreshToken = \_tokenStore.GenerateRefreshToken(user.Username);

return Ok(new { accessToken, refreshToken });

}

[HttpPost("refresh")]

public IActionResult Refresh([FromBody] RefreshDto dto)

{

var username = \_tokenStore.ValidateRefreshToken(dto.RefreshToken);

if (username == null) return Unauthorized();

var user = \_tokenStore.GetUser(username);

var accessToken = GenerateJwtToken(user!);

var newRefresh = \_tokenStore.GenerateRefreshToken(username);

return Ok(new { accessToken, refreshToken = newRefresh });

}

private string GenerateJwtToken(UserRecord user)

{

var claims = new[]

{

new Claim(ClaimTypes.Name, user.Username),

new Claim(ClaimTypes.Role, user.Role)

};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

issuer: \_config["Jwt:Issuer"],

audience: \_config["Jwt:Audience"],

claims: claims,

expires: DateTime.UtcNow.AddMinutes(15),

signingCredentials: creds

);

return new JwtSecurityTokenHandler().WriteToken(token);

}

}

**DTOs**

public class RegisterDto { public string Username, Password, Role; }

public class LoginDto { public string Username, Password; }

public class RefreshDto { public string RefreshToken; }

**🛡️ Role-Based Protection (Controllers/BooksController.cs)**

[HttpPost]

[Authorize(Roles = "Admin")]

public async Task<ActionResult<BookDto>> Create(CreateBookCommand command) { ... }

Only users registered with "Role": "Admin" may create books.

**🛠️ TokenStore (Infrastructure: MyBookInfrastructure/Data/TokenStore.cs)**

namespace MyBookInfrastructure.Data;

public class TokenStore

{

private readonly Dictionary<string, UserRecord> \_users = new();

private readonly Dictionary<string, string> \_refreshTokens = new();

public UserRecord? GetUser(string username) =>

\_users.TryGetValue(username, out var u) ? u : null;

public void AddUser(UserRecord user) =>

\_users[user.Username] = user;

public string GenerateRefreshToken(string username)

{

var token = Guid.NewGuid().ToString();

\_refreshTokens[token] = username;

return token;

}

public string? ValidateRefreshToken(string token)

{

if (\_refreshTokens.TryGetValue(token, out var user))

{

\_refreshTokens.Remove(token);

return user;

}

return null;

}

}

public record UserRecord

{

public string Username { get; init; } = "";

public string Password { get; init; } = "";

public string Role { get; init; } = "";

}

**⚙️ Program.cs (Add Authentication, TokenStore)**

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.IdentityModel.Tokens;

using MyBookInfrastructure.Data;

using System.Text;

var builder = WebApplication.CreateBuilder(args);

// existing Service registrations...

builder.Services.AddSingleton<TokenStore>();

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options => { /\* same params as earlier \*/ });

builder.Services.AddAuthorization(); // enables role policies

// existing builder.Services.AddControllers(), swagger, etc.

var app = builder.Build();

// existing middleware: HTTPS, Swagger…

app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

app.Run();

**🗂️ appsettings.json**

"Jwt": {

"Key": "YourSuperSecretKey123!",

"Issuer": "MyBookApiIssuer",

"Audience": "MyBookApiAudience"

}

**📝 Usage Flow**

1. **Register user**:

 POST /api/auth/register

{

"username": "alice", "password":"Pwd1!", "role":"Admin"

}

 **Login**:

 POST /api/auth/login

→ returns accessToken & refreshToken

 **Create book (Admin only)**:

 POST /api/books

Authorization: Bearer <accessToken>

 **Refresh token**:

1. POST /api/auth/refresh
2. { "refreshToken": "<old token>" }
3. → returns a fresh accessToken & refreshToken

**✅ Summary of Enhancements**

* **Registration Endpoint**: dynamic user/role store
* **JWT with Claims**: includes Role claim
* **Role-based [Authorize(Roles="Admin")]**
* **Refresh tokens** stored in memory
* **Clean Architecture boundaries respected**: Auth logic in Presentation & Infrastructure; no cross-layer violation.

