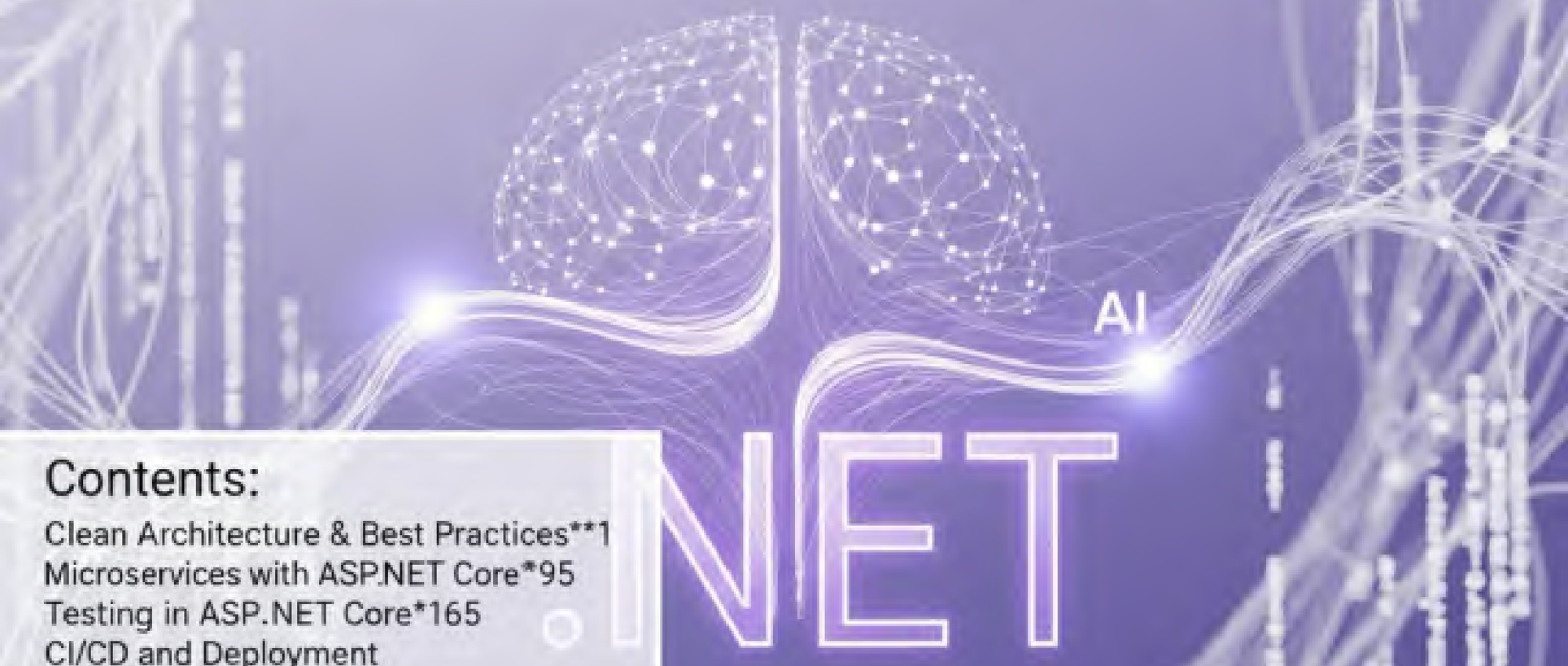
ASP.NET Core

Programming: Professional

(Integrative-Generative AI Edition)



Contents:

Author: Student Price Book Center

คำนำ

ในยุคที่การพัฒนาเว็บแอปพลิเคชันและระบบซอฟต์แวร์มีความซับซ้อนสูงขึ้นทุกวัน นักพัฒนาไม่เพียงต้องเข้าใจพื้นฐานการเขียนโปรแกรมและการสร้างฟังก์ชันการทำงาน แต่ยังจำเป็นต้องมีความรู้เชิงสถาปัตยกรรม (Architecture) การออกแบบระบบ (System Design) การจัดการวงจรชีวิตซอฟต์แวร์ (Software Lifecycle) ตลอดจนกระบวนการ Deploy และการดูแลรักษาระบบในระยะยาว หนังสือ **"ASP.NET Core Programming: Professional"** เล่มนี้ถูกเขียนขึ้นเพื่อตอบโจทย์ความต้องการดังกล่าว โดยเจาะลึกไปยังประเด็นที่มักจะเป็นความท้าทายของนักพัฒนามืออาชีพในการทำงานจริง

หนังสือเล่มนี้ต่อยอดจากความรู้พื้นฐานของ ASP.NET Core ไปสู่ระดับ **Professional** ผ่านการนำเสนอหัวข้อขั้นสูง ตั้งแต่ **Clean Architecture**, **Microservices**, **Testing**, **CI/CD and Deployment** จนถึง **Scaling และ Observability** แต่ละหัวข้อไม่เพียงอธิบายทฤษฎี หากยังนำเสนอด้วยโค้ดตัวอย่างที่สามารถนำไปใช้ได้จริง พร้อมโครงสร้างโปรเจกต์ที่ครบถ้วนและสามารถประยุกต์เข้ากับการพัฒนาในระดับองค์กร

บทที่ 16 ว่าด้วยเรื่อง *Clean Architecture & Best Practices* ซึ่งถือเป็นหัวใจสำคัญของการสร้างซอฟต์แวร์ที่มีความยืดหยุ่นสูง แยกความรับผิดชอบออกเป็นชั้น (Layer) อย่างชัดเจน ได้แก่ Presentation, Application, Infrastructure และ Domain หนังสือได้ยกตัวอย่างโปรเจกต์ *ProductApp* ซึ่งนำเสนอการใช้ Repository Pattern, Unit of Work และ Service Layer อย่างครบวงจร เพื่อนักพัฒนาสามารถนำไปเป็นโครงสร้างต้นแบบได้ทันที

บทที่ 17 มุ่งไปสู่ *Microservices with ASP.NET Core* อธิบายการสร้างระบบบริการย่อย (Microservices) ที่สามารถสื่อสารกันได้อย่างมีประสิทธิภาพ ทั้งผ่าน REST และ gRPC พร้อมอธิบาย Service Discovery และการใช้ API Gateway เช่น Ocelot และ YARP เนื้อหานี้ช่วยให้นักพัฒนามีความเข้าใจเชิงระบบ สามารถออกแบบแอปพลิเคชันที่รองรับการขยายตัวและการทำงานแบบกระจายตัว (Distributed System) ได้อย่างมั่นใจ

บทที่ 18 เจาะลึก *Testing in ASP.NET Core* เพราะคุณภาพของซอฟต์แวร์ไม่ได้วัดเพียงการทำงานได้จริง แต่ต้องตรวจสอบได้ในทุกระดับ ตั้งแต่ Unit Test, Integration Test ไปจนถึง Mocking Services ด้วย Moq และการทดสอบ API ผ่าน Postman และ Swagger โดยหนังสือได้นำเสนอกรณีศึกษาเชิงบูรณาการที่สามารถปรับใช้กับโปรเจกต์จริงได้ทันที

บทที่ 19 กล่าวถึง *CI/CD and Deployment* ซึ่งเป็นหัวใจของการนำซอฟต์แวร์ไปใช้งานจริงในองค์กรหรือบนระบบคลาวด์ เนื้อหาครอบคลุมการ Build และ Publish ด้วย dotnet publish, การ Deploy ไปยัง IIS, Linux, Docker และ Kubernetes ตลอดจนการจัดทำ CI/CD Pipeline ด้วย GitHub Actions และ Azure DevOps Pipeline ซึ่งช่วยยกระดับการพัฒนาให้เป็นระบบอัตโนมัติและลดความผิดพลาดในการ Deploy

