Step-by-Step Tutorial: Architectural Styles & Layering in .NET Projects

Step-by-Step Tutorial: Architectural Styles & Layering in .NET Projects

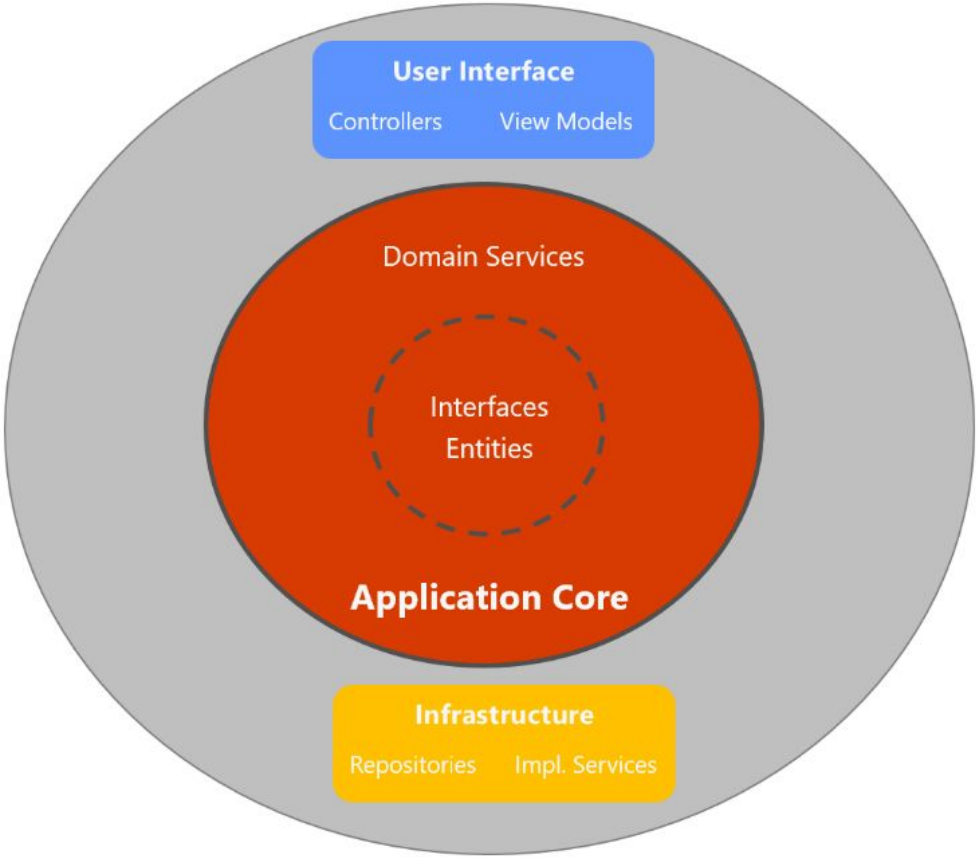
Overview

This tutorial walks through three popular architectural styles:

1. **Onion Architecture**
2. **Hexagonal (Ports & Adapters) Architecture**
3. **Vertical-Slice Architecture**

Finally, we cover **Trade-offs** and **Composition Root Best Practices** to tie everything together.

Clean Architecture Layers (Onion view)



External Dependencies

1. Onion Architecture

Why Onion Architecture?

- Provides flexible, sustainable, and portable architecture inspired by Domain-Driven Design.
- Promotes separation of concerns and low coupling, as layers depend inward only.
- Enhances maintainability: all code depends on deeper layers, isolating business rules.
- Improves testability: you can unit-test each layer in isolation.
- Allows swapping frameworks/technologies without impacting the core domain (e.g., RabbitMQ ↔ ActiveMQ, SQL ↔ MongoDB). (medium.com)

Principles

1. **Dependency Inversion**: Outer layers depend on inner layers; inner layers are unaware of outer ones. Define interfaces in core layers and implement them outward. (medium.com)
2. **Data Encapsulation**: Each layer hides implementation details and exposes interfaces. Use Data Transfer Objects (DTOs) at boundaries to avoid leaking formats. (medium.com)
3. **Separation of Concerns & Low Coupling**: Each layer handles distinct responsibilities, interacting through well-defined ports. Modules know only what they need. (medium.com)

Layers

1. **Domain Model/Entities**: Fundamental business concepts, independent of frameworks.
2. **Domain Services**: Encapsulate complex business logic and rules that don’t fit entity behaviors (e.g., tax and pricing algorithms). (medium.com)
3. **Application Services (Use Cases)**: Orchestrate domain services and entities to fulfill business scenarios without embedding rules.
4. **Infrastructure (Adapters)**: Communicate with external systems (databases, messaging, web) without containing business logic.
5. **Observability Services**: Monitor application health via metrics, logs, traces (e.g., ELK, Grafana, Datadog). (medium.com)

Testing Strategy:

- Unit tests for Domain Model, Domain Services, and Application Services.
- Integration tests for Infrastructure adapters.
- End-to-end/BDD tests for full workflows. (medium.com)

Do you need every layer?

