Clean architectures

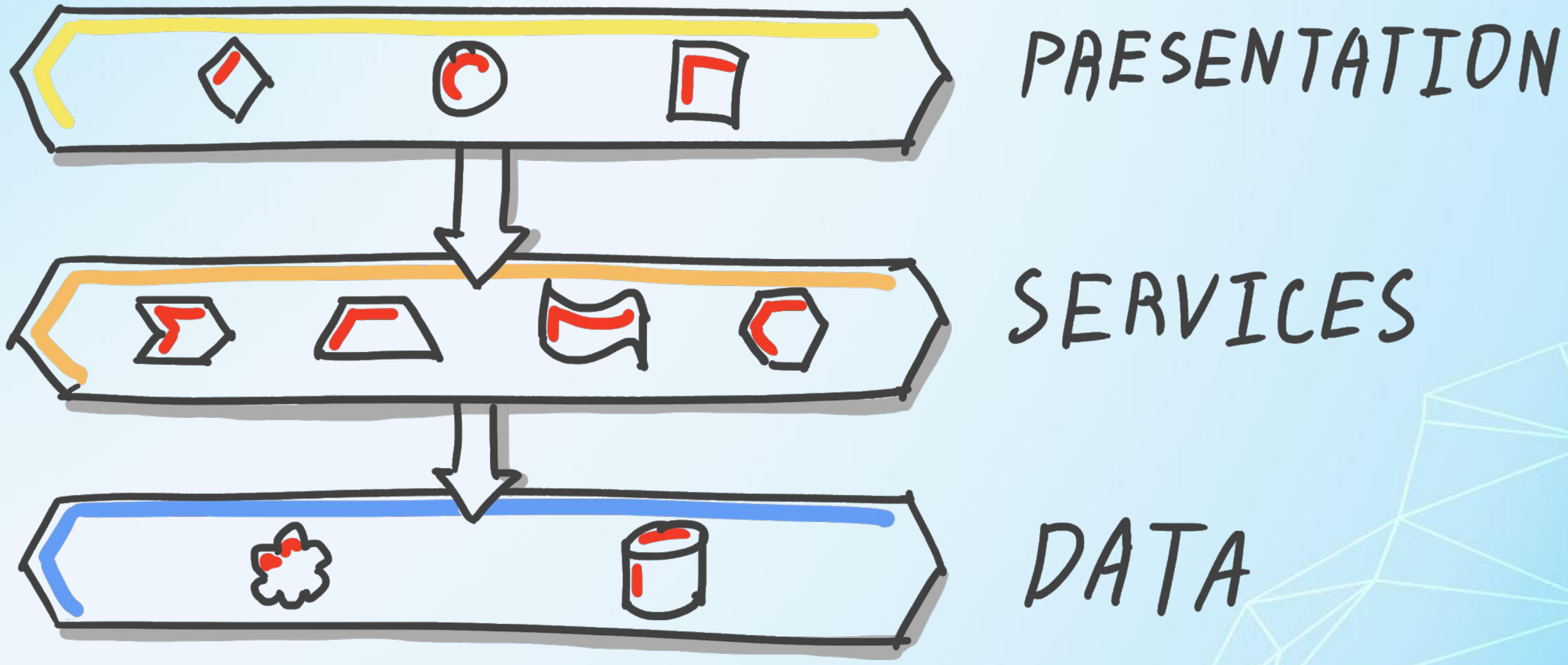
Patudin Ivan

https://github.com/grem0087/CleanArchitecture

Agenda

- Brief history
- What is clean architecture
- Why do you need clean architecture
- A story about architecture and subtleties of implementation

n-layer architecture overview



n-layer architecture disadvantages

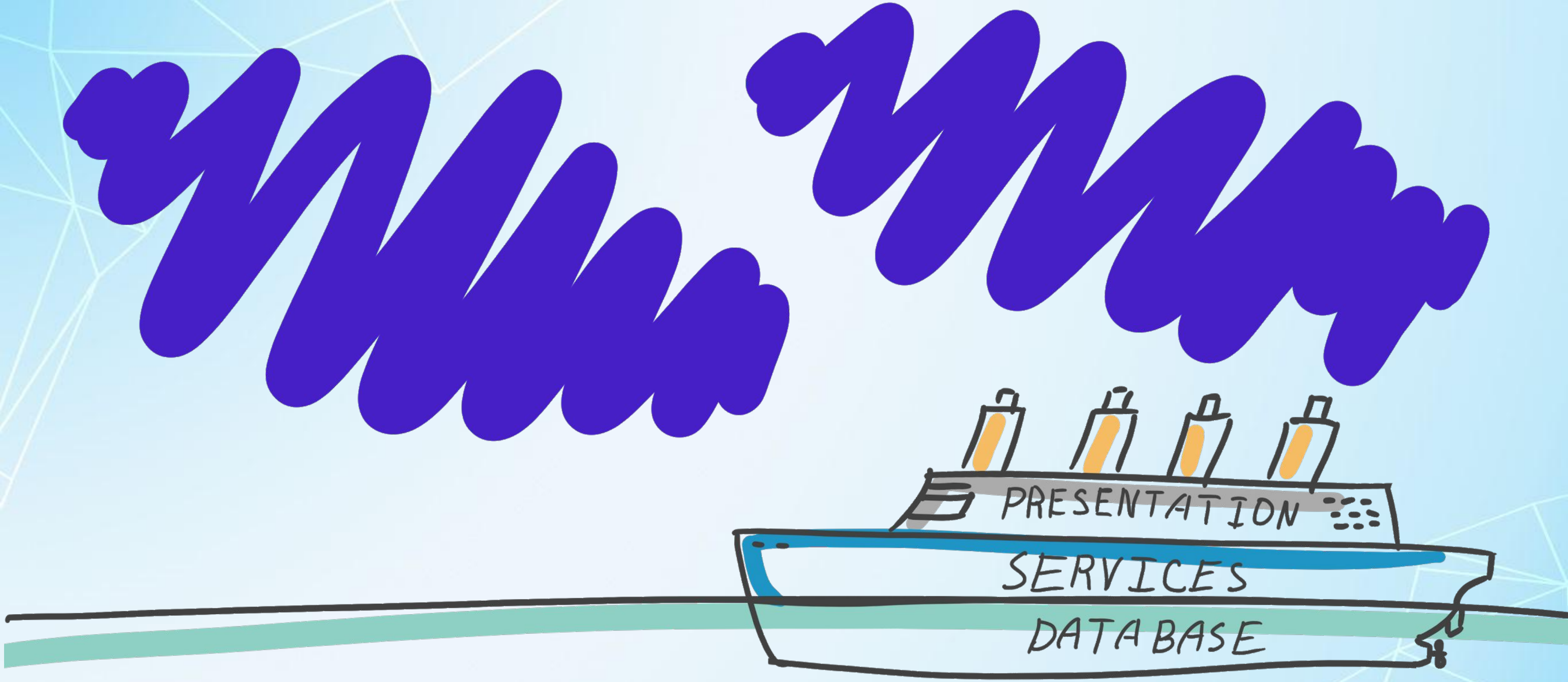
- Domain layer is closely related to the database
- Application logic layer over time grows larger and more complex
- Difficulty of making new changes