บทที่ 20 ปิดท้ายด้วยหัวข้อ *Scaling and Observability* ซึ่งเป็นสิ่งที่องค์กรขนาดใหญ่ให้ความสำคัญ เนื้อหาครอบคลุม Load Balancing, Horizontal Scaling, Health Checks, Application Insights, Distributed Tracing ด้วย OpenTelemetry และ Logging แบบมืออาชีพผ่าน Serilog, Seq

และ ELK Stack หัวข้อนี้ทำให้นักพัฒนามีมุมมองครบถ้วนทั้งด้านการเพิ่มประสิทธิภาพและการติดตามระบบ

หนังสือเล่มนี้ไม่ได้เป็นเพียงคู่มือเชิงเทคนิค แต่ยังเป็น “เพื่อนร่วมทาง” ของนักพัฒนา ASP.NET Core ที่กำลังก้าวจากระดับกลางไปสู่ระดับมืออาชีพ ทุกบทถูกออกแบบให้ **อ่านเข้าใจง่าย → ทำตามได้จริง → ขยายผลสู่การใช้งานจริงในองค์กร** เหมาะสำหรับนักพัฒนา Software Engineer, Solution Architect, DevOps Engineer ตลอดจนผู้ที่สนใจสร้างระบบขนาดใหญ่ที่มีความยั่งยืน

ผู้เขียนเชื่อมั่นว่า หากผู้อ่านได้ศึกษาและฝึกปฏิบัติตามเนื้อหาในหนังสือเล่มนี้จนจบ จะสามารถก้าวข้ามการเป็นเพียงนักพัฒนา ไปสู่การเป็น **นักออกแบบสถาปัตยกรรมซอฟต์แวร์ (Software Architect)** ที่เข้าใจทั้งการสร้าง การทดสอบ การ Deploy และการดูแลระบบในโลกแห่งการทำงานจริงได้อย่างแท้จริง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 16

Clean Architecture & Best Practices

(Clean Architecture & Best Practices)

เนื้อหา

- Clean Architecture & Best Practices
- Clean Architecture & Best Practices (เชิงลึกจริงจัง)
- แยก Layer: Presentation, Application, Infrastructure, Domain ใน Clean Architecture (ASP.NET Core)
- โปรเจกต์แรก: ProductApp (Clean Architecture, 4 layers)
- Repository Pattern + Unit of Work
- Service Layer
- โปรเจกต์บูรณาการ

บทนำ: Clean Architecture & Best Practices

การพัฒนาแอปพลิเคชันที่ซับซ้อนต้องอาศัยสถาปัตยกรรมซอฟต์แวร์ที่ชัดเจนและยืดหยุ่น **Clean Architecture** เป็นแนวทางที่ช่วยให้โค้ดมีความ maintainable, testable, และสามารถขยายฟีเจอร์ใหม่ได้โดยไม่กระทบส่วนอื่นของระบบ บทนี้มุ่งเน้นการทำความเข้าใจโครงสร้างและแนวปฏิบัติที่ดีที่สุดสำหรับนักพัฒนา ASP.NET Core

การแยกชั้นของระบบเป็นหัวใจสำคัญของ Clean Architecture โดยแนะนำ **Layer** ต่าง ๆ ได้แก่ **Presentation, Application, Infrastructure, และ Domain** การแยกชั้นช่วยให้แต่ละส่วนของระบบมีความรับผิดชอบชัดเจน ลดความซับซ้อน และทำให้สามารถปรับปรุงหรือเปลี่ยนแปลงแต่ละชั้นได้โดยไม่กระทบส่วนอื่น

ในส่วนของ **Domain Layer** จะเน้นโมเดลและ business logic ที่เป็นหัวใจของระบบ ส่วน **Application Layer** จะทำหน้าที่ orchestration ของฟีเจอร์และจัดการ workflow ของระบบ ขณะที่ **Infrastructure Layer** ดูแลการเข้าถึงข้อมูล, การสื่อสารกับ API, และการเชื่อมต่อบริการภายนอก **Presentation Layer** จะรับผิดชอบด้าน UI, API endpoints หรือ interface ที่ผู้ใช้โต้ตอบ

บทนี้ยังครอบคลุมการใช้ **Repository Pattern** และ **Unit of Work** เพื่อจัดการการเข้าถึงข้อมูลอย่างเป็นระบบ Repository ช่วยแยกการเข้าถึงข้อมูลออกจาก business logic ทำให้โค้ดยืดหยุ่นและทดสอบได้ง่าย ส่วน Unit of Work ทำหน้าที่จัดการ transaction และความสอดคล้องของข้อมูล

นอกจากนี้ การออกแบบ **Service Layer** เป็นอีกหนึ่งแนวทางสำคัญที่ช่วยให้การประมวลผลทางธุรกิจเป็นระเบียบและสามารถทดสอบได้ง่าย นักพัฒนาจะได้เรียนรู้การสร้าง service ที่ encapsulate logic และ orchestrate interaction ระหว่าง repository และ domain entities

บทนี้ยังเน้น **Best Practices** เช่น การแยก responsibility, การเขียนโค้ดที่ testable, และการทำให้ระบบมี modularity สูง เทคนิคเหล่านี้ช่วยให้นักพัฒนาสามารถสร้างแอปพลิเคชันที่ maintainable และปรับตัวต่อความต้องการใหม่ ๆ ได้อย่างรวดเร็ว

ท้ายที่สุด ความรู้ในบทนี้ช่วยให้นักพัฒนาสามารถออกแบบระบบ ASP.NET Core ที่มี **โครงสร้างชัดเจน, รองรับการทดสอบและขยายฟีเจอร์, และเป็นไปตามมาตรฐาน Clean Architecture** ทำให้ซอฟต์แวร์มีคุณภาพสูงและพร้อมใช้งานในงานจริง

Clean Architecture & Best Practices

- แยก Layer: Presentation, Application, Infrastructure, Domain
- การใช้ Repository Pattern, Unit of Work
- การออกแบบ Service Layer

Clean Architecture & Best Practices

☐ **1. การแยก Layer: Presentation, Application, Infrastructure, Domain**

Clean Architecture (Uncle Bob) นิยมใช้หลักการ "Dependency Inversion" ⟶ Core (Domain, Application) **ไม่ขึ้นตรงกับ Infrastructure**

☐ โครงสร้างโปรเจกต์ที่นิยม

```
MyApp/
 ├── MyApp.Api/              ⟶ Presentation Layer (Controllers, Middleware, Filters)
 ├── MyApp.Application/      ⟶ Application Layer (Use Cases, Service Interfaces, DTOs)
 ├── MyApp.Domain/           ⟶ Domain Layer (Entities, Enums, Core Business Rules)
 ├── MyApp.Infrastructure/   ⟶ Infrastructure Layer (EF Core, Repositories, External APIs)
 └── MyApp.Tests/            ⟶ Unit Tests
```

- **Domain Layer** ⟶ ไม่ขึ้นกับ Framework/DB ใด ๆ (POCO Classes, Business Rules)
- **Application Layer** ⟶ Contains **Use Cases**, Service Interfaces, DTOs
- **Infrastructure Layer** ⟶ Implements Repository, External Services, EF Core, File Storage
- **Presentation Layer** ⟶ ASP.NET Core (Controllers, API Endpoints)