Depends on your application's complexity. For simpler CRUD-centric apps, you might skip Domain Services but always enforce outer→inner dependencies. (medium.com)

Steps:

Step 1: Create the solution and projects

```
mkdir OnionDemo && cd OnionDemo

dotnet new sln -n OnionDemo

# Domain

mkdir src/OnionDemo.Domain

cd src/OnionDemo.Domain

dotnet new classlib

# Application

cd ../..

mkdir src/OnionDemo.Application

cd src/OnionDemo.Application

dotnet new classlib
```

```
# Infrastructure
cd ../..
mkdir src/OnionDemo.Infrastructure
cd src/OnionDemo.Infrastructure
dotnet new classlib
# API
cd ../../..
mkdir src/OnionDemo.Api
cd src/OnionDemo.Api
dotnet new webapi
```

Step 2: Define your domain

```
// src/OnionDemo.Domain/Entities/Product.cs
public class Product {
        public Guid Id { get; private set; }
        public string Name { get; private set; }
        public decimal Price { get; private set; }
        // Behavior
        public void ChangePrice(decimal newPrice) {
        if (newPrice <= 0) throw new ArgumentException("Price must be positive");
        Price = newPrice;
        }
}
```

Step 3: Define application services and interfaces

```
// src/OnionDemo.Application/Interfaces/IProductRepository.cs
public interface IProductRepository {
    Task<Product> GetByIdAsync(Guid id);
    Task SaveAsync(Product product);
}
```

```
// src/OnionDemo.Application/Services/ProductService.cs
public class ProductService {
        private readonly IProductRepository _repo;
        public ProductService(IProductRepository repo) => _repo = repo;

        public async Task UpdatePrice(Guid id, decimal newPrice) {
        var product = await _repo.GetByIdAsync(id);
        product.ChangePrice(newPrice);
        await _repo.SaveAsync(product);
        }
}
```

Step 4: Implement infrastructure

```
// src/OnionDemo.Infrastructure/Data/ProductRepository.cs
public class ProductRepository : IProductRepository {
        private readonly DemoDbContext _ctx;
        public ProductRepository(DemoDbContext ctx) => _ctx = ctx;

        public Task<Product> GetByIdAsync(Guid id) => _ctx.Products.FindAsync(id).AsTask();
        public Task SaveAsync(Product product) {
        _ctx.Products.Update(product);
        return _ctx.SaveChangesAsync();
        }
}
```

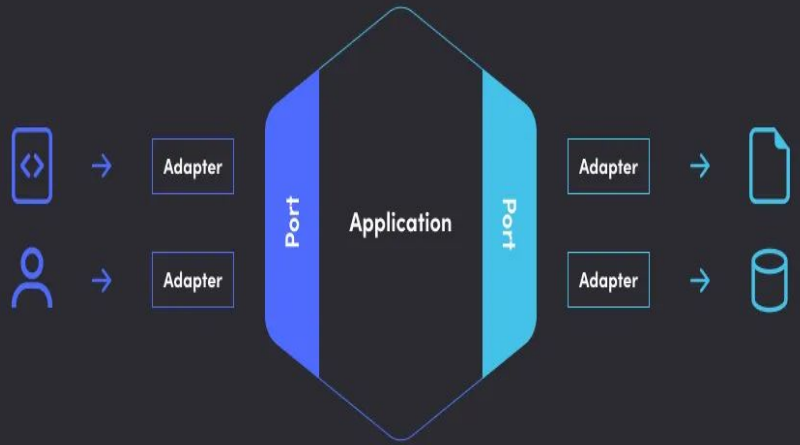
Step 5: Configure Dependency Injection (Composition Root)

```
// src/OnionDemo.Api/Program.cs

builder.Services.AddDbContext<DemoDbContext>(opt => ...);

builder.Services.AddScoped<IProductRepository, ProductRepository>();

builder.Services.AddScoped<ProductService>();
```

Step 6: Expose via Web API

```
// src/OnionDemo.Api/Controllers/ProductController.cs
[ApiController]
[Route("api/products")]
public class ProductController : ControllerBase {
        private readonly ProductService _service;
        public ProductController(ProductService service) => _service = service;

        [HttpPut("{id}/price")]
        public async Task<IActionResult> ChangePrice(Guid id, [FromBody] decimal newPrice) {
        await _service.UpdatePrice(id, newPrice);
        return NoContent();
        }
}
```

Hexagonal Architecture in .NET the fastest (right) way



2. Hexagonal (Ports & Adapters) (Ports & Adapters) Architecture

Hexagonal Architecture emphasizes isolating the domain inside a hexagon, with **Ports** (interfaces) on one side and **Adapters** (implementations) on the outside.