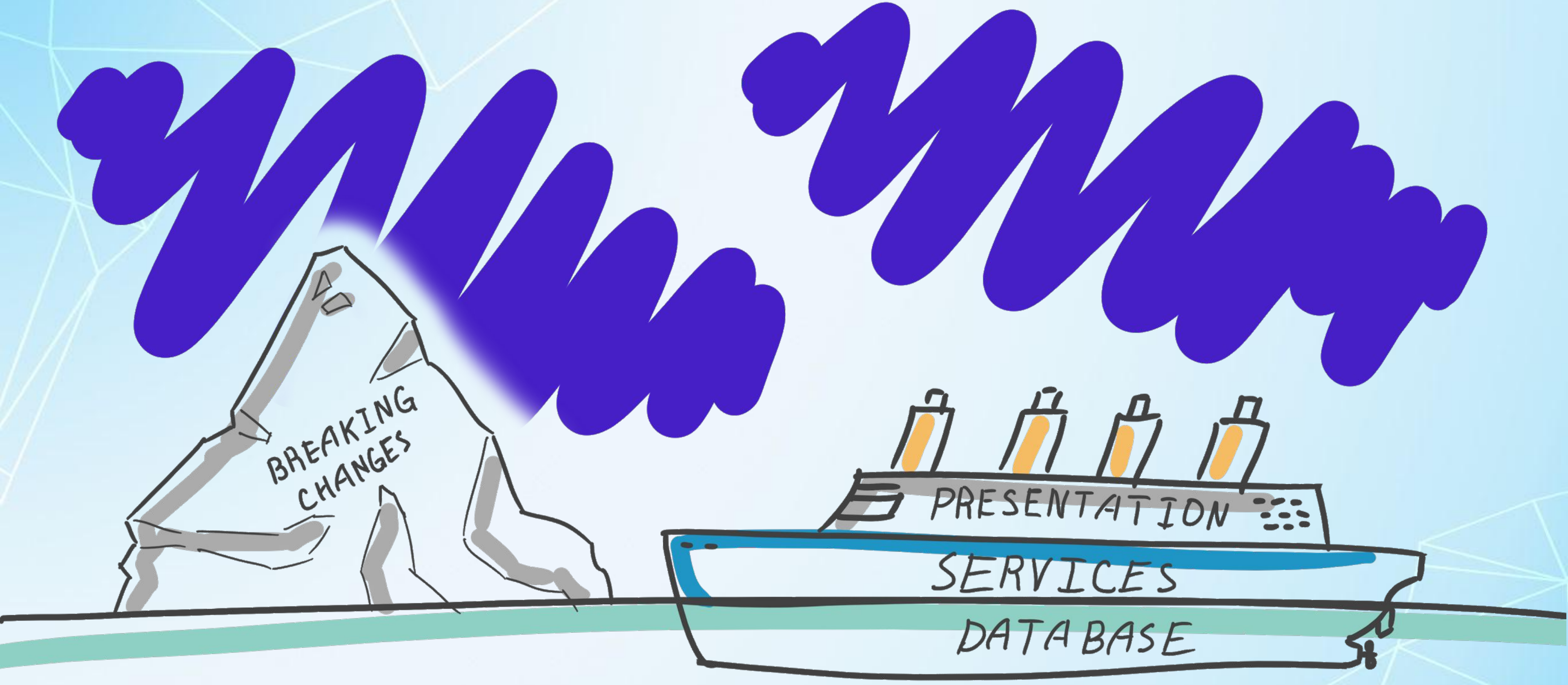
n-layer architecture

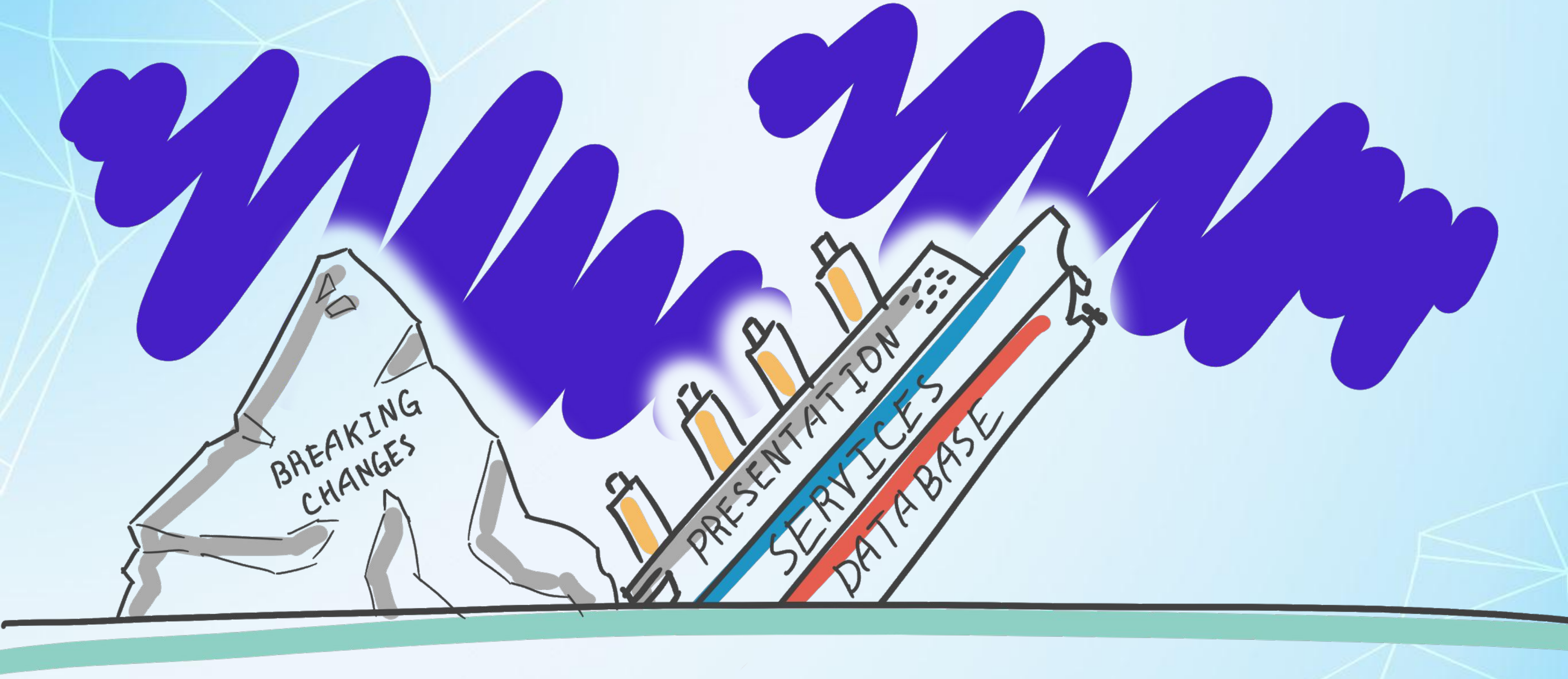
- Most common architecture
- Suitable for small applications
- Usually suitable for already existing projects