☐ **2. การใช้ Repository Pattern, Unit of Work**

Repository Pattern

- ทำหน้าที่เป็น abstraction ระหว่าง **Domain/Application** และ **Infrastructure**
- ลดการผูกติดกับ EF Core โดยตรง

```
// Domain Layer
public interface IGenericRepository<T> where T : class
{
    Task<T?> GetByIdAsync(int id);
    Task<IEnumerable<T>> GetAllAsync();
    Task AddAsync(T entity);
    void Update(T entity);
    void Delete(T entity);
}
// Infrastructure Layer (EF Core)
public class GenericRepository<T> : IGenericRepository<T> where T : class
{
    private readonly AppDbContext _context;
    public GenericRepository(AppDbContext context) => _context = context;

    public async Task<T?> GetByIdAsync(int id) => await _context.Set<T>().FindAsync(id);
    public async Task<IEnumerable<T>> GetAllAsync() => await
_context.Set<T>().ToListAsync();
    public async Task AddAsync(T entity) => await _context.Set<T>().AddAsync(entity);
    public void Update(T entity) => _context.Set<T>().Update(entity);
    public void Delete(T entity) => _context.Set<T>().Remove(entity);
}
```

Unit of Work Pattern

- รวมหลาย repository ไว้ด้วยกัน
- ช่วยจัดการ transaction

```
public interface IUnitOfWork : IDisposable
{
    IGenericRepository<Product> Products { get; }
    IGenericRepository<Order> Orders { get; }
    Task<int> SaveAsync();
}
```

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;
    public IGenericRepository<Product> Products { get; }
    public IGenericRepository<Order> Orders { get; }

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
        Products = new GenericRepository<Product>(_context);
        Orders = new GenericRepository<Order>(_context);
    }

    public async Task<int> SaveAsync() => await _context.SaveChangesAsync();
    public void Dispose() => _context.Dispose();
}
```

☐ 3. การออกแบบ Service Layer

Service Layer

- อยู่ใน **Application Layer**
- ใช้ประสานงาน Use Case
- ไม่ผูกกับ EF Core ตรง ๆ ⟶ ใช้ผ่าน Repository / Unit of Work

```
public interface IOrderService
{
    Task CreateOrderAsync(int productId, int quantity);
    Task<IEnumerable<Order>> GetAllOrdersAsync();
}
public class OrderService : IOrderService
{
    private readonly IUnitOfWork _uow;

    public OrderService(IUnitOfWork uow) => _uow = uow;

    public async Task CreateOrderAsync(int productId, int quantity)
```

```
    {
        var product = await _uow.Products.GetByIdAsync(productId);
        if (product == null) throw new Exception("Product not found");

        var order = new Order { ProductId = productId, Quantity = quantity, CreatedAt =
DateTime.UtcNow };
        await _uow.Orders.AddAsync(order);
        await _uow.SaveAsync();
    }

    public async Task<IEnumerable<Order>> GetAllOrdersAsync() => await
_uow.Orders.GetAllAsync();
}
```

☐ **Best Practices สรุป**

1. **แยก Layer ชัดเจน** ⟶ ไม่เขียน EF Core ใน Controller
2. **ใช้ Repository + Unit of Work** ⟶ Abstract DB access
3. **Service Layer** ⟶ ใช้สำหรับ Business Logic (ไม่ใส่ใน Controller)
4. **DTO / ViewModel** ⟶ อย่าส่ง Entity กลับไปตรง ๆ
5. **Dependency Injection (DI)** ⟶ ทำทุกอย่างเป็น Interface-based
6. **Validation** ⟶ ใช้ FluentValidation / Data Annotations ใน Application Layer
7. **Transaction** ⟶ ใช้ Unit of Work
8. **Testing** ⟶ Unit Test Service/Domain โดยไม่ต้องพึ่ง Database จริง

Clean Architecture & Best Practices (เชิงลึกจริงจัง)

Clean Architecture & Best Practices (เชิงลึกจริงจัง) สำหรับ ASP.NET Core โดยโฟกัส 3 หัวข้อที่คุณระบุ:

- การแยกเลเยอร์ **Presentation / Application / Infrastructure / Domain**
- **Repository Pattern + Unit of Work** (ข้อดี/ข้อควรระวัง + ตัวอย่างโค้ด)
- การออกแบบ **Service Layer** (Use Case-centric, Transaction, Validation, Mapping, CQRS/Specification)

ด้านล่างนี้เป็นทั้งแนวคิด โฟลว์ การตัดสินใจเชิงสถาปัตย์ และตัวอย่างโค้ดประกอบที่ “ต่อให้ทีมใหญ่ก็ใช้ได้จริง”

1) การแยก Layer: Presentation, Application, Infrastructure, Domain

1.1 โครงสร้างโฟลเดอร์ (ที่ใช้ได้จริง)

MyShop/

```
├── MyShop.Api/              // Presentation: Controllers, Filters, Auth, Swagger
├── MyShop.Application/      // Use Cases (Commands/Queries), DTOs, Validators, Interfaces
├── MyShop.Domain/           // Entities, Value Objects, Domain Events, Aggregates, Rules
├── MyShop.Infrastructure/   // EF Core, Repositories, UoW, External Services, Outbox
└── MyShop.Tests/            // Unit & Integration Tests
```

1.2 กฎ Dependency (หัวใจของ Clean Architecture)

- **Domain**: หัวใจธุรกิจ ⟶ *ไม่อ้างอิง framework/EF* ใด ๆ
- **Application**: ใช้ **Interfaces** เพื่อเรียก Infrastructure ผ่าน DI (ไม่อ้างอิง implementation)
- **Infrastructure**: อ้างอิง Application/Domain เพื่อ “implement” interfaces (เช่น Repository)
- **Presentation**: เรียก Application (Use Cases) เท่านั้น (อย่า EF ใน Controller)

หลักการ: **Dependencies ชี้เข้า Core** (Domain, Application) เสมอ

1.3 Data Flow (อ่านแล้วสร้างภาพได้ทันที)

HTTP Request ⟶ Controller (API)

⟶ สร้าง/แม็พ **DTO** ⟶ เรียก **Use Case** (Service/Application Handler)

⟶ ใช้ **Repositories/UoW** ทำงานกับ DB

⟶ โยน **Domain Events** (ถ้ามี)

⟶ Commit Transaction (**UoW.SaveAsync**)

⟶ แม็พกลับ **Response DTO** ⟶ ส่งออก

---

2) Domain (Core Business)

2.1 องค์ประกอบ

- **Entity**: มี Identity, มี business invariants
- **Value Object**: ไม่มี identity (เช่น Money, Email)
- **Aggregate Root**: ขอบเขต transaction consistency (เช่น Order รวม Items)
- **Domain Event**: สิ่งที่เกิดขึ้นใน domain (เช่น OrderPlaced)

ตัวอย่าง (ย่อ) – Domain Entity + ValueObject + Event

```
// MyShop.Domain/ValueObjects/Money.cs
public record Money(decimal Amount, string Currency)
{
```

```
    public static Money operator *(Money m, int qty) => new(m.Amount * qty, m.Currency);
}

// MyShop.Domain/Entities/Product.cs
public class Product
{
    public int Id { get; private set; }
    public string Name { get; private set; }
    public Money Price { get; private set; }
    public int Stock { get; private set; }

    private Product() { } // EF
    public Product(string name, Money price, int stock)
    {
        if (string.IsNullOrWhiteSpace(name)) throw new DomainException("Product name
required");
        if (stock < 0) throw new DomainException("Stock cannot be negative");
        Name = name;
        Price = price;
        Stock = stock;
    }

    public void ReduceStock(int qty)
    {
        if (qty <= 0) throw new DomainException("Qty must be positive");
        if (Stock < qty) throw new DomainException("Insufficient stock");
        Stock -= qty;
    }
}