Steps:

Step 1: Define Domain and Ports

src/HexDemo.Domain

└── Entities, Domain Events

src/HexDemo.Application

└── Ports (interfaces) for inbound & outbound operations

```
// src/HexDemo.Application/Ports/IProductReader.cs  (Inbound Port)
public interface IProductReader {
        Task<ProductDto> GetById(Guid id);
}

// src/HexDemo.Application/Ports/IProductWriter.cs  (Outbound Port)
public interface IProductWriter {
        Task Save(ProductDto dto);
}
```

Step 2: Write Application Interactors

```
// src/HexDemo.Application/Interactors/ProductInteractor.cs
public class ProductInteractor : IProductReader {
        private readonly IProductWriter _writer;
        private readonly IProductRepository _repo; // domain repository

        public ProductInteractor(IProductWriter w, IProductRepository r) {
        _writer = w;
        _repo = r;
        }
        public async Task<ProductDto> GetById(Guid id) =>
        await _repo.Find(id).ToDto();
}
```

Step 3: Build Adapters

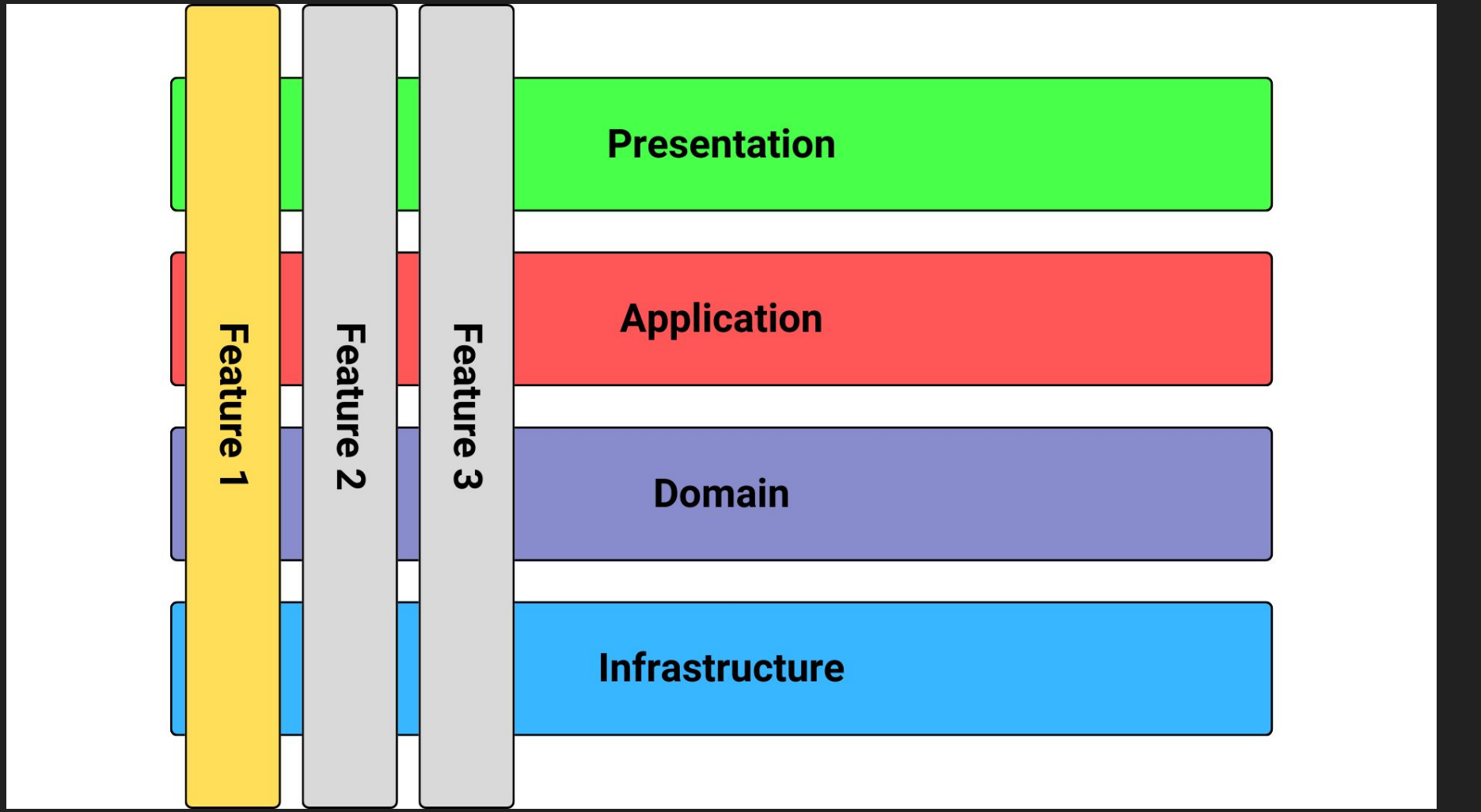
- **Inbound Adapters**: Web API controllers implementing inbound ports.
- **Outbound Adapters**: EF Core or third-party integrations implementing outbound ports.

```
// src/HexDemo.Api/Adapters/ProductController.cs

public class ProductController : ControllerBase, IProductReader {

    private readonly IProductReader _reader;

    public ProductController(IProductReader reader) => _reader = reader;


    [HttpGet("/products/{id}")]

    public Task<ProductDto> GetById(Guid id) => _reader.GetById(id);

}
```

```
// src/HexDemo.Infrastructure/Adapters/ProductEfAdapter.cs
public class ProductEfAdapter : IProductWriter {
        private readonly DemoDbContext _ctx;
        public ProductEfAdapter(DemoDbContext ctx) => _ctx = ctx;
        public async Task Save(ProductDto dto) {
        var entity = dto.ToEntity();
        _ctx.Products.Update(entity);
        await _ctx.SaveChangesAsync();
        }
}
```