When is it not suitable?

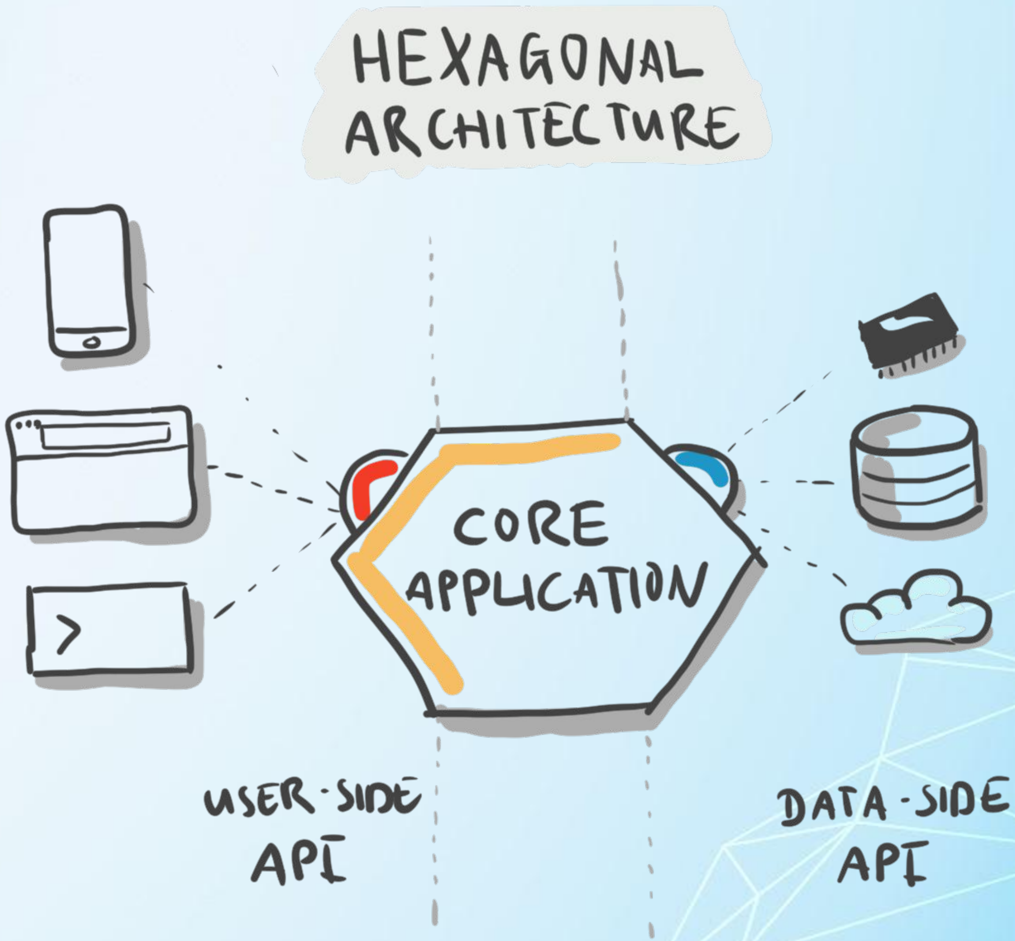
- When a large project is just starting
- When you often have to experiment
- When there is no understanding of further direction of a project