// MyShop.Domain/Entities/Order.cs (Aggregate Root)
public class Order
{
    public int Id { get; private set; }
```

```
    public DateTime CreatedAt { get; private set; } = DateTime.UtcNow;
    private readonly List<OrderItem> _items = new();
    public IReadOnlyCollection<OrderItem> Items => _items.AsReadOnly();
    public Money Total => new(_items.Sum(i => i.Subtotal.Amount), "USD");

    private Order() { }
    public static Order Create() => new();

    public void AddItem(Product product, int qty)
    {
        if (product == null) throw new DomainException("Product required");
        product.ReduceStock(qty);
        _items.Add(new OrderItem(product.Id, product.Price, qty));
        // Domain event ตัวอย่าง: OrderItemAdded (ไม่จำเป็นต้องใส่ทุกเคส)
    }
}

public class OrderItem
{
    public int Id { get; private set; }
    public int ProductId { get; private set; }
    public Money Price { get; private set; }
    public int Quantity { get; private set; }
    public Money Subtotal => Price * Quantity;

    private OrderItem() { }
    public OrderItem(int productId, Money price, int quantity)
    {
        ProductId = productId;
        Price = price;
        Quantity = quantity;
    }
}
```

```
public class DomainException : Exception
{
    public DomainException(string message) : base(message) {}
}
```

---

3) Repository Pattern & Unit of Work (เชิงลึก)

3.1 ทำไมใช้ Repository?

- แยก Business Logic ออกจาก ORM/EF ⟶ **ทดสอบง่าย**
- ควบคุม query ที่ซับซ้อน และซ่อนรายละเอียด persistence
- แต่… **อย่า wrapper EF แบบ 1:1** เกินเหตุ (anti-pattern)

3.2 Interface ที่ "พอดี"

- แนะนำ **Repository ต่อ Aggregate Root** (ไม่ generic อะลัยๆไปหมด)
- เพิ่ม **เมธอดเฉพาะ domain** (เช่น GetBySkuAsync, GetTopSellingAsync)
- ใช้ **Specification Pattern** สำหรับ query ที่ประกอบได้

ตัวอย่าง – Repository Interface (ต่อ Aggregate)

```
// MyShop.Application/Abstractions/IProductRepository.cs
public interface IProductRepository
{
    Task<Product?> GetByIdAsync(int id, CancellationToken ct = default);
    Task AddAsync(Product product, CancellationToken ct = default);
    Task<bool> ExistsAsync(int id, CancellationToken ct = default);
    Task<List<Product>> SearchAsync(ProductSpec spec, CancellationToken ct = default);
}

// ตัวอย่าง Spec (ประกอบ query แบบปลอดภัย/ทดสอบง่าย)
public record ProductSpec(string? NameContains = null, int? MinStock = null, int? MaxStock =
null);
```

ตัวอย่าง – Unit of Work

```
// MyShop.Application/Abstractions/IUnitOfWork.cs
public interface IUnitOfWork
{
    Task<int> SaveChangesAsync(CancellationToken ct = default);
}
```

ตัวอย่าง – EF Core Implementation

```
// MyShop.Infrastructure/Repositories/ProductRepository.cs
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _db;
    public ProductRepository(AppDbContext db) => _db = db;

    public async Task<Product?> GetByIdAsync(int id, CancellationToken ct = default) =>
        await _db.Products.FirstOrDefaultAsync(x => x.Id == id, ct);

    public async Task AddAsync(Product product, CancellationToken ct = default) =>
        await _db.Products.AddAsync(product, ct);

    public async Task<bool> ExistsAsync(int id, CancellationToken ct = default) =>
        await _db.Products.AnyAsync(x => x.Id == id, ct);

    public async Task<List<Product>> SearchAsync(ProductSpec spec, CancellationToken ct =
default)
    {
        IQueryable<Product> q = _db.Products.AsQueryable();
        if (!string.IsNullOrWhiteSpace(spec.NameContains))
            q = q.Where(p => p.Name.Contains(spec.NameContains));
        if (spec.MinStock.HasValue)
            q = q.Where(p => p.Stock >= spec.MinStock.Value);
        if (spec.MaxStock.HasValue)
            q = q.Where(p => p.Stock <= spec.MaxStock.Value);

        return await q.ToListAsync(ct);
    }
}

// MyShop.Infrastructure/Persistence/UnitOfWork.cs
public class UnitOfWork : IUnitOfWork
{
```

```
    private readonly AppDbContext _db;
    public UnitOfWork(AppDbContext db) => _db = db;
    public Task<int> SaveChangesAsync(CancellationToken ct = default) =>
_db.SaveChangesAsync(ct);
}
```

หมายเหตุ: EF Core เองมี “implicit unit of work” ผ่าน DbContext. การมี IUnitOfWork แยกทำให้เราควบคุม boundary/transaction และทดสอบง่าย

3.3 ข้อควรระวัง (สำคัญ)

- **อย่าทำ Generic Repository** ที่มีแต่ Add/Get/Update/Delete แล้วบังคับทุกอย่างผ่านมัน—มักจะลงท้ายด้วย query กระจัดกระจายที่แก้ยาก
- **DbContext Lifetime**: ใช้ **Scoped** ต่อ request. อย่าทำ Singleton
- **Transaction Boundary**: กำหนดให้ชัดว่า Use Case ไหนบ้างที่ commit พร้อมกัน
- **Optimistic Concurrency**: ใช้ RowVersion/Timestamp ป้องกันทับข้อมูล

ตัวอย่าง Concurrency Token:

```
public class Product
{
    public int Id { get; private set; }
    public string Name { get; private set; } = "";
    public int Stock { get; private set; }
    public byte[] RowVersion { get; private set; } = Array.Empty<byte>(); // <-- Concurrency

    // ...
}

// OnModelCreating
builder.Entity<Product>()
       .Property(p => p.RowVersion)
       .IsRowVersion();
```

---

4) Service Layer (Use Cases) – ออกแบบอย่างไรให้แข็งแรง

4.1 หลักคิด

- 1 Use Case = 1 เมธอด/Handler (ชัดเจน, testable, transactional)
- ไม่ทำงาน EF ใน Controller ⟶ ให้มาที่นี่
- จัดการ **Validation, Transaction, Domain Events, Mapping, Caching**, **Observability**

4.2 ตัวอย่าง Use Case: PlaceOrder

Request/Response DTO (แยกจาก Domain)

```
// MyShop.Application/Orders/PlaceOrder.cs
public record PlaceOrderCommand(int ProductId, int Quantity);
public record PlaceOrderResult(int OrderId, decimal TotalAmount, string Currency);
```

Validator (FluentValidation หรือ DataAnnotations)

```
public class PlaceOrderValidator : AbstractValidator<PlaceOrderCommand>
{
    public PlaceOrderValidator()
    {
        RuleFor(x => x.ProductId).GreaterThan(0);
        RuleFor(x => x.Quantity).GreaterThan(0);
    }
}
```

Handler/Service (Transaction + Domain)

```
public interface IOrderService
{
    Task<PlaceOrderResult> PlaceOrderAsync(PlaceOrderCommand cmd, CancellationToken ct
= default);
}

public class OrderService : IOrderService
{
    private readonly IProductRepository _products;
    private readonly IOrderRepository _orders;  // เหมือนกับ ProductRepository (ต่อ Aggregate)
    private readonly IUnitOfWork _uow;
    private readonly PlaceOrderValidator _validator = new();

    public OrderService(IProductRepository products, IOrderRepository orders, IUnitOfWork uow)
    {
        _products = products;
        _orders = orders;
        _uow = uow;
    }
```