Step 4: Composition Root

```
builder.Services.AddScoped<IProductReader, ProductInteractor>();

builder.Services.AddScoped<IProductWriter, ProductEfAdapter>();

// Domain repository also registered
```


Feature 1
Feature 2
Feature 3
Presentation
Application
Domain
Infrastructure

3. Vertical-Slice Architecture

Vertical-Slice groups code by feature rather than by layer, improving focus and reducing navigation.

Steps:

Step 1: Organize Folder Structure

```
src/MyApp.Api
└── Features
    ├── Products
    │   ├── Create
    │   │   ├── CreateCommand.cs
    │   │   ├── CreateHandler.cs
    │   │   └── CreateValidator.cs
    │   └── UpdatePrice
    │       ├── UpdatePriceCommand.cs
    │       ├── UpdatePriceHandler.cs
    │       └── UpdatePriceValidator.cs
    └── Shared (DTOs, common utilities)
```

Step 2: Define a Command & Handler

```
// UpdatePriceCommand.cs

public record UpdatePriceCommand(Guid Id, decimal NewPrice) : IRequest;

// UpdatePriceHandler.cs

public class UpdatePriceHandler : IRequestHandler<UpdatePriceCommand> {

        private readonly DemoDbContext _ctx;

        public UpdatePriceHandler(DemoDbContext ctx) => _ctx = ctx;

        public async Task<Unit> Handle(UpdatePriceCommand req, CancellationToken ct) {

        var product = await _ctx.Products.FindAsync(req.Id);

        product.ChangePrice(req.NewPrice);

        await _ctx.SaveChangesAsync(ct);

        return Unit.Value;

        }

}
```

Step 3: Wire up MediatR & Validation

```
builder.Services.AddMediatR(cfg =>
cfg.RegisterServicesFromAssembly(typeof(Program).Assembly));

builder.Services.AddValidatorsFromAssembly(typeof(Program).Assembly);
```

Step 4: Expose Endpoints

```
app.MapPut("/products/{id}/price", async (Guid id, decimal newPrice, IMediator
mediator) => {

    await mediator.Send(new UpdatePriceCommand(id, newPrice));

    return Results.NoContent();

});
```

4. Trade-offs & Composition Root Best Practices

Concern	Onion	Hexagonal	Vertical-Slice
Modularity	High, clear layers	High, port abstractions	High, feature focus
Complexity	Medium to high	High	Low to medium
Testability	Excellent	Excellent	Excellent
Ease of onboarding	Steeper learning curve	Steep understanding ports	Shallower for features

Composition Root Best Practices

1. **Single Location**: Wire up all dependencies in `Program.cs` (or `Startup`) only.
2. **Avoid Service Locator**: Do not resolve services manually inside classes; rely on constructor injection.
3. **Group Registrations**: Use extension methods like `AddInfrastructure(this IServiceCollection)` to keep `Program.cs` clean.
4. **Avoid Anti-patterns**:
    - No registration inside domain or application layers.
    - No `new` for services; let DI container manage lifetimes.