PRESENTATION
SERVICES
DATABASE


BREAKING CHANGES
PRESENTATION
SERVICES
DATABASE

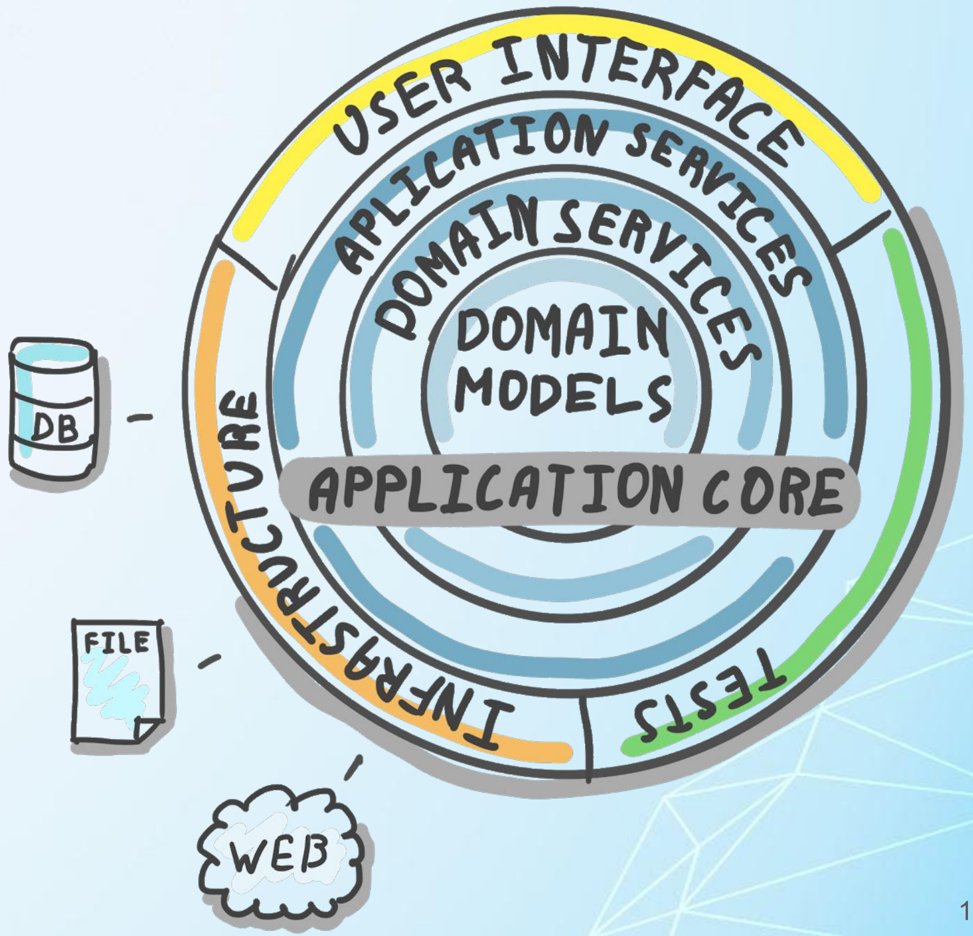

BREAKING CHANGES
PRESENTATION
SERVICES
DATABASE

- Alistair Cockburn invented it in 2005
- Levels are highlighted:
  - Core
  - Applications
  - Infrastructures
- Ports - interfaces
- Adapters - implementation



Onion

- Jeffrey Palermo 2008
- This architecture is not appropriate for small websites. It is appropriate for long-lived business applications as well as applications with complex behavior.



What is Clean Architecture

Term was invented by Robert Martin. Clean Architecture is a compilation of principles and requirements. Most importantly from:

- Screaming Architecture by himself
- Hexagonal Architecture (a.k.a. Ports and Adapters) by Alistair Cockburn
- Onion Architecture by Jeffrey Palermo

Solving problems

- Application layer cohesion
- Complexity of development and introduction of new changes
- System support difficulty
- Testing difficulty

Principles

SOLID

Single responsibility; Dependency Inversion;

Components

- Our application consists of components
- Some components are core business rules, other are plugins that contain technical implementation

Component principles

Reuse / Release Equivalence Principle (REP)

”The granule of reuse is the granule of release. Only components that are released through a tracking system can effectively be reused.”