```
    public async Task<PlaceOrderResult> PlaceOrderAsync(PlaceOrderCommand cmd,
CancellationToken ct = default)
    {
        await _validator.ValidateAndThrowAsync(cmd, ct);

        var product = await _products.GetByIdAsync(cmd.ProductId, ct)
                    ?? throw new DomainException("Product not found");

        var order = Order.Create();
        order.AddItem(product, cmd.Quantity); // ตัดสต๊อกที่ Domain

        await _orders.AddAsync(order, ct);
        await _uow.SaveChangesAsync(ct); // Transaction boundary

        return new PlaceOrderResult(order.Id, order.Total.Amount, order.Total.Currency);
    }
}
```

Controller (บาง เบา เรียก Use Case อย่างเดียว)

```
// MyShop.Api/Controllers/OrdersController.cs
[ApiController]
[Route("api/orders")]
public class OrdersController : ControllerBase
{
    private readonly IOrderService _svc;
    public OrdersController(IOrderService svc) => _svc = svc;

    [HttpPost]
    public async Task<ActionResult<PlaceOrderResult>> Place([FromBody]
PlaceOrderCommand cmd, CancellationToken ct)
        => Ok(await _svc.PlaceOrderAsync(cmd, ct));
}
```

---

5) CQRS & Specification (เมื่อ Query/Command ซับซ้อน)

- **CQRS**: แยก Command (เปลี่ยน state) กับ Query (อ่านอย่างเดียว)
- **ข้อดี**: โค้ดชัดเจน, scale/query tuning แยกกัน, ลด over-fetching
- **Specification Pattern**: ห่อ logic เงื่อนไข query ให้นำกลับมาใช้ซ้ำได้

ตัวอย่าง Query แยกไฟล์:

```
public record GetProductsQuery(string? NameContains = null, int? MinStock = null);
public record ProductDto(int Id, string Name, int Stock);

public interface IProductQueries
{
    Task<List<ProductDto>> GetAsync(GetProductsQuery query, CancellationToken ct =
default);
}

public class ProductQueries : IProductQueries
{
    private readonly AppDbContext _db;
    public ProductQueries(AppDbContext db) => _db = db;

    public async Task<List<ProductDto>> GetAsync(GetProductsQuery q, CancellationToken ct
= default)
    {
        IQueryable<Product> query = _db.Products.AsNoTracking();

        if (!string.IsNullOrWhiteSpace(q.NameContains))
            query = query.Where(p => p.Name.Contains(q.NameContains));
        if (q.MinStock.HasValue)
            query = query.Where(p => p.Stock >= q.MinStock);

        return await query
            .Select(p => new ProductDto(p.Id, p.Name, p.Stock))
            .ToListAsync(ct);
    }
}
```

หมายเหตุ: **Application Layer** สามารถมี “Query Services” ที่อ่านผ่าน DbContext โดย **AsNoTracking** และไม่แตะ domain (common practice สำหรับ read models)

---

6) Transaction, Outbox, และ Integration

- **Transaction Boundary**: กำหนดชัดที่ Use Case/Service
- **Outbox Pattern**: เมื่อ publish event ออกไปยัง message broker (เช่น RabbitMQ/Kafka) ⟶ บันทึก event ในตาราง “Outbox” ใน transaction เดียวกับ DB แล้ว background processor จึงส่งออก (ป้องกัน lost/mismatch)
- **Retry/Resiliency**: ใช้ **Polly** (retry, circuit breaker) ใน Infrastructure เวลาเรียก external services

---

7) Mapping, Validation, and Security

- **DTO ↔ Domain**: ใช้ **AutoMapper** (หรือ manual mapping สำหรับ use case ที่สำคัญ)
- **Validation**: Application layer (FluentValidation) + Validation Pipeline
- **Security**: อย่าปล่อย Entity กลับ API, ป้องกัน over-posting, ใช้ **model binding whitelist** (BindProperty/FromBody DTO), ปิดการ track entity ที่ไม่จำเป็น

---

8) Performance & Observability (ที่ควรมีใน Best Practices)

- **EF Core**: ใช้ AsNoTracking() สำหรับ read-only, SplitQuery() กรณี include เยอะ, ระวัง N+1
- **Caching**: Query Layer ใส่ IMemoryCache/IDistributedCache, เคลียร์ cache หลัง command สำเร็จ
- **Logging**: Structured logging (Serilog), CorrelationId, Request/Response logging (ระวัง PII)
- **Metrics/Tracing**: OpenTelemetry (OTLP), trace ระดับ Use Case/Repository
- **Paging**: เสมอ—อย่า ToListAsync() ทั้งตาราง

---

9) DI Registration (แนะนำแบบชัด ๆ)

```
// Program.cs (MyShop.Api)
builder.Services.AddControllers();

// Application
builder.Services.AddScoped<IOrderService, OrderService>();
```

```
builder.Services.AddScoped<IProductQueries, ProductQueries>();

// Infrastructure
builder.Services.AddDbContext<AppDbContext>(opt =>
    opt.UseSqlServer(builder.Configuration.GetConnectionString("Default")));
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
```

---

10) Testing Strategy (ที่ทีมใหญ่ใช้)

- **Unit Test**: Domain (Entities/ValueObjects/Rules) + Application Services (mock repo)
- **Integration Test**: Repository จริงกับ **Testcontainer** (SQL Server/Postgres)
- **Contract Test**: API กับ client (e.g. Pact)
- **E2E/Smoke**: Minimal flow ผ่าน Use Case สำคัญ

สรุป Decision Guide (Cheat Sheet)

- **Controller**: บางที่สุด ⟶ เรียก **Use Case** ใน **Application**
- **Application**: จัด **Transaction**, **Validation**, **Mapping**, **Domain Rules**
- **Domain**: กฎธุรกิจบริสุทธิ์, ไม่ผูก EF/Framework
- **Infrastructure**: จัดการ persistence/IO จริง (EF, External APIs, Outbox)
- **Repository**: ต่อ Aggregate, เมธอดหมายถึง “business intent” มากกว่า CRUD ดิบ
- **UoW**: รวมหลาย repo + บังคับ boundary ของ commit
- **Query**: แยก read model (CQRS), ใช้ AsNoTracking, ทำ Paging + Caching
- **Concurrency**: ใช้ RowVersion/Optimistic Concurrency
- **Observability**: Logging + Metrics + Tracing ครบ
- **Security**: DTO-only, Validation, Over-posting protection, ไม่ leak entity

---

แยก Layer: Presentation, Application, Infrastructure, Domain ใน Clean Architecture (ASP.NET Core)

---

1) แนวคิดหลักของการแยก Layer

Clean Architecture เน้น **Separation of Concerns** และ **Dependency Rule**

□ **กฎทอง (Dependency Rule):**

- **Presentation** ⟶ เรียกใช้ **Application** เท่านั้น
- **Application** ⟶ เรียกใช้ **Domain** และ **Interfaces** (ไม่รู้จัก EF หรือ Framework)
- **Infrastructure** ⟶ Implement Interfaces ของ Application และอ้างอิง Framework/DB
- **Domain** ⟶ หัวใจธุรกิจ ไม่อ้างอิงใครเลย

---

2) โครงสร้างโฟลเดอร์

```
MyShop/
  ├── MyShop.Api/              // Presentation Layer (Controllers, Filters, Middleware)
  ├── MyShop.Application/      // Application Layer (Use Cases, DTOs, Interfaces, Validators)
  ├── MyShop.Domain/           // Domain Layer (Entities, Value Objects, Business Rules, Events)
  ├── MyShop.Infrastructure/   // Infrastructure Layer (EF Core, Repositories, External Services)
  └── MyShop.Tests/            // Unit & Integration Tests
```

---

3) ตัวอย่างโค้ดแต่ละ Layer

3.1 Domain Layer (MyShop.Domain)