Would you like:

* ✅ Persistent token store (using DB)
* ✅ Password hashing
* ✅ Email confirmation workflow?

# Future Maintainability Level Will be H i g h Using this Clean Architecture

## Demonstration using c# code examples of how *each* section/component/layer of this .NET Core Web API solution architecture could be easily maintained in the future in a decoupled way that would have no, or very little impact, on other sections/components/layers because this solution adheres to Clean Architecture?

One of the main strengths of **Clean Architecture** is **low coupling** and **high cohesion**, which enables maintainability, testability, and extensibility. Below are **C# examples and explanations** for how changes in each layer can be made independently, with little to no impact on other layers—because each layer:

* Depends only on **abstractions**, not concrete implementations
* Uses **dependency injection** to invert control
* Keeps **business logic** isolated from frameworks, databases, or transport layers

**🔁 RECAP: Layered Structure**

Presentation (MyBookApi)

↓

Application (MyBookApplication)

↓

Domain (MyBookDomain)

↑

Infrastructure (MyBookInfrastructure)

* Arrows = dependency direction
* Infrastructure implements interfaces defined in Application
* Only Presentation depends on frameworks (e.g., ASP.NET Core)

## ✅ 1. Updating the Database Engine (Infrastructure)

**🔧 Scenario:**

You want to switch from SQL Server to **PostgreSQL** or **MongoDB**.

**🧩 What you change:**

Only BookRepository and ApplicationDbContext.

// Before (SQL Server - BookRepository.cs)

public async Task<Book?> GetByIdAsync(Guid id)

=> await \_context.Books.FindAsync(id);

// After (MongoDB - BookRepositoryMongo.cs)

public async Task<Book?> GetByIdAsync(Guid id)

{

return await \_mongoCollection.Find(b => b.Id == id).FirstOrDefaultAsync();

}

**📦 No impact to:**

* Controllers
* Application layer
* Domain layer

You just change the implementation of IBookRepository in DI.

## ✅ 2. Modifying a Business Rule (Domain)

**🔧 Scenario:**

You want to enforce a new rule: a book title must include at least one verb.

**🧩 What you change:**

You move this logic into a Domain Service or Validator.

// Domain/Services/BookValidator.cs

public static class BookValidator

{

public static bool HasVerbInTitle(string title)

{

return title.Contains("Run") || title.Contains("Learn"); // simplified

}

}

Then you update the command handler or a domain rule enforcement call:

if (!BookValidator.HasVerbInTitle(request.Title))

throw new InvalidOperationException("Book title must contain a verb.");

**📦 No impact to:**

* EF Core
* HTTP controllers
* Repositories

All external layers still call the same interface and DTO.

## ✅ 3. Changing an API Contract (Presentation)

**🔧 Scenario:**

You want to rename Author to Writer in your public API.

**🧩 What you change:**

Update only the BookDto and controller:

public class BookDto

{

public Guid Id { get; set; }

public string Title { get; set; }

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

}

And map the new property:

return new BookDto

{

Id = book.Id,

Title = book.Title,

Writer = book.Author // internal domain model unchanged

};

**📦 No impact to:**

* Domain model
* Repository
* EF Core
* Application logic

You're only modifying the external contract.

## ✅ 4. Replacing MediatR with a Native Pipeline (Application)

**🔧 Scenario:**

You want to remove MediatR to reduce dependency overhead.

**🧩 What you change:**

* Convert command/query handlers into services implementing interfaces.
* Inject those services directly into controllers.

public interface ICreateBookService

{

Task<BookDto> CreateAsync(string title, string author);

}

Update controller:

public class BooksController : ControllerBase

{

private readonly ICreateBookService \_service;

public BooksController(ICreateBookService service) => \_service =\_

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

ChatGPT is still generating a response...