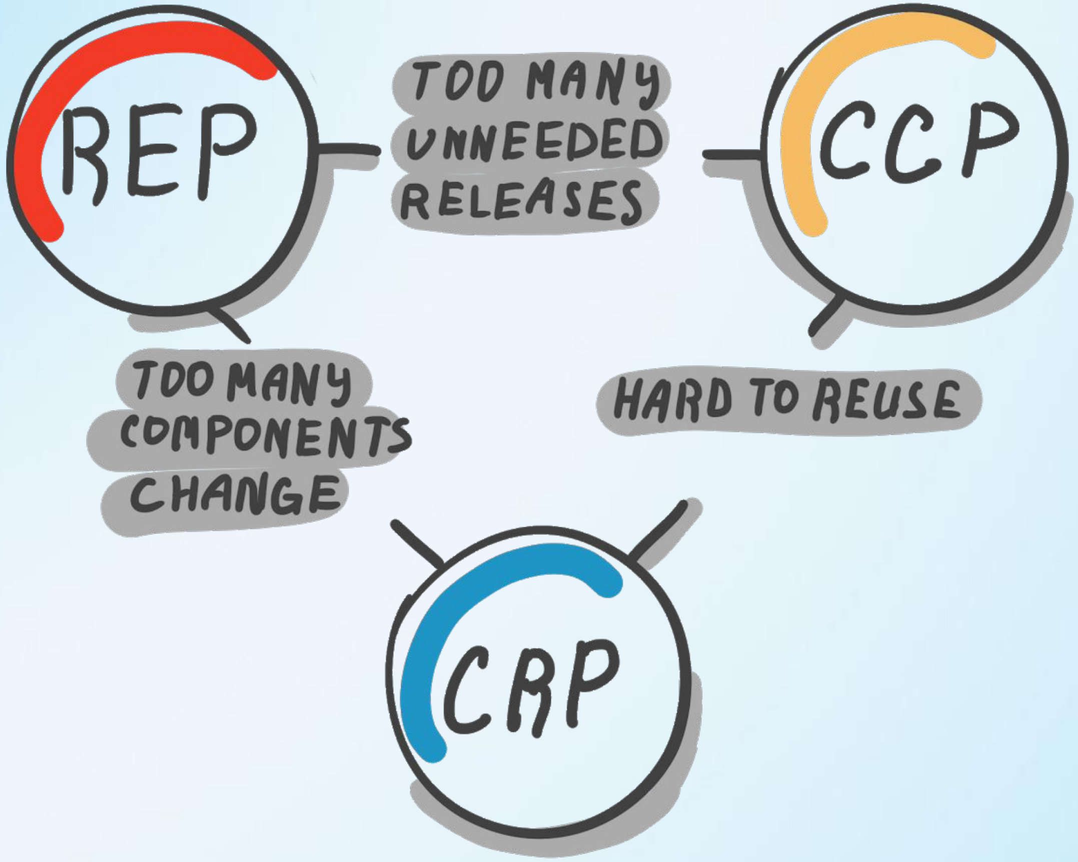
Common Closure Principle (CCP)

”The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.”

Common Reuse Principle (CRP)

”The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all.”

Tension Diagram for Component Cohesion



Advantages

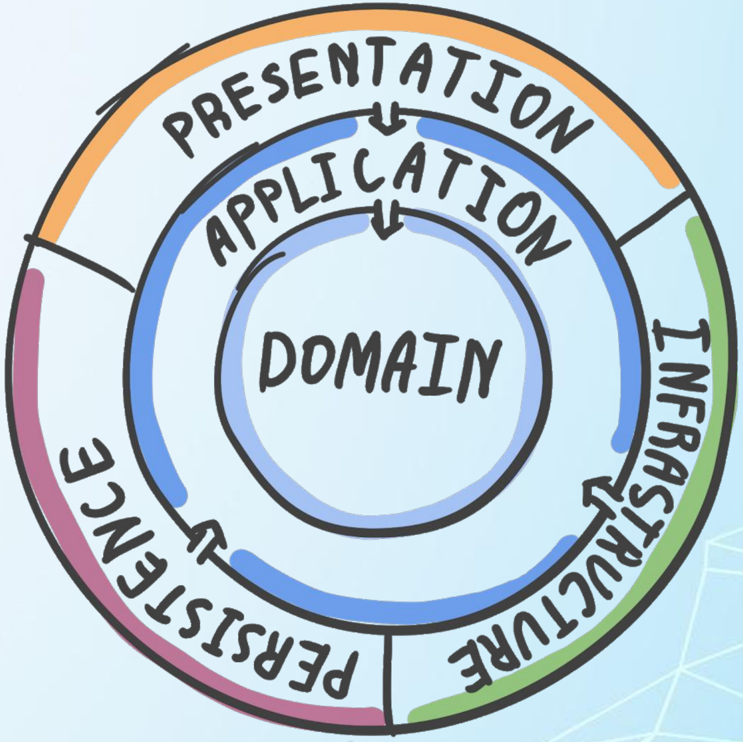
- Extensibility
- Maintainability
- Testability

Requirements

- Independent of Framework
- Testable
- Independent of UI
- Independent of Database
- Independent of any external agents, clients

Clean architecture

- Core
  - Domain
  - Application
    - Application interfaces
- Infrastructure
  - External clients
  - Implementations
  - Data
- Presentation (Web)