```
// Entities/Product.cs
namespace MyShop.Domain.Entities;

public class Product
{
    public int Id { get; private set; }
    public string Name { get; private set; }
    public int Stock { get; private set; }

    private Product() { } // EF Core ใช้

    public Product(string name, int stock)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Product name is required");
        if (stock < 0)
            throw new ArgumentException("Stock cannot be negative");
```

```
        Name = name;
        Stock = stock;
    }

    public void ReduceStock(int qty)
    {
        if (qty <= 0) throw new InvalidOperationException("Invalid qty");
        if (Stock < qty) throw new InvalidOperationException("Not enough stock");
        Stock -= qty;
    }
}
```

3.2 Application Layer (MyShop.Application)

```
// Interfaces/IProductRepository.cs
using MyShop.Domain.Entities;

namespace MyShop.Application.Interfaces;

public interface IProductRepository
{
    Task<Product?> GetByIdAsync(int id, CancellationToken ct = default);
    Task AddAsync(Product product, CancellationToken ct = default);
    Task SaveChangesAsync(CancellationToken ct = default);
}

// UseCases/CreateProduct/CreateProductCommand.cs
namespace MyShop.Application.UseCases.CreateProduct;

public record CreateProductCommand(string Name, int Stock);
public record CreateProductResult(int Id, string Name, int Stock);

// UseCases/CreateProduct/CreateProductHandler.cs
using MyShop.Application.Interfaces;
using MyShop.Domain.Entities;
```

```
namespace MyShop.Application.UseCases.CreateProduct;

public class CreateProductHandler
{
    private readonly IProductRepository _repo;
    public CreateProductHandler(IProductRepository repo) => _repo = repo;

    public async Task<CreateProductResult> Handle(CreateProductCommand cmd,
CancellationToken ct = default)
    {
        var product = new Product(cmd.Name, cmd.Stock);
        await _repo.AddAsync(product, ct);
        await _repo.SaveChangesAsync(ct);

        return new CreateProductResult(product.Id, product.Name, product.Stock);
    }
}
```

3.3 Infrastructure Layer (MyShop.Infrastructure)

```
// Persistence/AppDbContext.cs
using Microsoft.EntityFrameworkCore;
using MyShop.Domain.Entities;

namespace MyShop.Infrastructure.Persistence;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) {}
    public DbSet<Product> Products => Set<Product>();
}

// Repositories/ProductRepository.cs
using Microsoft.EntityFrameworkCore;
using MyShop.Application.Interfaces;
```

```
using MyShop.Domain.Entities;

namespace MyShop.Infrastructure.Repositories;

public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _db;
    public ProductRepository(AppDbContext db) => _db = db;

    public async Task<Product?> GetByIdAsync(int id, CancellationToken ct = default) =>
        await _db.Products.FirstOrDefaultAsync(p => p.Id == id, ct);

    public async Task AddAsync(Product product, CancellationToken ct = default) =>
        await _db.Products.AddAsync(product, ct);

    public async Task SaveChangesAsync(CancellationToken ct = default) =>
        await _db.SaveChangesAsync(ct);
}
```

3.4 Presentation Layer (MyShop.Api)

```
// Controllers/ProductsController.cs
using Microsoft.AspNetCore.Mvc;
using MyShop.Application.UseCases.CreateProduct;

namespace MyShop.Api.Controllers;

[ApiController]
[Route("api/products")]
public class ProductsController : ControllerBase
{
    private readonly CreateProductHandler _handler;
    public ProductsController(CreateProductHandler handler) => _handler = handler;

    [HttpPost]
```

```
    public async Task<ActionResult<CreateProductResult>> Create(CreateProductCommand
cmd, CancellationToken ct)
    {
        var result = await _handler.Handle(cmd, ct);
        return Ok(result);
    }
}
```

3.5 Program.cs (DI Wiring)

```
var builder = WebApplication.CreateBuilder(args);

// Infrastructure
builder.Services.AddDbContext<AppDbContext>(opt =>
    opt.UseInMemoryDatabase("MyShopDb"));
builder.Services.AddScoped<IProductRepository, ProductRepository>();

// Application
builder.Services.AddScoped<CreateProductHandler>();

builder.Services.AddControllers();

var app = builder.Build();
app.MapControllers();
app.Run();
```

---

4) การรันจริง (ผลลัพธ์)

Request

```
POST /api/products
Content-Type: application/json

{
  "name": "Laptop",
  "stock": 5
}
```

Response

```
{
  "id": 1,
  "name": "Laptop",
  "stock": 5
}
```

☐ จุดสังเกต:

- Controller ไม่รู้จัก EF ⟶ เรียก Use Case เท่านั้น
- Use Case ไม่รู้จัก EF ⟶ ใช้ Interface (IProductRepository)
- Repository อยู่ Infrastructure ⟶ อ้าง EF จริง
- Domain (Product) ไม่มี EF reference ใด ๆ

ตัวอย่างโปรเจกต์ 6 ตัว (3 พื้นฐาน + 3 แนวประยุกต์) ซึ่ง **โชว์การแยก Layer แบบ Clean Architecture** (Presentation / Application / Domain / Infrastructure) แต่ละตัวเป็นชุดไฟล์ที่ "รันได้จริง" (ใช้ InMemory EF Core เพื่อให้ทดสอบได้ง่าย) — พร้อม **โครงสร้าง**, **ไฟล์โค้ดเต็ม**, **คำอธิบายโค้ดสำคัญ** และ **ตัวอย่างผลการรัน (curl + JSON response)**

หมายเหตุ: ทุกตัวอย่างใช้ .NET 7/8 style WebApplication/minimal DI แต่คงโครงสร้างแนว Clean Architecture แยกโฟลเดอร์ตาม Layer เพื่อให้อ่านง่ายและนำไปต่อยอดได้จริง

ตัวอย่างพื้นฐาน 1 — Product CRUD (Basic Create / Read)

โครงสร้าง

```
ProductApp/
├── Program.cs
├── Domain/
│   └── Entities/Product.cs
├── Application/
│   ├── Interfaces/IProductRepository.cs
│   └── UseCases/CreateProduct/CreateProductHandler.cs
├── Infrastructure/
│   ├── Persistence/AppDbContext.cs
│   └── Repositories/ProductRepository.cs
└── Presentation/
    └── Controllers/ProductsController.cs
```

Program.cs

```
using Microsoft.EntityFrameworkCore;
using ProductApp.Application.Interfaces;
using ProductApp.Infrastructure.Persistence;
using ProductApp.Infrastructure.Repositories;
using ProductApp.Application.UseCases.CreateProduct;

var builder = WebApplication.CreateBuilder(args);

// Infrastructure
builder.Services.AddDbContext<AppDbContext>(opt =>
opt.UseInMemoryDatabase("ProductDb"));
builder.Services.AddScoped<IProductRepository, ProductRepository>();

// Application (handlers)
builder.Services.AddScoped<CreateProductHandler>();

// Presentation
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();
```

Domain/Entities/Product.cs

```
namespace ProductApp.Domain.Entities;

public class Product
{
    public int Id { get; private set; }
    public string Name { get; private set; } = "";
    public decimal Price { get; private set; }
    public int Stock { get; private set; }

    private Product() { } // for EF
```

```
    public Product(string name, decimal price, int stock)
    {
        if(string.IsNullOrWhiteSpace(name)) throw new ArgumentException("Name required");
        if(price < 0) throw new ArgumentException("Price invalid");
        if(stock < 0) throw new ArgumentException("Stock invalid");