Domain

- Entities
- Exceptions
- Enumerables
- Domain Events
- Domain Models
- Interfaces
- Domain object internal logic (validation)

Should not contain any links to ORMs, frameworks and should not have database knowledge/dependencies

```
public class Order
{
    public Order()
    {
        Products = new HashSet<OrderProduct>();
    }

    [Key]
    public int Id { get; set; }

    [MaxLength(256)]
    [Column(TypeName = "nvarchar(24)")]
    public string Descriptions { get; set; }
    public DateTime? OrderDate { get; set; }

    public Address Address { get; set; }
    public Customer Customer { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }

    public ICollection<OrderProduct> Products { get; private set; }

    public decimal TotalPrice { get; set; }
}
```

```
public class Order
{
    public Order()
    {
        Products = new HashSet<OrderProduct>();
    }


    public int Id { get; set; }



    public string Descriptions { get; set; }
    public DateTime? OrderDate { get; set; }

    public Address Address { get; set; }
    public Customer Customer { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }

    public ICollection<OrderProduct> Products { get; private set; }

    public decimal TotalPrice { get; set; }
}
```

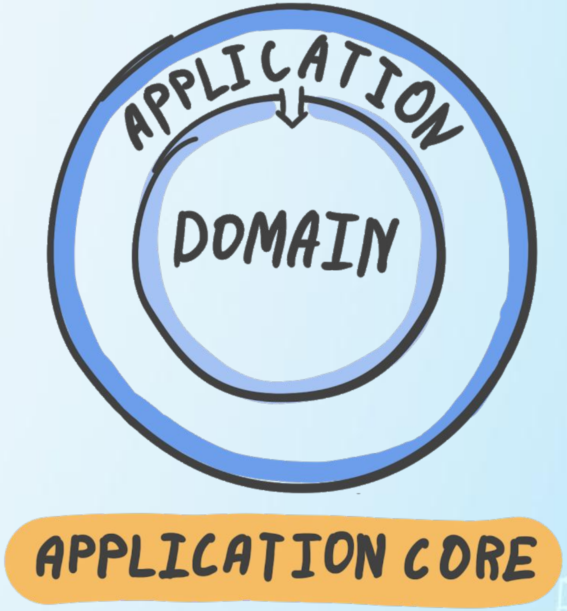
```
public class Burger
{
    public Burger(int id, string name, BurgerType type, decimal price, string description) {}

    public int Id { get; private set; }

    public string Name { get; private set; }
    public decimal Price { get; private set; }

    public string Description { get; set; }

    public void ChangeName(string name)
    {
        if (string.IsNullOrEmpty(name)) throw new InvalidNameException("Burger name is empty.");
        Name = name;
    }

    public void ChangePrice(decimal price)
    {
        if (price <= 0) throw new InvalidPriceException("Burger price can not be zero or less.");
        Price = price;
    }
}
```

Domain

- Avoid using attributes that lead to unnecessary dependencies, use FluentApi
- Use private setters and object initialization
- Use your own domain-level exceptions

Application

- DTOs and Models
- Application Logic
- Interfaces of:
  - mappers
  - external services
- Commands/Queries or Services
- Validators

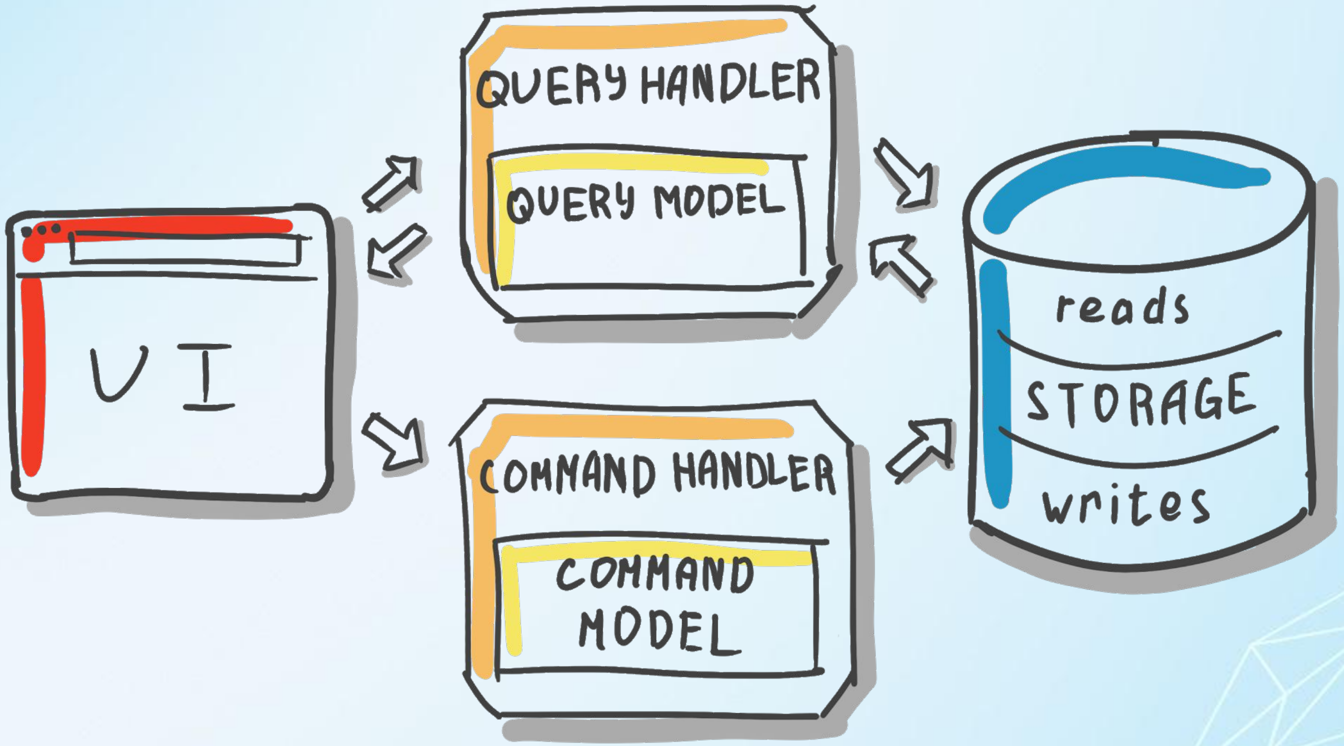


CQRS

vs

SERVICES

CQRS Pattern



I couldn't become fatter.

I just came out of service

```csharp
public class CreateOrderCommand : IRequest<int>
{
    [Required]
    [MaxLength(28)]
    public string Name { get; set; }
    public string Street { get; set; }

    [MaxLength(28)]
    public string City { get; set; }
    public string House { get; set; }

    [RegularExpression(@"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}", ErrorMessage = "Wrong phone number")]
    public string Phone { get; set; }


    public ICollection<OrderBurgerModel> Burgers { get; set; }
}
```

```
public class CreateOrderCommand : IRequest<int>
{


    public string Name { get; set; }
    public string Street { get; set; }


    public string City { get; set; }
    public string House { get; set; }


    public string Phone { get; set; }


    public ICollection<OrderBurgerModel> Burgers { get; set; }
}
```

```
public class CreateOrderCommandValidator : AbstractValidator<CreateOrderCommand>
{
    public CreateOrderCommandValidator()
    {
        RuleFor(x => x.Phone).NotEmpty();
        RuleFor(x => x.Burgers.All(b => b.Quantity > 0));

        RuleFor(x => x.City).MaximumLength(28);
        RuleFor(x => x.Name).MaximumLength(28);

        RuleFor(x => x.Phone)
            .Matches(@"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}")
            .WithMessage("Invalid phone number");

        RuleFor(x => x.Phone)
            .NotEmpty()
            .When(x=>string.IsNullOrEmpty(x.Street) || string.IsNullOrEmpty(x.House))
            .WithMessage("You should state phone or address");
    }
}
```

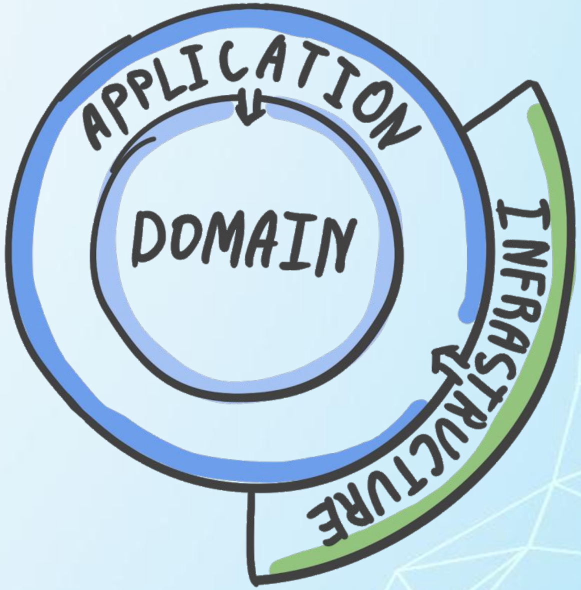
```csharp
public class CreateOrderCommandHandler : IRequestHandler<CreateOrderCommand, int>
 {
     private INotificationService _notificationService;
     private readonly IOrderRepository _orderRepository;
     private readonly IMapper _mapper;

     public CreateOrderCommandHandler( IOrderRepository orderRepository,
           IMapper mapper, INotificationService notificationService) {}

     public async Task<int> Handle(CreateOrderCommand request, CancellationToken cancellationToken)
     {
         var orderEntity = _mapper.Map<Order>(request);

         _orderRepository.Add(orderEntity);

         _orderRepository.SaveAll();
         await _notificationService.SendAsync(new Message
             {To = "MyLittleFriend", Body = $"OrderCreated with Id {orderEntity.Id}"});

         return orderEntity.Id;
     }
 }
```

Application

- Contains workflow of application
- Contains logic of workflow
- You can use FluentValidation instead of validation attributes
- Does not depend on infrastructure and data layers

Infrastructure

- External services implementations
- API Clients
- Mapper (Binding) rules
- etc.



```
public class NotificationService : INotificationService
{
    private readonly Producer<Null, string> _producer;
    private Consumer<Null, string> _consumer;

    private readonly IDictionary<string, object> _producerConfig;

    public NotificationService(string host)
    {
        _producerConfig = new Dictionary<string, object> {{"bootstrap.servers", host}};

        _producer = new Producer<Null, string>(_producerConfig, new StringSerializer(Encoding.UTF8));
    }

    public async Task SendAsync(Message message)
    {
        await _producer.ProduceAsync(message.To, null, message.Body);
    }
}
```