        Name = name;
        Price = price;
        Stock = stock;
    }
}
```

Application/Interfaces/IProductRepository.cs

```
using ProductApp.Domain.Entities;

namespace ProductApp.Application.Interfaces;
public interface IProductRepository
{
    Task<Product?> GetByIdAsync(int id, CancellationToken ct = default);
    Task AddAsync(Product product, CancellationToken ct = default);
    Task<List<Product>> GetAllAsync(CancellationToken ct = default);
    Task SaveChangesAsync(CancellationToken ct = default);
}
```

Application/UseCases/CreateProduct/CreateProductHandler.cs

```
using ProductApp.Application.Interfaces;
using ProductApp.Domain.Entities;

namespace ProductApp.Application.UseCases.CreateProduct;
public record CreateProductCommand(string Name, decimal Price, int Stock);
public record CreateProductResult(int Id, string Name, decimal Price, int Stock);

public class CreateProductHandler
{
    private readonly IProductRepository _repo;
    public CreateProductHandler(IProductRepository repo) => _repo = repo;
```

```
    public async Task<CreateProductResult> Handle(CreateProductCommand cmd,
CancellationToken ct = default)
    {
        var p = new Product(cmd.Name, cmd.Price, cmd.Stock);
        await _repo.AddAsync(p, ct);
        await _repo.SaveChangesAsync(ct);
        return new CreateProductResult(p.Id, p.Name, p.Price, p.Stock);
    }
}
```

Infrastructure/Persistence/AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using ProductApp.Domain.Entities;

namespace ProductApp.Infrastructure.Persistence;
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> opts) : base(opts) {}
    public DbSet<Product> Products => Set<Product>();
}
```

Infrastructure/Repositories/ProductRepository.cs

```
using Microsoft.EntityFrameworkCore;
using ProductApp.Application.Interfaces;
using ProductApp.Domain.Entities;
using ProductApp.Infrastructure.Persistence;

namespace ProductApp.Infrastructure.Repositories;
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _db;
    public ProductRepository(AppDbContext db) => _db = db;

    public async Task AddAsync(Product product, CancellationToken ct = default) => await
_db.Products.AddAsync(product, ct);
```

```
    public async Task<Product?> GetByIdAsync(int id, CancellationToken ct = default) => await
_db.Products.FirstOrDefaultAsync(p => p.Id == id, ct);
    public async Task<List<Product>> GetAllAsync(CancellationToken ct = default) => await
_db.Products.ToListAsync(ct);
    public async Task SaveChangesAsync(CancellationToken ct = default) => await
_db.SaveChangesAsync(ct);
}
```

Presentation/Controllers/ProductsController.cs

```
using Microsoft.AspNetCore.Mvc;
using ProductApp.Application.UseCases.CreateProduct;

namespace ProductApp.Presentation.Controllers;
[ApiController]
[Route("api/products")]
public class ProductsController : ControllerBase
{
    private readonly CreateProductHandler _createHandler;
    private readonly ProductApp.Application.Interfaces.IProductRepository _repo;
    public ProductsController(CreateProductHandler createHandler,
ProductApp.Application.Interfaces.IProductRepository repo)
    {
        _createHandler = createHandler;
        _repo = repo;
    }

    [HttpPost]
    public async Task<IActionResult> Create(CreateProductCommand cmd, CancellationToken
ct)
    {
        var result = await _createHandler.Handle(cmd, ct);
        return CreatedAtAction(nameof(GetById), new { id = result.Id }, result);
    }

    [HttpGet("{id}")]
```

```
    public async Task<IActionResult> GetById(int id, CancellationToken ct)
    {
        var p = await _repo.GetByIdAsync(id, ct);
        if (p == null) return NotFound();
        return Ok(new { p.Id, p.Name, p.Price, p.Stock });
    }

    [HttpGet]
    public async Task<IActionResult> List(CancellationToken ct) => Ok(await
_repo.GetAllAsync(ct));
}
```

คำอธิบายสั้น ๆ

- **Domain**: Product เป็น POCO ไม่มี reference กับ EF/Framework
- **Application**: มี interface repository + handler (CreateProductHandler) รับผิดชอบ use-case
- **Infrastructure**: ให้ AppDbContext (EF InMemory) และ implement repository
- **Presentation**: Controller เรียกใช้ handler และ repository ผ่าน DI — ไม่มี EF code ตรงนี้

ตัวอย่างผลการรัน (curl)

1. สร้าง:

```
curl -X POST http://localhost:5000/api/products -H "Content-Type:application/json" -d
'{"name":"Laptop","price":1299.99,"stock":10}'
```

Response 201:

```
{ "id": 1, "name": "Laptop", "price": 1299.99, "stock": 10 }
```

2. ดึงรายการ:

```
curl http://localhost:5000/api/products
```

Response 200:

```
[ { "id":1, "name":"Laptop", "price":1299.99, "stock":10 } ]
```

ตัวอย่างพื้นฐาน 2 — Orders + Unit of Work (Transaction Boundary)

โครงสร้าง

```
OrderApp/
  ├── Program.cs
  ├── Domain/Entities/{Product.cs, Order.cs, OrderItem.cs}
  ├── Application/Interfaces/{IProductRepository.cs,IOrderRepository.cs,IUnitOfWork.cs}
```

```
├── Application/UseCases/PlaceOrder/PlaceOrderHandler.cs
├── Infrastructure/Persistence/AppDbContext.cs
├── Infrastructure/Repositories/{ProductRepository.cs,OrderRepository.cs}
└── Presentation/Controllers/OrdersController.cs
```

วอร์มย่อ (ไฟล์สำคัญ)

Application/Interfaces/IUnitOfWork.cs

```
public interface IUnitOfWork
{
    Task<int> SaveChangesAsync(CancellationToken ct = default);
}
```

Application/UseCases/PlaceOrder/PlaceOrderHandler.cs

```
using OrderApp.Application.Interfaces;
using OrderApp.Domain.Entities;

public record PlaceOrderCommand(int ProductId, int Quantity);
public record PlaceOrderResult(int OrderId, decimal Total);

public class PlaceOrderHandler
{
    private readonly IProductRepository _products;
    private readonly IOrderRepository _orders;
    private readonly IUnitOfWork _uow;

    public PlaceOrderHandler(IProductRepository products, IOrderRepository orders,
IUnitOfWork uow)
    {
        _products = products; _orders = orders; _uow = uow;
    }

    public async Task<PlaceOrderResult> Handle(PlaceOrderCommand cmd, CancellationToken
ct = default)
    {
        var prod = await _products.GetByIdAsync(cmd.ProductId, ct)
                ?? throw new Exception("Product not found");
```

```
        if (prod.Stock < cmd.Quantity) throw new Exception("Insufficient stock");

        var order = Order.Create();
        order.AddItem(prod.Id, prod.Price, cmd.Quantity);

        prod.ReduceStock(cmd.Quantity); // domain operation
        await _orders.AddAsync(order, ct);

        await _uow.SaveChangesAsync(ct); // commit both product stock change + order insert

        return new PlaceOrderResult(order.Id, order.Total.Amount);
    }
}
```

Infrastructure/Persistence/UnitOfWork.cs

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _db;
    public UnitOfWork(AppDbContext db) => _db = db;
    public Task<int> SaveChangesAsync(CancellationToken ct = default) =>
_db.SaveChangesAsync(ct);
}
```

Presentation/Controllers/OrdersController.cs

```
[ApiController]
[Route("api/orders")]
public class OrdersController : ControllerBase
{
    private readonly PlaceOrderHandler _handler;
    public OrdersController(PlaceOrderHandler handler) => _handler = handler;

    [HttpPost]
    public async Task<IActionResult> Place(PlaceOrderCommand cmd, CancellationToken ct)
    {
        var res = await _handler.Handle(cmd, ct);
```

```
        return Created("", res);
    }
}
```

คำอธิบาย

- ใช้ **UnitOfWork** เป็น transaction boundary ⟶ commit ครบทั้งการเปลี่ยนแปลง stock และการบันทึก order
- Domain logic เช่น ReduceStock อยู่ใน Product class (business rules อยู่ใน Domain)
- Controller เรียก Use Case เท่านั้น

ตัวอย่าง run

1. สร้าง Product (ตามตัวอย่างแรก) id=1 stock 10
2. Place order:

```
curl -X POST http://localhost:5000/api/orders -H "Content-Type:application/json" -d
'{"productId":1,"quantity":2}'
```

Response 201:

```
{"orderId":1,"total":2599.98}
```

3. ดึง product ⟶ stock เหลือ 8

ตัวอย่างพื้นฐาน 3 — Query Read Service (Query-only / AsNoTracking)

โครงสร้าง

```
QueryApp/
  ├── Program.cs
  ├── Domain/Entities/Product.cs
  ├── Application/Queries/IProductQueries.cs
  ├── Infrastructure/Persistence/AppDbContext.cs
  └── Presentation/Controllers/ProductsReadController.cs
```

Application/Queries/IProductQueries.cs

```
public record ProductDto(int Id, string Name, decimal Price, int Stock);
public record GetProductsQuery(string? NameContains, int Page=1, int PageSize=20);

public interface IProductQueries
{
    Task<List<ProductDto>> GetAsync(GetProductsQuery query, CancellationToken ct =
default);
}
```

Infrastructure Implementation (ProductQueries)

- Uses AsNoTracking() and projection to DTOs — **no domain entity modifications**

คำอธิบาย

- แยก Query services เพื่อ performance tuning (no tracking, optimized selects)
- Presentation calls IProductQueries, not repositories

ตัวอย่าง run

curl "http://localhost:5000/api/read/products?nameContains=lap"

Response 200 (list of ProductDto)

ตัวอย่างแนวประยุกต์ 1 — Inventory Reservation (Concurrency + RowVersion)

โครงสร้าง (ย่อ)

InventoryApp/

```
├── Domain/Entities/Product.cs (with byte[] RowVersion for concurrency)
├── Application/UseCases/ReserveStock/ReserveStockHandler.cs
├── Infrastructure/Persistence/AppDbContext.cs (configure IsRowVersion)
└── Presentation/Controllers/InventoryController.cs
```

ประเด็นสำคัญ

- Domain: product has RowVersion byte[] property
- Infrastructure: configure IsRowVersion() in EF fluent
- Use case: try update stock, catch DbUpdateConcurrencyException ⟶ surface as 409 Conflict

ตัวอย่างผลการรัน

1. Two concurrent requests try to reserve last unit ⟶ one succeeds (200), อีกอันได้ 409 Conflict with message "Concurrency conflict, try again"

ตัวอย่างแนวประยุกต์ 2 — CQRS Read Model with Cached Queries

โครงสร้าง (ย่อ)

CqrsApp/

```
├── Application/Commands/... (mutations use repositories + UoW)
├── Application/Queries/ProductReadModel.cs (optimized projection)
├── Infrastructure/Caching/MemoryCacheProductQueries.cs
└── Presentation/Controllers/ProductQueryController.cs
```

ประเด็นสำคัญ

- Read queries use IProductQueries implementation that caches result in IMemoryCache for short time
- Commands update DB and invalidate cache via interface ICacheInvalidator

ตัวอย่างผลการรัน

- First GET /api/query/products ⟶ from DB (200)
- Subsequent GET within cache window ⟶ from memory cache (200, faster)
- After product update (command), cache invalidated so next read reflects new state

---

ตัวอย่างแนวประยุกต์ 3 — Outbox Pattern (simulate) + Domain Events

โครงสร้าง (ย่อ)

OutboxApp/

├── Domain/Entities/Order.cs (raises DomainEvent OrderPlaced)

├── Application/UseCases/PlaceOrder -> writes order + outbox event record in same transaction

├── Infrastructure/Persistence/OutboxMessage.cs (table)

├── Infrastructure/Background/OutboxDispatcher (reads outbox, publishes to fake broker, marks sent)

└── Presentation/Controllers/OrdersController.cs

ประเด็นสำคัญ

- PlaceOrder use-case writes Order and inserts OutboxMessage in same DbContext transaction
- Background worker (hosted service) periodically reads unsent outbox records and "publishes" them (simulate via console) then marks them sent
- Guarantees reliable delivery without distributed transactions

ผลการรัน (simulation)

1. POST /api/orders ⟶ 201 Order placed
2. Background worker logs: Publishing Outbox message {id} payload... ⟶ marks sent

---

คำอธิบายเชิงลึก (ทำไมทำแบบนี้)

- **แยก Layer** ทำให้โค้ดอ่านง่าย ทดสอบง่าย (mock repository in unit tests) และเปลี่ยน persistence โดยไม่กระทบ business logic
- **Repository per Aggregate** ให้เมธอดที่สื่อความหมายของธุรกิจ (readability)

- **UnitOfWork** ใช้เพื่อกำหนด transaction boundary — ควร commit ใน Application Layer (UseCase)
- **Use Case / Service Layer**: รับผิดชอบ orchestration, validation, mapping, transaction เป็นจุดเดียวของการเปลี่ยนแปลง state
- **Query separation** (CQRS): แยก read model เพื่อ performance / optimization (AsNoTracking, selects only needed columns)
- **Outbox pattern**: ให้ความน่าเชื่อถือเมื่อต้องส่ง event ไปยังระบบภายนอก โดยใช้ single DB transaction

---

วิธีรัน & ทดลอง (แนะนำ)

- สร้างโฟลเดอร์ project แต่ละตัวตามโครงสร้างด้านบน
- สร้าง .csproj ปกติสำหรับแอป (single Web project per example)
- ติดตั้งแพ็กเกจ Microsoft.EntityFrameworkCore.InMemory และ Microsoft.AspNetCore.* ที่จำเป็น
- รัน dotnet run แล้วทดสอบด้วย curl หรือ Postman ตามตัวอย่างผลการรันด้านบน

โปรเจกต์แรก: ProductApp (Clean Architecture, 4 layers)

โปรเจกต์แรก: ProductApp (Clean Architecture, 4 layers) — จะให้ไฟล์ครบทุกไฟล์ที่ต้องมีเพื่อรันได้จริง (ใช้ **EF Core InMemory** เพื่อความสะดวกในการทดลอง) พร้อม **คำอธิบายโค้ด** และ **ตัวอย่างผลการรัน (curl / JSON)**

ขนาดโปรเจกต์เป็นแบบ “single .NET web project” แต่แยกโฟลเดอร์ทางตรรกะตาม Layer (Domain / Application / Infrastructure / Presentation) — วิธีนี้จะง่ายต่อการคัดลอกไปรันทันที และยังสะท้อน Clean Architecture ได้ชัดเจน

สรุปเร็ว ๆ (what you'll get)

- ชื่อโปรเจกต์: ProductApp
- .NET SDK: **7+ / 8+** (โค้ดใช้ WebApplication minimal hosting)
- Persistence: **EF Core InMemory**
- ฟีเจอร์: Create / GetById / List Products (Use Case handler, Repository, Domain entity, Controller)
- โครงสร้างโฟลเดอร์ & ไฟล์ครบ — คัดลอกแล้ว dotnet run ได้ทันที

โครงสร้างโปรเจกต์ (ไฟล์ทั้งหมด)