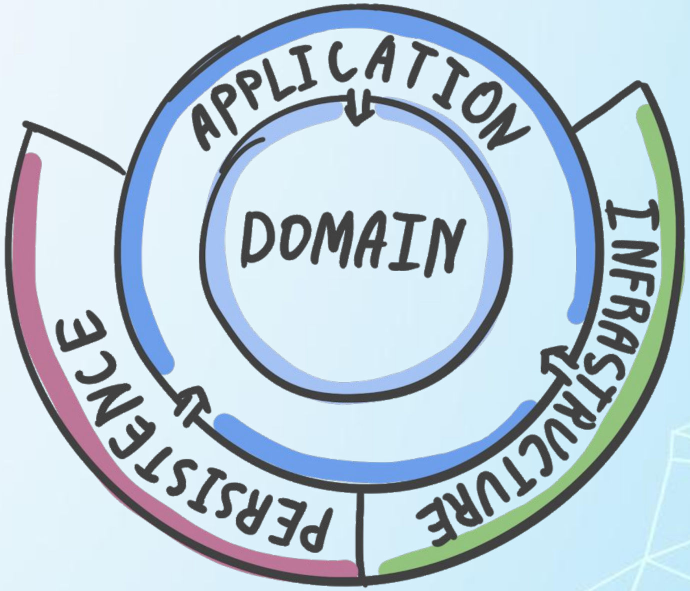
```
public class OrderProfile : Profile
{
    public OrderProfile()
    {
        CreateMap<CreateBurgerCommand, Burger>()
            .ForMember( dest => dest.Description, opt => opt.MapFrom(src => src.Description) )
            .ForMember(dest => dest.Price,
                opt => opt.MapFrom( src  => GetPrice(src.Discount, src.Price)));
    }

    private decimal GetPrice(DiscountType discountType, decimal firstPrice)
    {
        switch (discountType)
        {
            case DiscountType.Minimal: return firstPrice * 0.1m;

            case  DiscountType.Maximum: return firstPrice * 0.5m;

            case  DiscountType.Avarage: return firstPrice * 0.3m;

            default: throw new NotImplementedException($"DiscountType {discountType} unknown.");
        }
    }
}
```

Infrastructure

- Other layers do not depend on infrastructure
- For binding it is better to use a mapper
- Contains implementations of all external clients and interfaces advertised at lower levels

Persistent (DataBase)

- DbContext
- Migrations
- Configurations
- Repositories

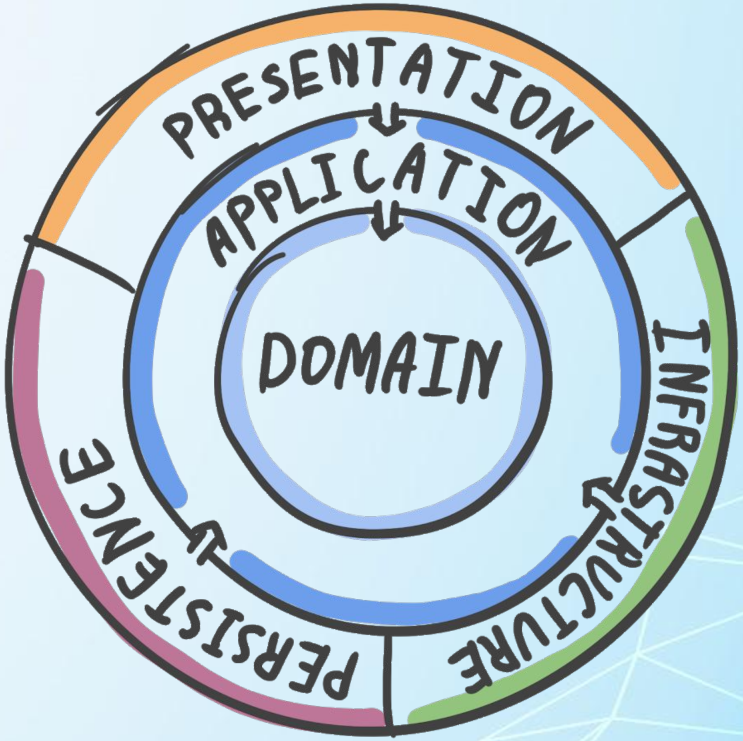


```csharp
public class BurgerMarketDbContext : DbContext, IBurgerMarketDbContext
{
    public BurgerMarketDbContext(DbContextOptions<BurgerMarketDbContext> options)
        : base(options)
    {
    }

    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<Burger> Burgers { get; set; }
    public DbSet<Drink> Drinks { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Get all configurations from assembly
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(BurgerMarketDbContext).Assembly);

    }
}
```

```
public class OrderConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.HasKey(b => b.Id);

        builder.Property(e => e.AddressId)
            .HasColumnName("AddressID")
            .IsRequired();

        builder.Property(e => e.CustomerId)
            .HasColumnName("CustomerId")
            .IsRequired();

    }
}
```

Presentation

- SPA - Angular/React
- Razor
- WebForms
- Mobile Apps
- Best practice is for controllers to not contain logic



Summary

- You have to consider where to use
- Very useful if app has large/frequently changing domain
- Very useful if application roadmap is unknown
- Not a silver bullet

https://github.com/dotnet-architecture/eShopOnWeb

https://github.com/dotnet-architecture/eShopOnContainers

https://jeffreypalermo.com/2013/08/onion-architecture-part-4-after-four-years/

https://bitbucket.org/jeffreypalermo/onion-architecture/src/default/Core/

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

https://jimmybogard.com/

https://github.com/jbogard/ContosoUniversityDotNetCore-Pages?WT.mc_id=-blog-scottha

https://docs.microsoft.com/ru-ru/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures