Модульный монолит вместо микросервисов: как, когда и зачем

Денис Цветцих

DOTNEXT

DevBrothers

Обо мне

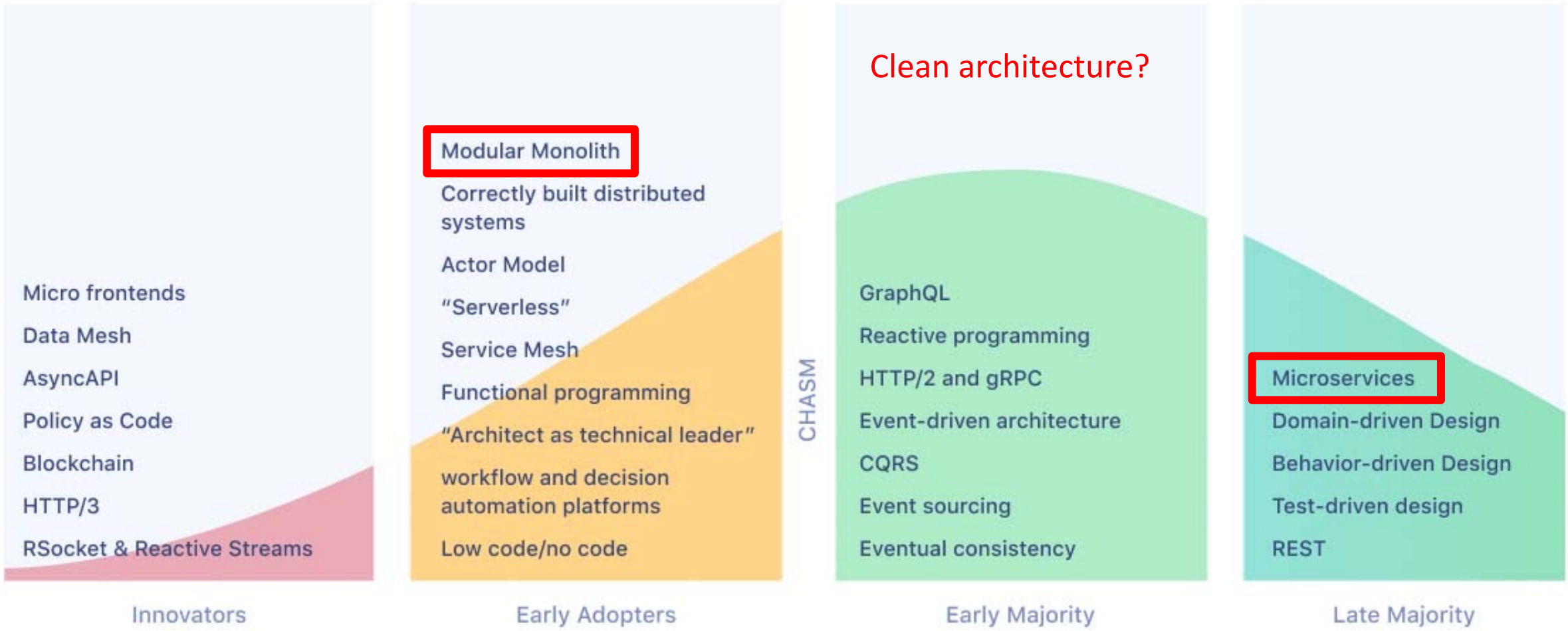
- Положительный опыт в монолитах
  - От стартапа до ERP
- Положительный опыт в микросервисах
  - Они нужны не всегда

Вопросы агитаторам за микросервисы

- От микросервисов была польза?
  - Конечно, да!
- Какая?
  - Несколько команд
  - Изоляция частей системы
- Есть микросервисы, которых больше одного инстанса?
  - Только один сказал да
- Так зачем микросервисы?
  - Ну… э… отстань, зануда!

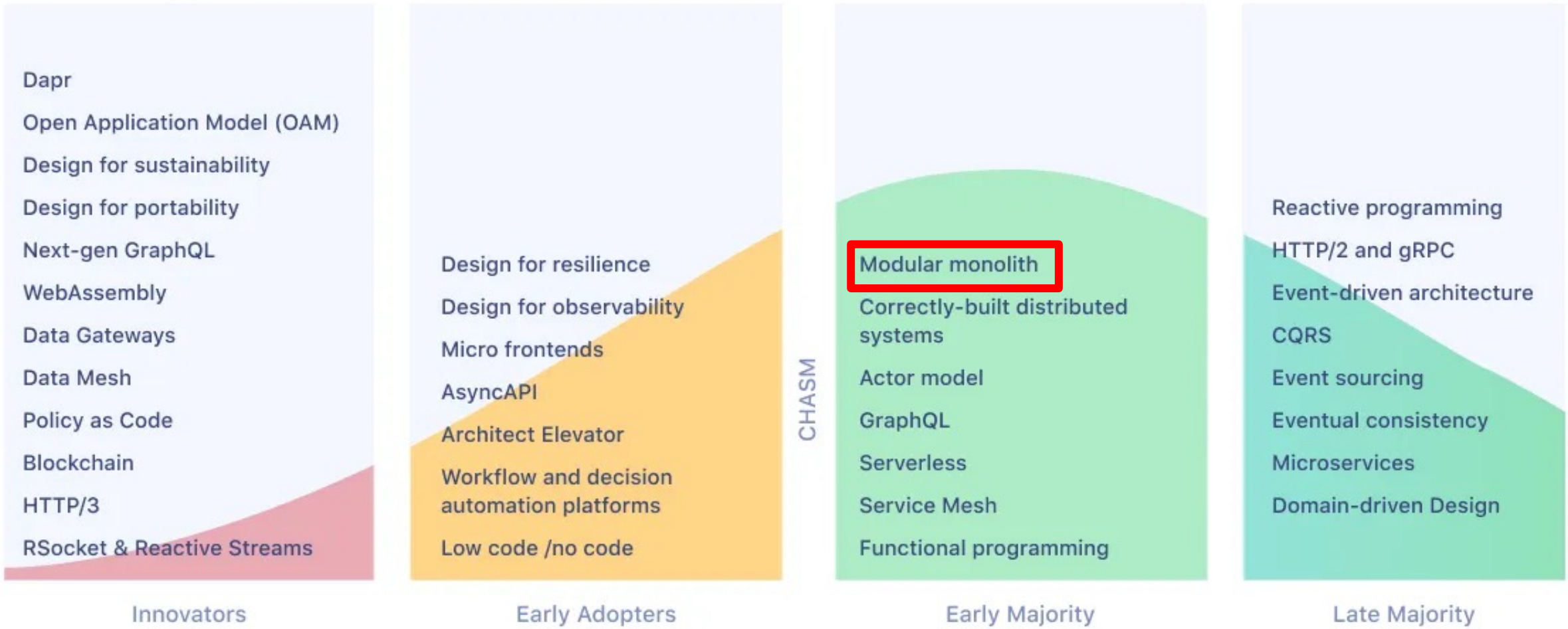
Software Development
Architecture and Design 2020 Q2 Graph
http://infoq.link/architecture-trends-2020

InfoQ



Software Development
Architecture and Design 2021 Graph
http://infoq.link/architecture-trends-2021

InfoQ



Монолит

Микросервисы

Поговорим о монолитах

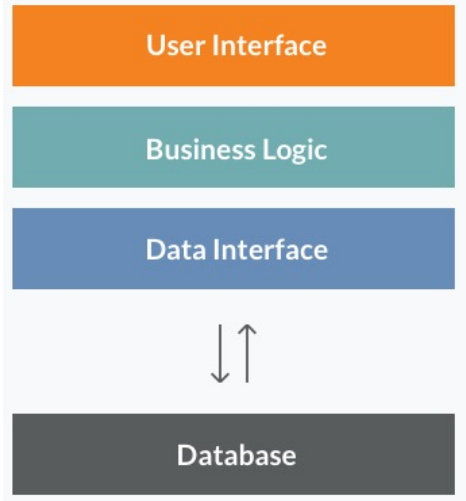
- Что такое монолит
- Когда его использовать
- Когда ему пора стать модульным
- Как перейти к модульному монолиту

Примеры кода на C#

Что такое монолит

- Один .exe файл (Роберт Мартин)

- Единый процесс, единое веб-приложение или единая служба (Microsoft)

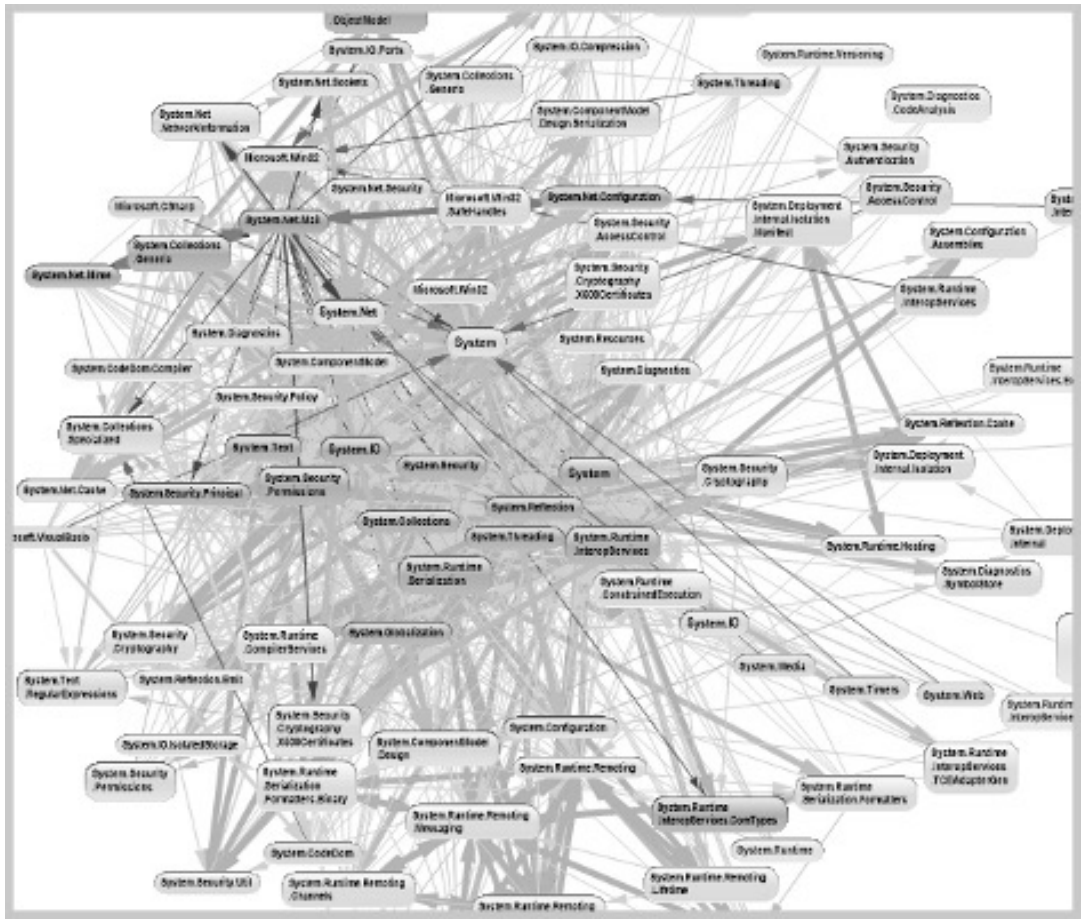


Достоинства монолитов

- Быстро и просто разрабатывать
  - Легче отлаживать (одна транзакция, один процесс)
  - Согласованность данных (одна база)
- Не нужна супер квалифицированная команда
  - Дешево
- Легко деплоить

За что их ругают

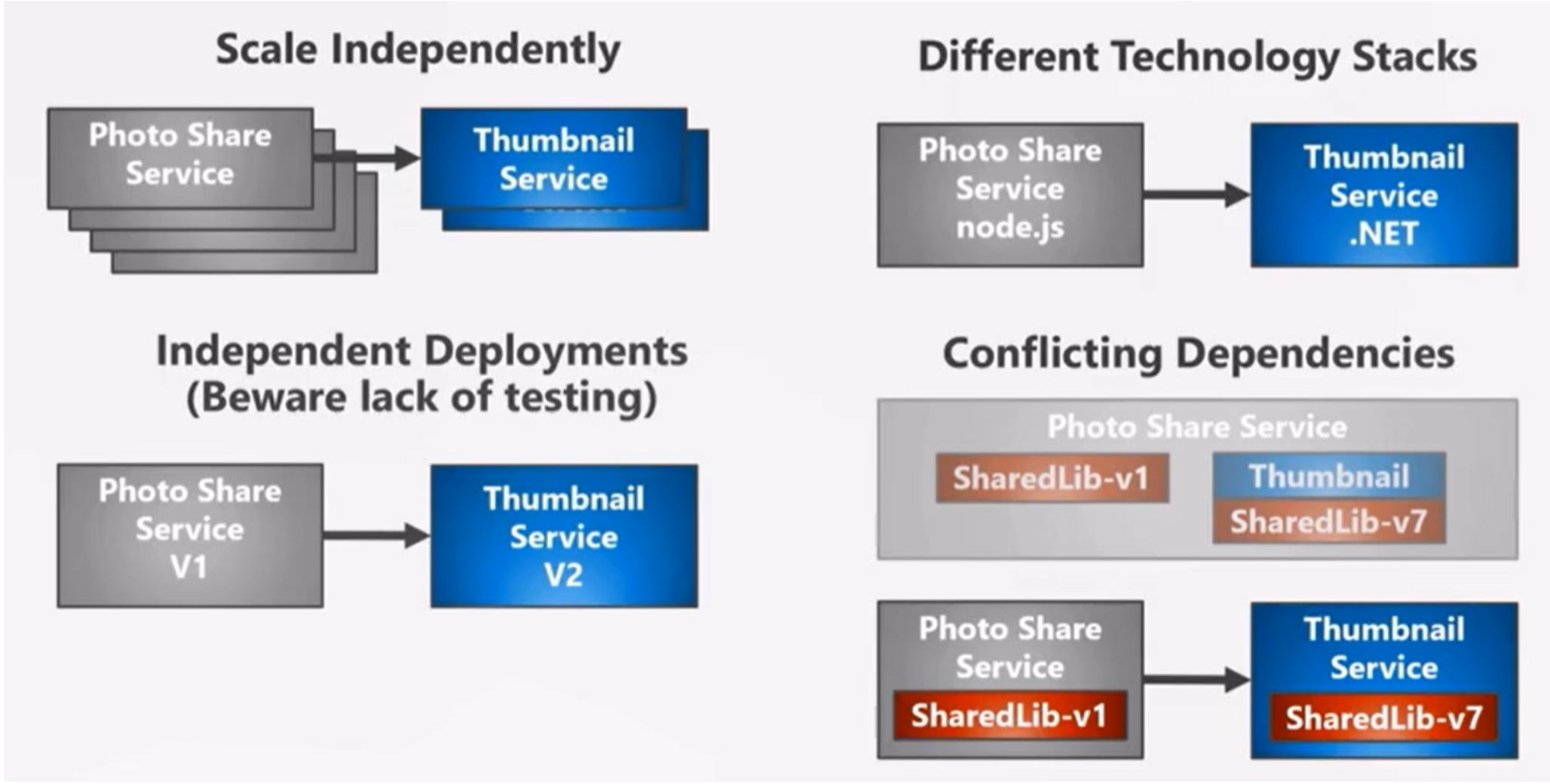
- Энтропия – со временем проект превращается в легаси
  - Большой ком грязи
  - Спагетти-код



И на сцену выходят микросервисы

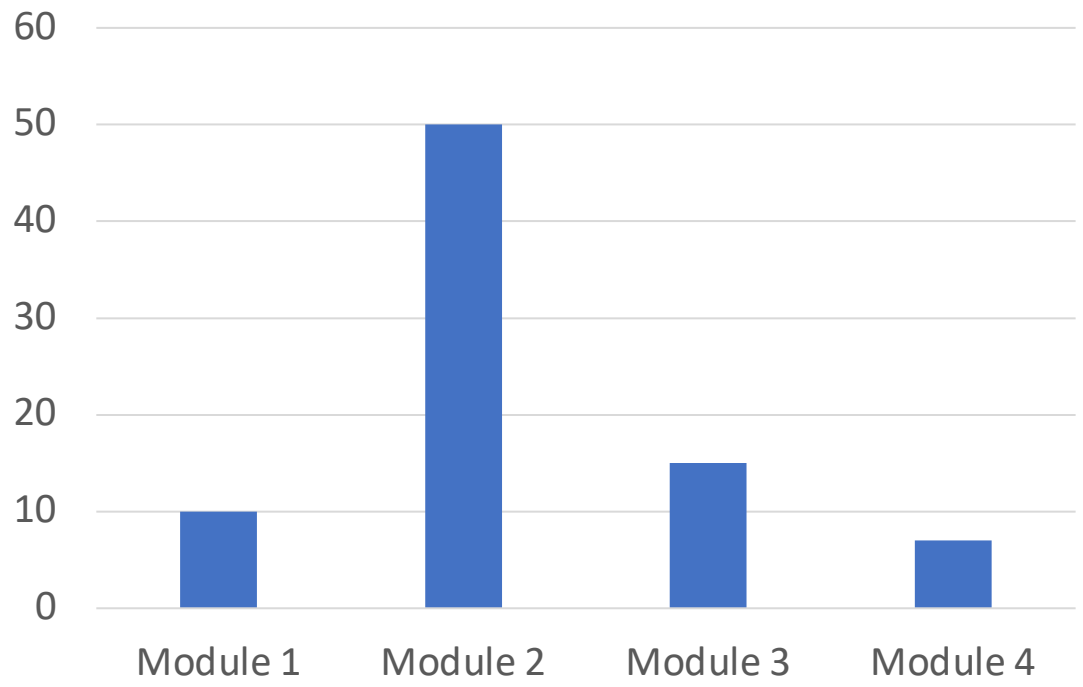
- Решение проблем монолита
  - Физическая изоляция микросервисов
  - Независимая разработка микросервисов
- Дополнительные бонусы
  - Независимое масштабирование
  - Независимый деплой

Преимущества микросервисов (Рихтер)



Независимое масштабирование – это главное

- Гибкое масштабирование
  - на уровне Application
  - на уровне БД
- Изоляция
  - Приятный бонус
  - Необходимое зло
    (цена масштабируемости)



Когда используют микросервисы

- Продвигают компании типа Netflix, Amazon
- Интернет-проекты, а не кровавый Enterprise
- Сложность – много пользователей
- Предметная область проще
  - Проще проводить границы микросервисов
  - Сложнее при этом накосячить
  - Проще переделать границы или жить с теми, что есть

«Микросервисы, но» в Enterprise

- Все в одной репе (у всех одна версия)
- Все работают с одной базой
- Все деплоятся одновременно

Когда делать монолиты

- Стартап
  - Чем быстрее на рынок, тем лучше
- Внутренняя автоматизация (тот самый Enterprise)
  - Количество пользователей – десятки, максимум сотни
  - База в Гб, а не Тб
  - Сложные бизнес-процессы
- Не бизнес-приложения
- Встраиваемые системы

Как ускорять монолиты

- Вертикальное масштабирование
- CQRS
  - Денормализация, отдельная Read модель
  - Возможно отдельная Read база
- Горизонтальное масштабирование
  - Еще один ApplicationServer и LoadBallancer
  - Еще один DB Server и синхронизация баз
  - https://youtu.be/vvMfFFTd94E

Зачем нужен модульный монолит

- Сохранить достоинства монолита
  - Простоту разработки и отладки
  - Простоту деплоя
  - Согласованность данных
- Добавить изоляцию модулей, как в микросервисах

Растет проект

- Новые отделы
- Новые процессы
- Новый смысл старых терминов
  - клиент у бухгалтеров и маркетологов
- Выделяем Bounded Contexts

Растет команда

- Все уже не могут знать и делать всё
- Выделяем команды и их зоны ответственности
- Проводим технические границы в проекте

Проект возможно станет HighLoad

- Но нет опыта в предметной области
- Но нет опыта в микросервисах
- Но возможно и не станет HighLoad ☺

- Проводим логические, а не физические границы модулей
  - Проще нащупать эти границы
  - Проще перейти к микросервисам

Предварительный этап

- Подсистемы – в отдельные процессы
  - Репортинг
  - Планировщик задач
  - Интеграции, email/смс рассылки

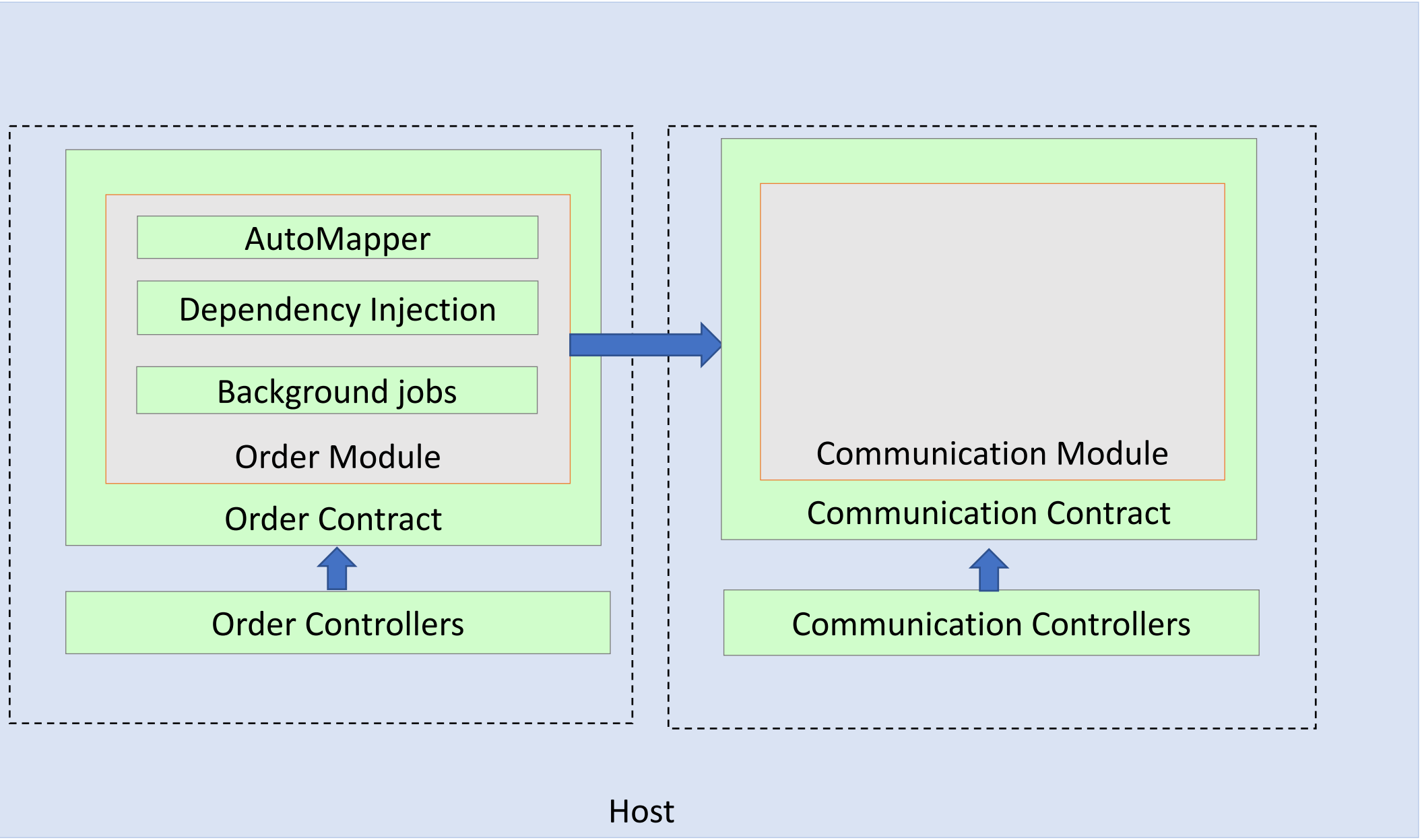
- Модули – предварительный этап

Пример: создание заказа

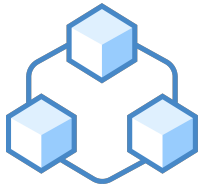
```
protected override async Task Handle(CreateOrderRequest request)
{
    var order = _mapper.Map<Order>(request.CreateOrderDto);
    _dbContext.Orders.Add(order);

    var newMail = new Email
    {
        Address = _currentUserService.Email,
        Subject = "Order created",
    };
    _dbContext.Emails.Add(newMail);

    await _dbContext.SaveChangesAsync();
}
```

В монолите

- Один ORM контекст (UnitOfWork)
  - В нем и Orders, и Communication
- Бизнес-сценарий реализован в одном месте
  - методе сервиса или хендлере
- Согласованность данных


AutoMapper
Dependency Injection
Background jobs
Order Module
Order Contract
Order Controllers
Communication Module
Communication Contract
Communication Controllers
Host

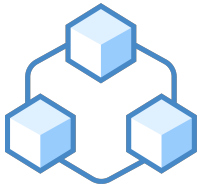
Чем модуль похож на микросервис



- Открытый контракт для других модулей и контроллеров
- Модуль инкапсулирован
  - напрямую методы и классы нельзя использовать из других модулей

- Но это все еще монолит
  - все модули в одном процессе
  - используют одну инфраструктуру хост-процесса (DI контейнер)

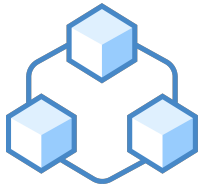
Чем модуль похож на плагин



- Открыта инфраструктура
  - DI модуль
  - Automapper Profile
  - Background Job registry
- С помощью инфраструктуры хост подключает модуль

- Но в отличие от плагинов
  - модули взаимодействуют друг с другом
  - нет динамической подгрузки

Хост берет на себя инфраструктуру



- Транзакции
- Авторизация и аутентификация
  - не надо передавать между микросервисами инфу о пользователе
- Генерация URL
- Логгирование
  - не надо передавать CorrelationId между микросервисами
- Возврат HTTP кодов в соответствии с результатами обработки запроса

Регистрация модулей в DI хоста

```
services.AddAutoMapper(
    typeof(OrdersAutoMapperProfile),
    typeof(CommunicationAutoMapperProfile));


services.AddControllers()
   .AddApplicationPart(typeof(OrdersController).Assembly)
   .AddApplicationPart(typeof(CommunicationsController).Assembly);


JobManager.Initialize(new CommunicationJobRegistry());


//И все сервисы их модулей
```

Нужен ли инициализатор модуля?

```
public static class OrderModuleInitializer
{
    public static void AddOrderModule(this IServiceCollection services)
    {
        //Register all
    }
}
```

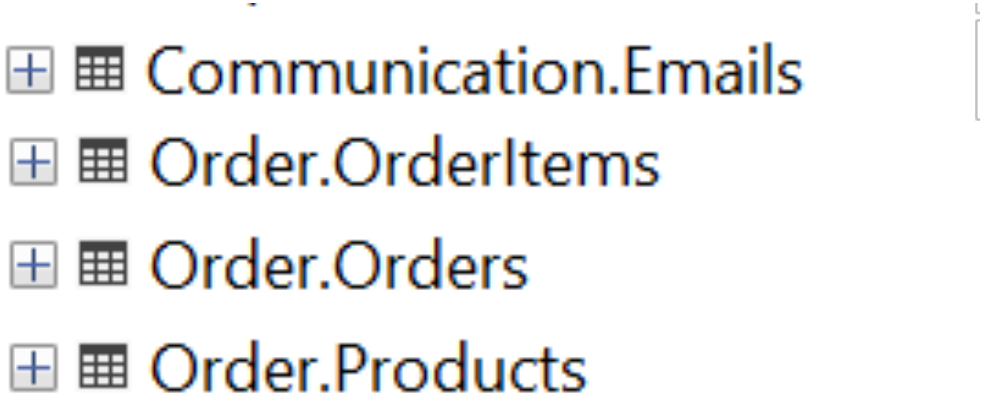
Инициализатор модуля – дело вкуса

- Если не хочется – можно не делать)
  - Явных плюсов нет
- Если очень хочется – можно сделать)
- Главное – не делать лишнего
  - Отдельный DI контейнер на каждый модуль
  - Отдельные экземпляры общих сервисов на каждый модуль

Доступ к данным

- Изоляция модулей: у каждого
  - свой ORM контекст
  - своя модель данных
  - своя схема в базе
- Но
  - все модули работают в одной транзакции бд
  - одна база и целостность данных при помощи внешних ключей

Отдельная схема на каждый модуль

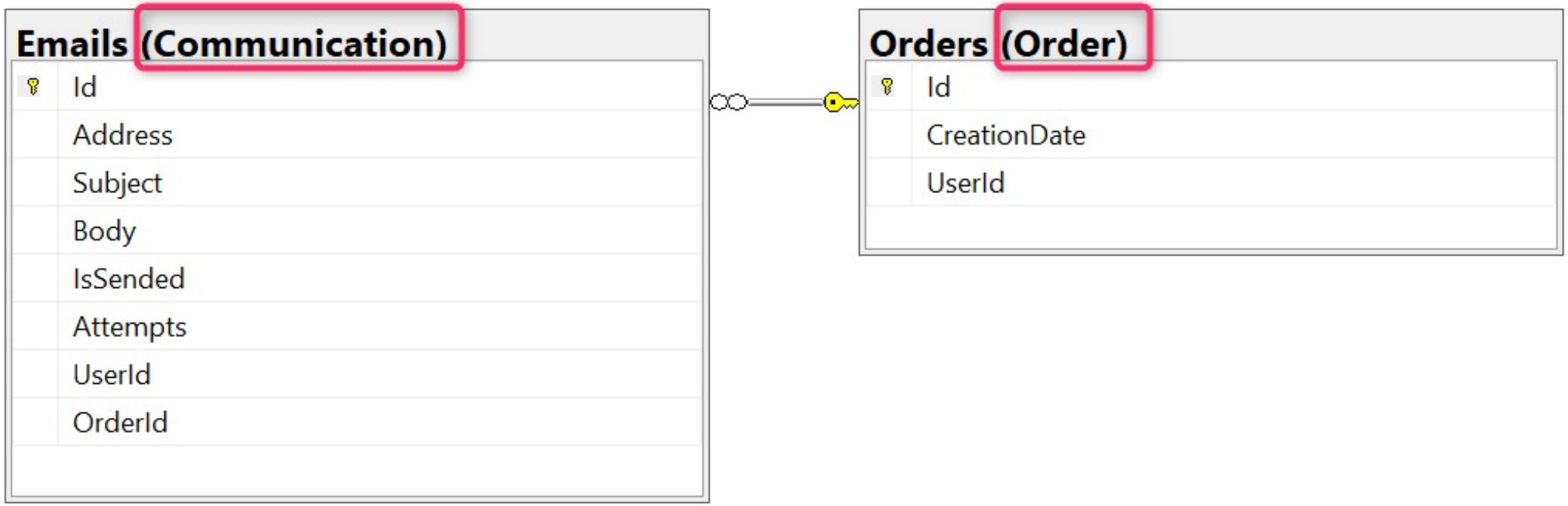



```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Communication");
}
```

Нет NavigationProterties между модулями

```
internal class Email
{
    public string Address { get; set; }

    public int OrderId { get; set; }
    public Order Order { get; set; }
}
```

Ключам между схемами быть!



Конвенция именования для миграций

	MigrationId	ProductVersion
1	20200315150200_Order_Initial	3.1.0
2	20200315150453_Communication_Initial	3.1.0

```
public partial class Order_Initial : Migration
{
}
```

PS скрипт для накатывания всех миграций

```
dotnet ef database update
  --context OrderDbContext
  --project Order\Shop.Order.DataAccess.MsSql
  --startup-project Shop.Web
dotnet ef database update
  --context CommunicationDbContext
  --project Communication\Shop.Communication.DataAccess.MsSql
  --startup-project Shop.Web
```

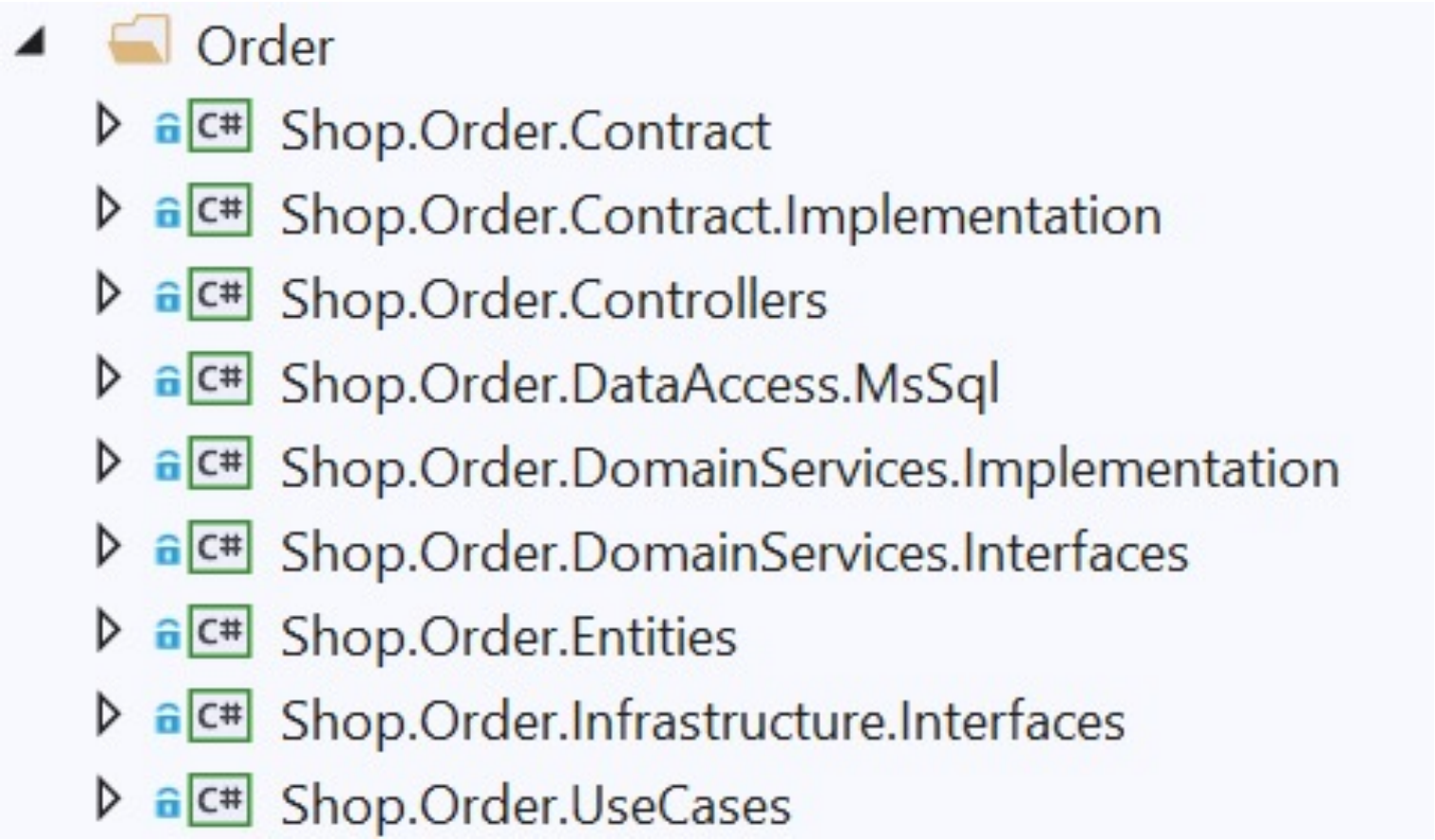
Нужен порядок модулей

В зависимостях модулей не должно быть циклов (арх. тест)

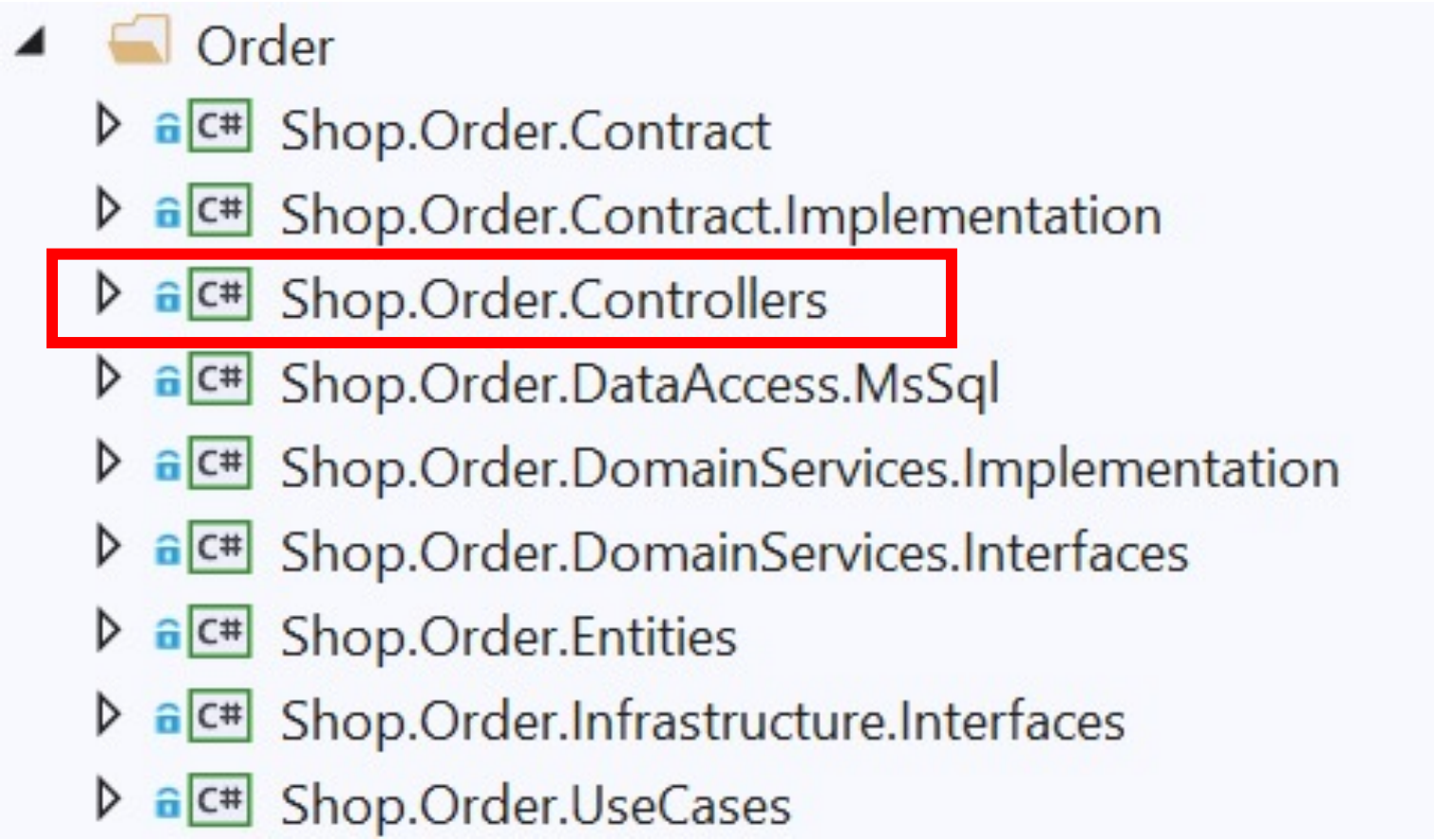
Проект


Solution 'ModularMonolith
Communication
Framework
Order
Shop.Web

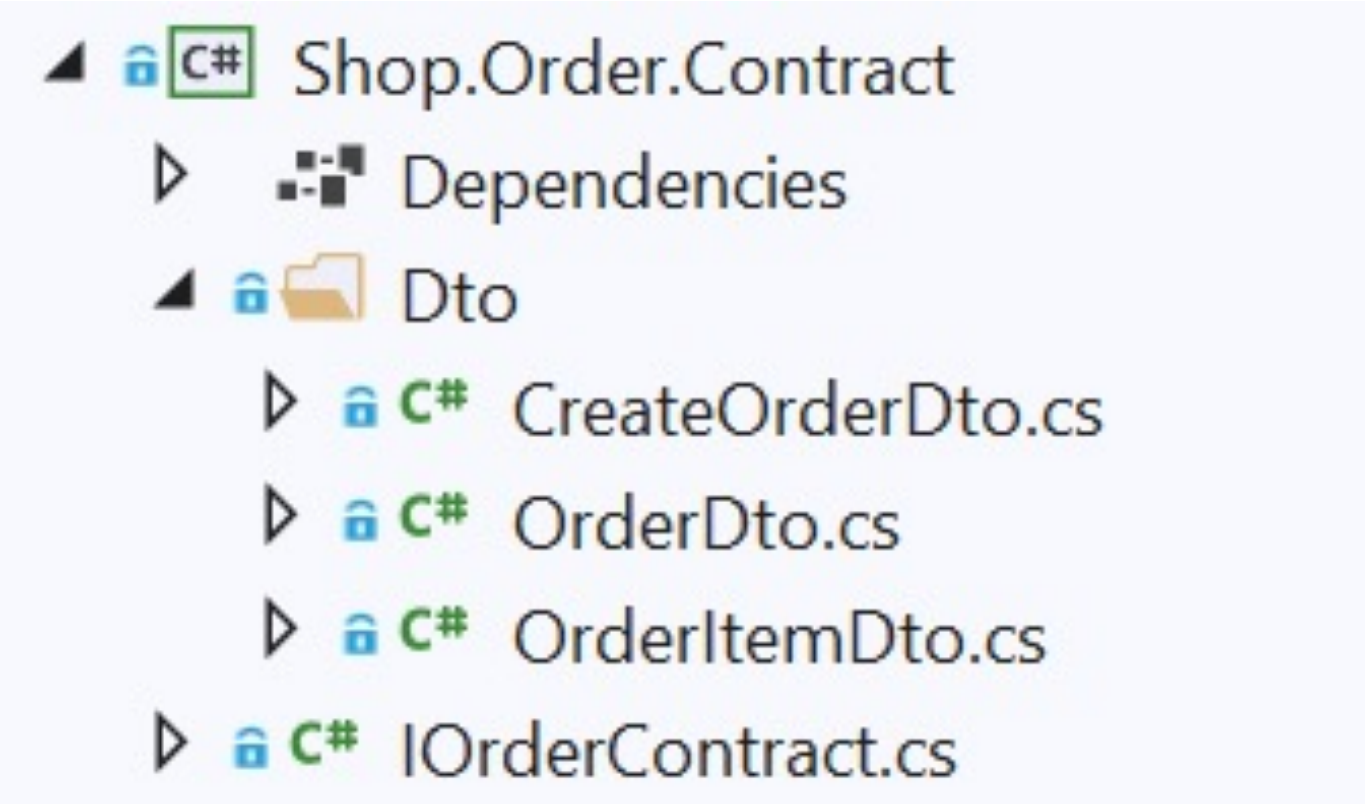
Модуль


Order
Shop.Order.Contract
Shop.Order.Contract.Implementation
Shop.Order.Controllers
Shop.Order.DataAccess.MsSql
Shop.Order.DomainServices.Implementation
Shop.Order.DomainServices.Interfaces
Shop.Order.Entities
Shop.Order.Infrastructure.Interfaces
Shop.Order.UseCases

Контроллер – часть модуля



Контракт – интерфейс и DTO



Контракт модуля

```
public interface IOrderContract
{
    Task<int> CreateOrderAsync(CreateOrderDto createOrderDto);

    Task<OrderDto> GetOrderAsync(int orderId);
}
```

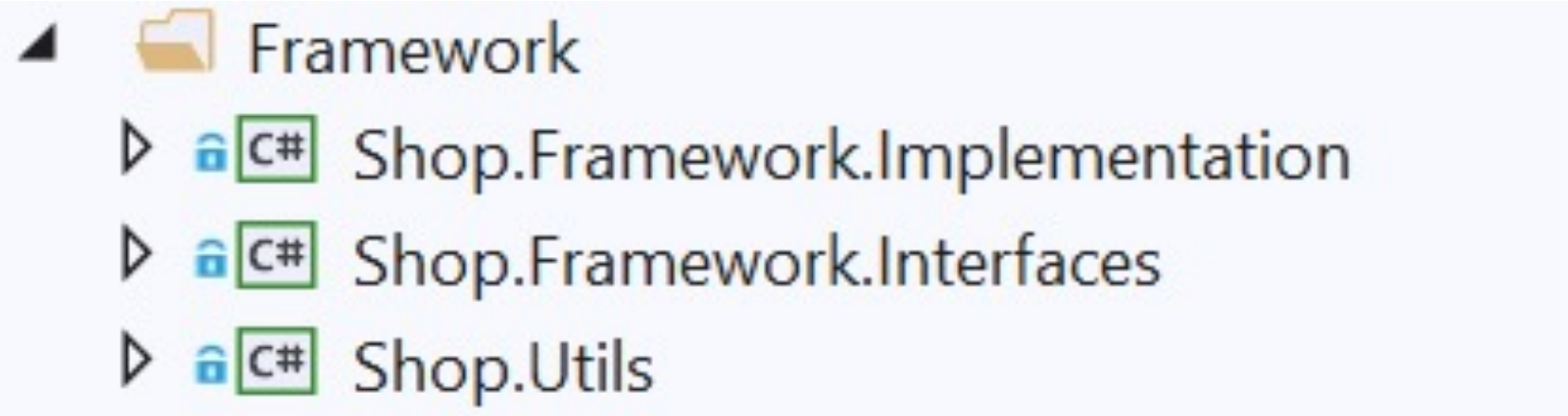
Реализация контракта - фасад

```
internal class OrderContract : IOrderContract
{
    public async Task<OrderDto> GetOrderAsync(int orderId)
    {
        return await _mediator.Send(new GetOrderRequest { Id = orderId });
    }
}
```

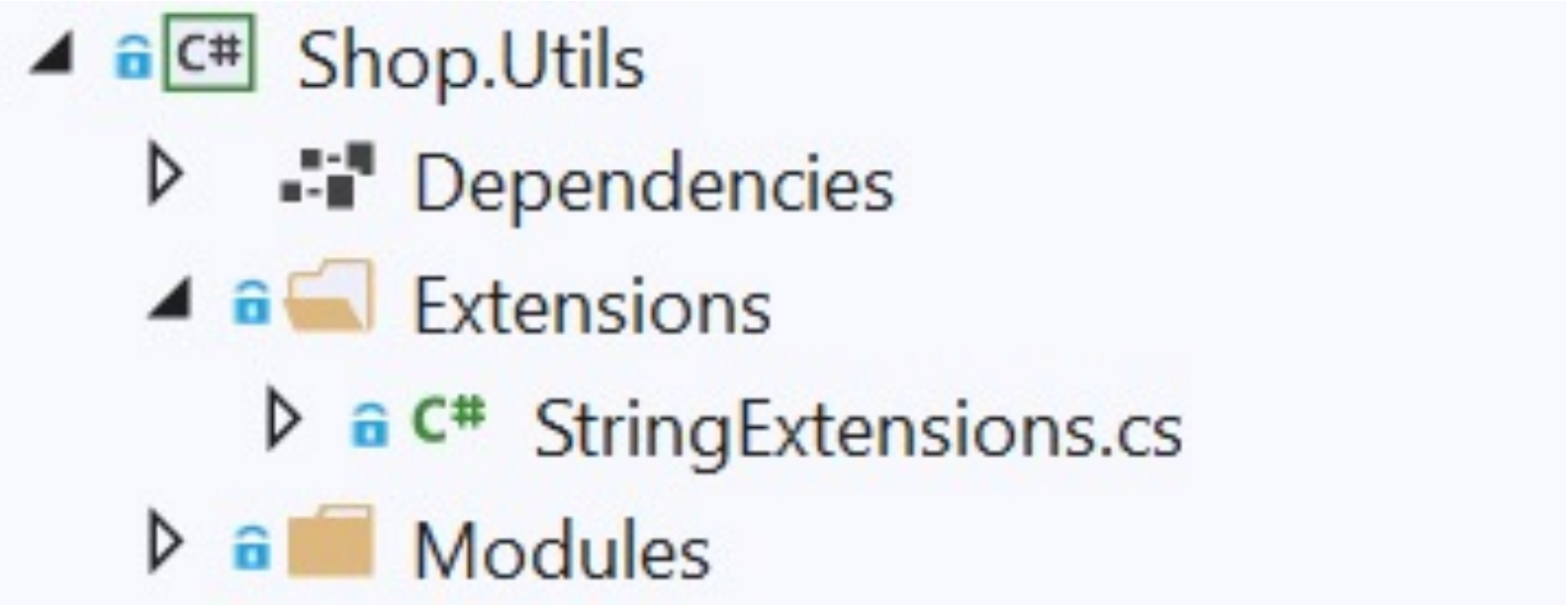
DTO - POCO для контроллеров или др. модулей

```
public class OrderDto
{
    public int Id { get; set; }

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

Фреймворк


Framework
Shop.Framework.Implementation
Shop.Framework.Interfaces
Shop.Utils

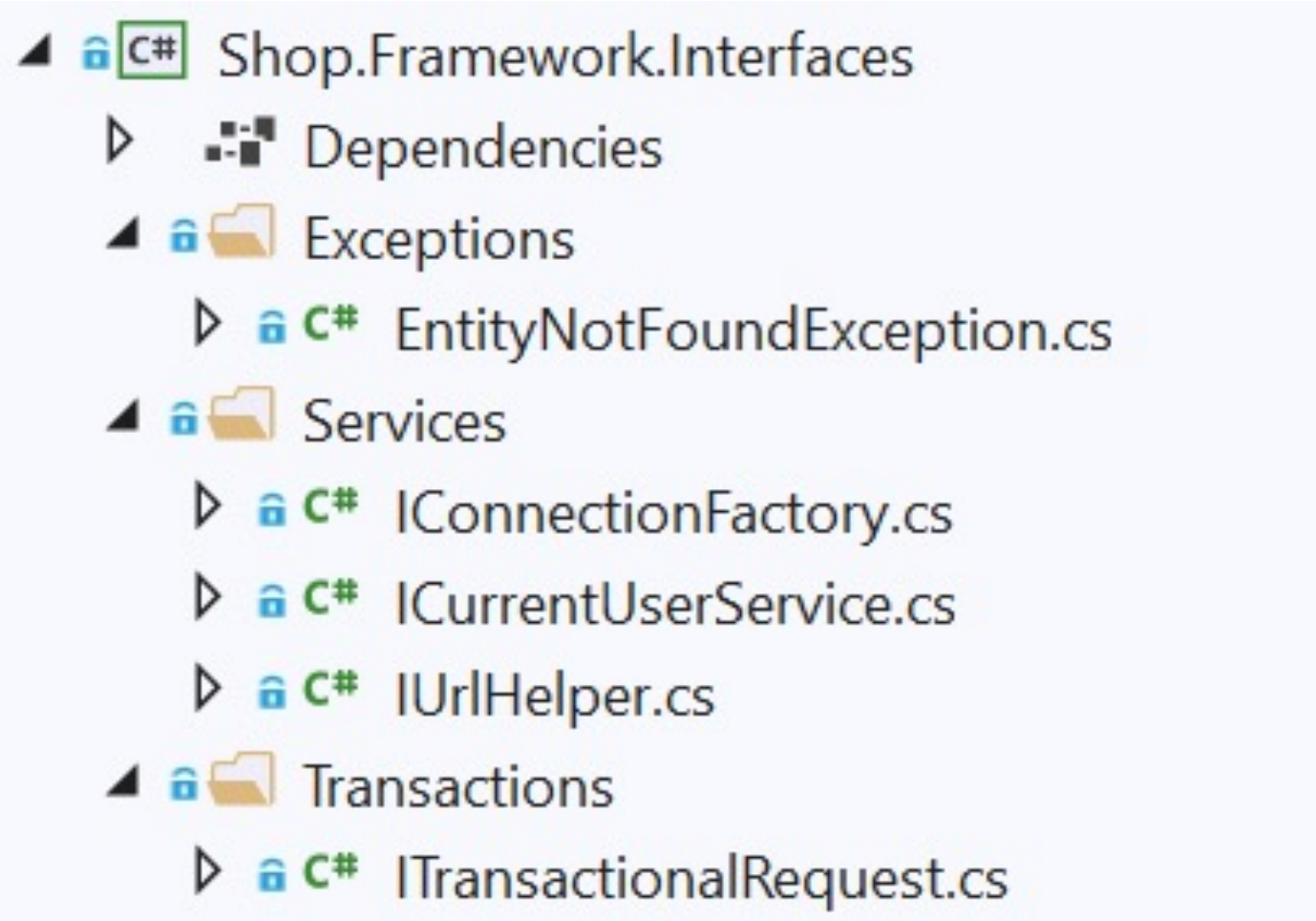
Utils – тележка с полезностями



StringExtensions

```
public static class StringExtensions
{
    public static bool IsEmpty(this string value)
    {
        return string.IsNullOrEmpty(value);
    }
}
```

Framework.Interfaces



OrderModule: создание заказа

```
public async Task<int> Handle(CreateOrderRequest request)
{
    var order = _mapper.Map<Order>(request.CreateOrderDto);
    _dbContext.Orders.Add(order);

    await _dbContext.SaveChangesAsync();

    await _communicationContract
        .SendEmailAsync(_currentUserService.Email, "Order created")

    return order.Id;
}
```

CommunicationModule: отправка email

```
internal class SendEmailRequestHandler : AsyncRequestHandler<OrderCreatedMessage>
{
    protected override async Task Handle(OrderCreatedMessage request)
    {
        var newMail = new Email
        {
            Address = request.Address,
            Subject = request.Subject,
            Body = request.Body
        };

        _dbContext.Emails.Add(newMail);

        await _dbContext.SaveChangesAsync();
    }
}
```

Почему не доменные события?

- DomainEvents – дань моде DDD
- Если нет миграции к микросервисам – они не нужны

DbTransactions

- Один ConnectionString и одно подключение на все модули
- Один пользователь с правами на всю базу
  - Ручной Join между модулями сработает
- Транзакционность - АОП или пайплайны

Пример настройки пайплайна 20 строк кода

```
public class DbTransactionPipelineBehavior<TRequest, TResponse> :
       IPipelineBehavior<TRequest, TResponse>  where TRequest : ITransactionalRequest
{
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate next)
    {
        if (_connectionFactory.IsTransactionStarted)
            return await next();

        await using var connection = _connectionFactory.GetConnection();
        await using var transaction = _connectionFactory.GetTransaction();

        var result = await next();

        transaction.Commit();

        return result;
    }
}
```

Autofac для Connection

```
containerBuilder.Register(
        componentContext =>
        {
            var optionsBuilder = new DbContextOptionsBuilder<TDbContext>();
            var connectionFactory = componentContext.Resolve<IConnectionFactory>(
            optionsBuilder.UseSqlServer(connectionFactory.GetConnection());
            return optionsBuilder.Options;
        })
    .InstancePerLifetimeScope();
```

Autofac для Transaction

```
containerBuilder.RegisterType<TDbContext>()
    .OnActivated(args =>
    {
        var t = args.Context.Resolve<IConnectionFactory>().GetTransaction();
        args.Instance.Database.UseTransaction(t);
    })
    .InstancePerLifetimeScope()
    .As<TDbContextInterface>();
```

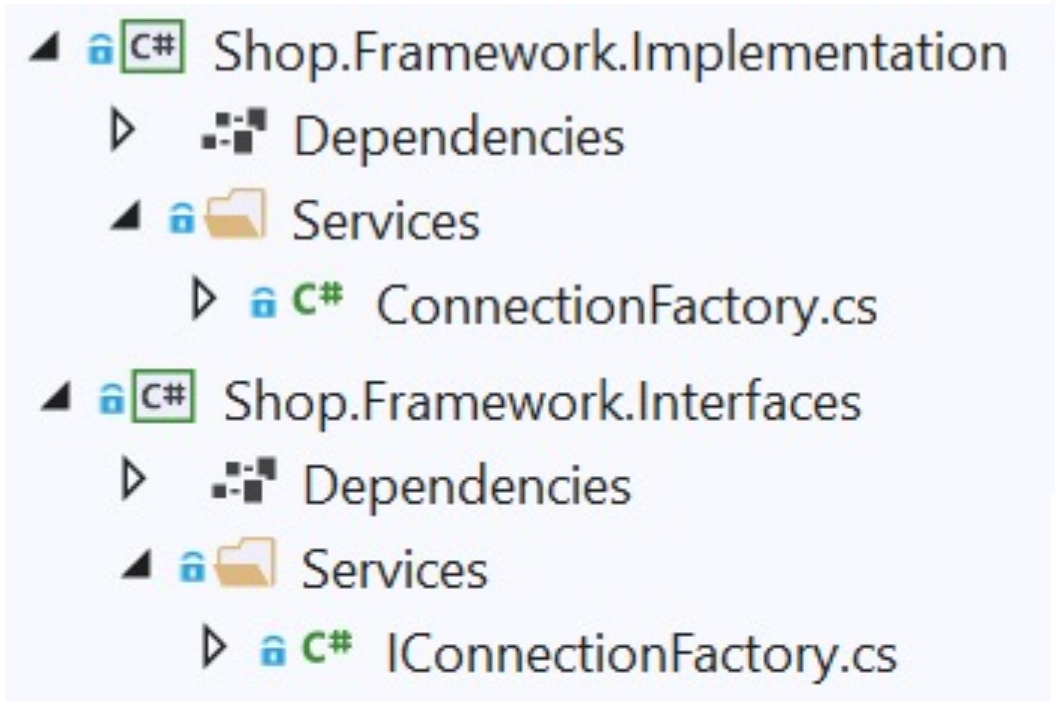
Можно более строго

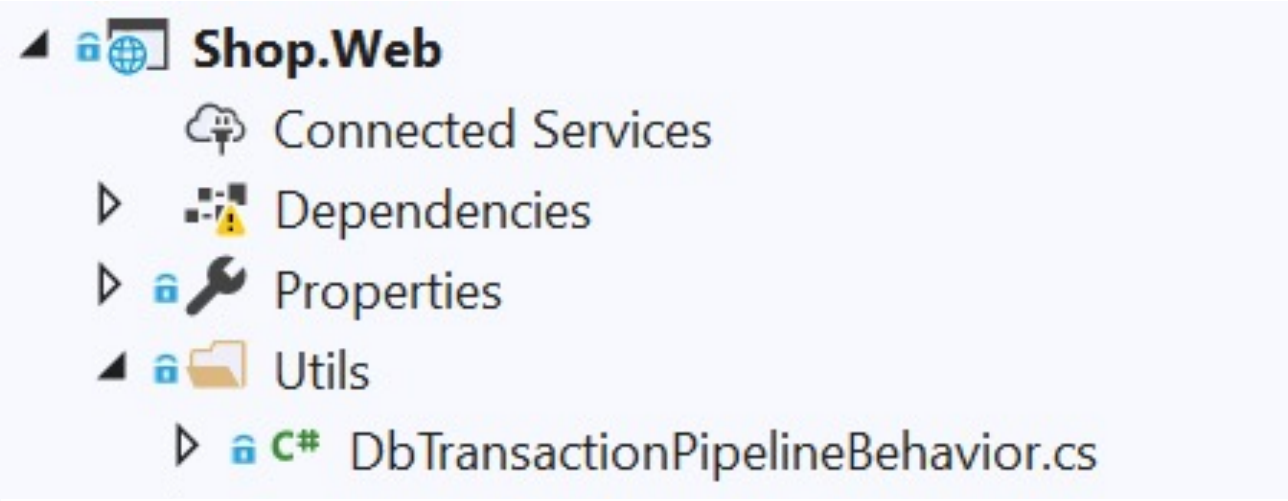
- Для каждого модуля
  - свое подключение
  - свой пользователь с правами только на схему этого модуля
- Ручной запрос с джойном по разным модулям не сработает ☺
- Не нужна инициализация DbContext транзакцией через DI
- Но нужны распределенные транзакции
  - Реализация зависит от платформы, ORM, операционной системы
  - Уложится в 20 строк кода

Распределенные транзакции

```
public class TransactionScopePipelineBehavior<TRequest, TResponse> :
    IPipelineBehavior<TRequest, TResponse> where TRequest : ITransactionalRequest
{
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate next)
    {
        if (_connectionFactory.IsConnectionOpened)
            return await next();

        using var scope = new TransactionScope(TransactionScopeOption.Required,
            new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted },
            TransactionScopeAsyncFlowOption.Enabled);

        await using var connection = _connectionFactory.GetConnection();
        var result = await next();
        scope.Complete();
        return result;
    }
}
```

Инфраструктура для транзакций БД



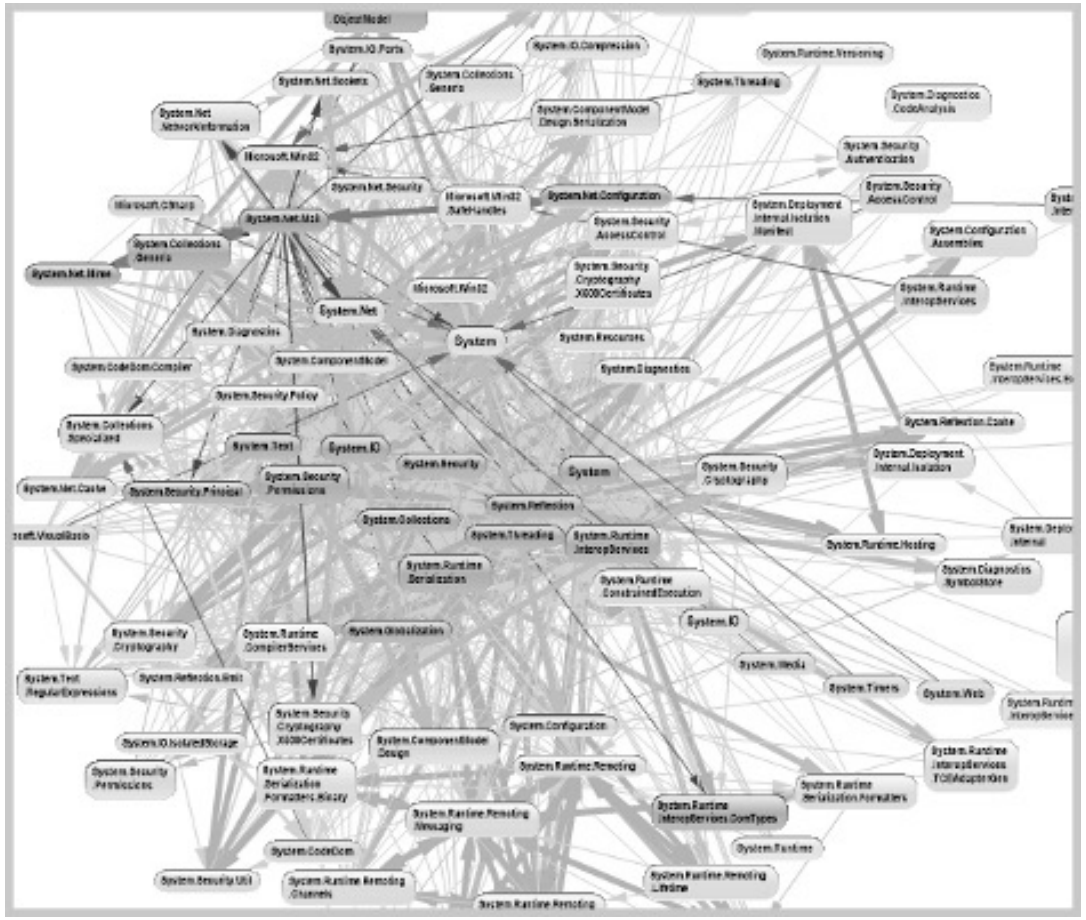


Кейс: финтех-проект

- Было:
  - Один продукт
  - Для внутреннего использования

- Надо:
  - Несколько продуктов (страховки, кредиты)
  - Несколько видов продуктов (КАСКО, ОСАГО)
  - Выход в онлайн

При этом

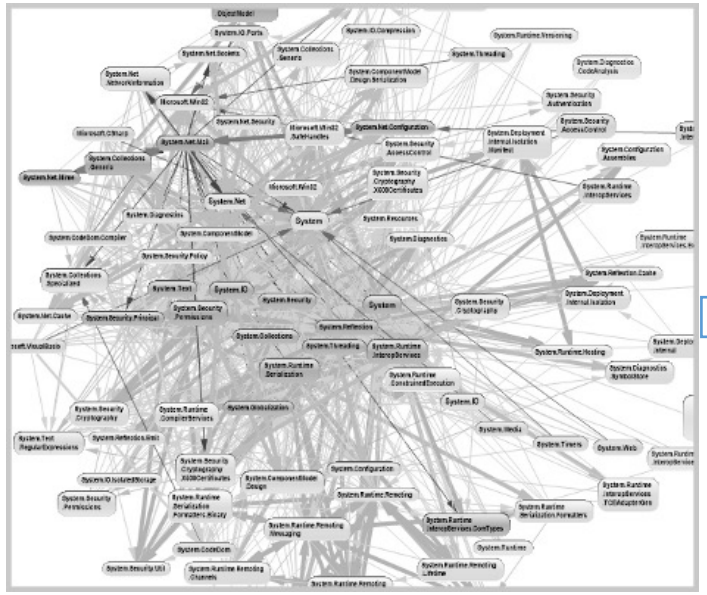


Расчет стоимости продукта:

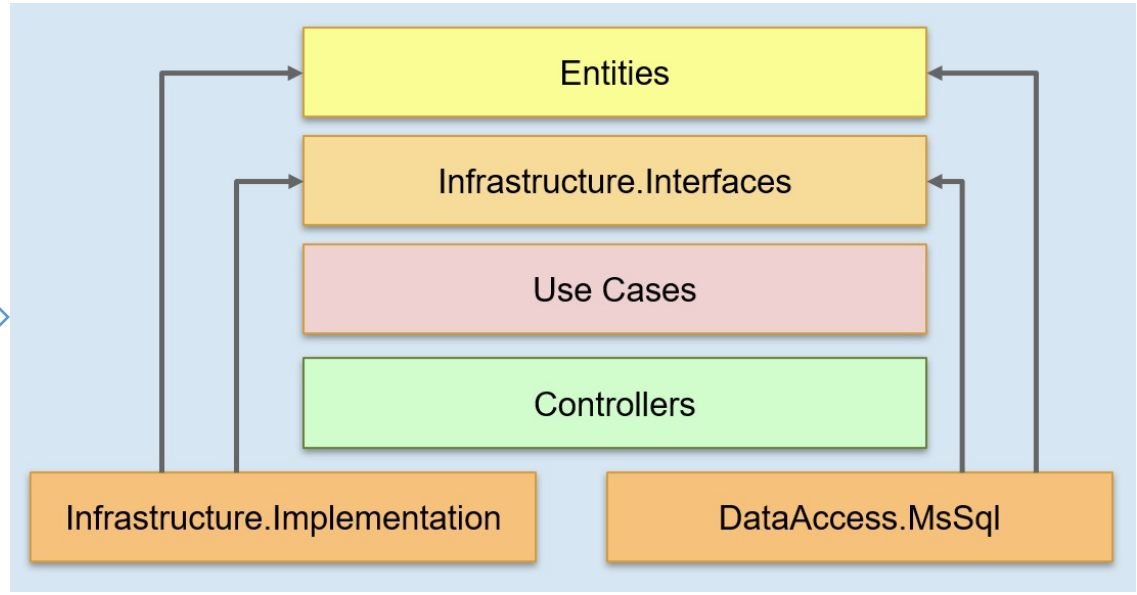
- Коммит данных в базу
- Расчет
- Откат транзакции ☺

Одна база для Application и Reporting

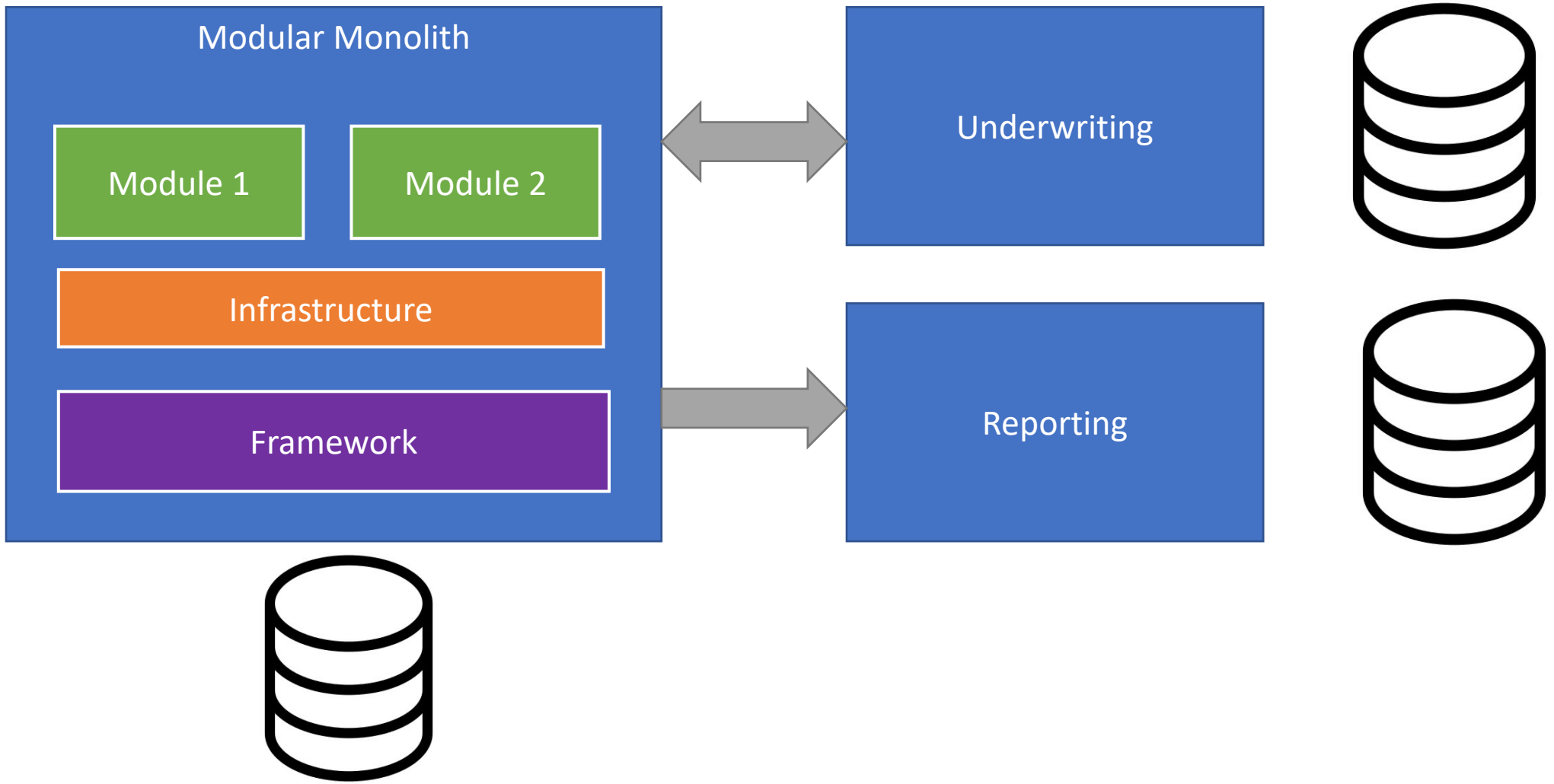
Первый шаг - рефакторинг



Clean Architecture



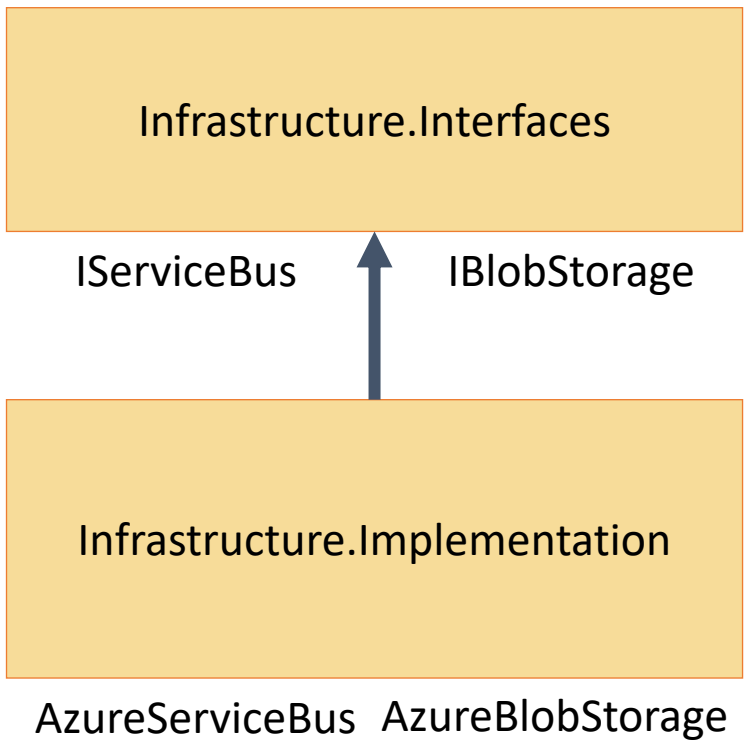
Что получили

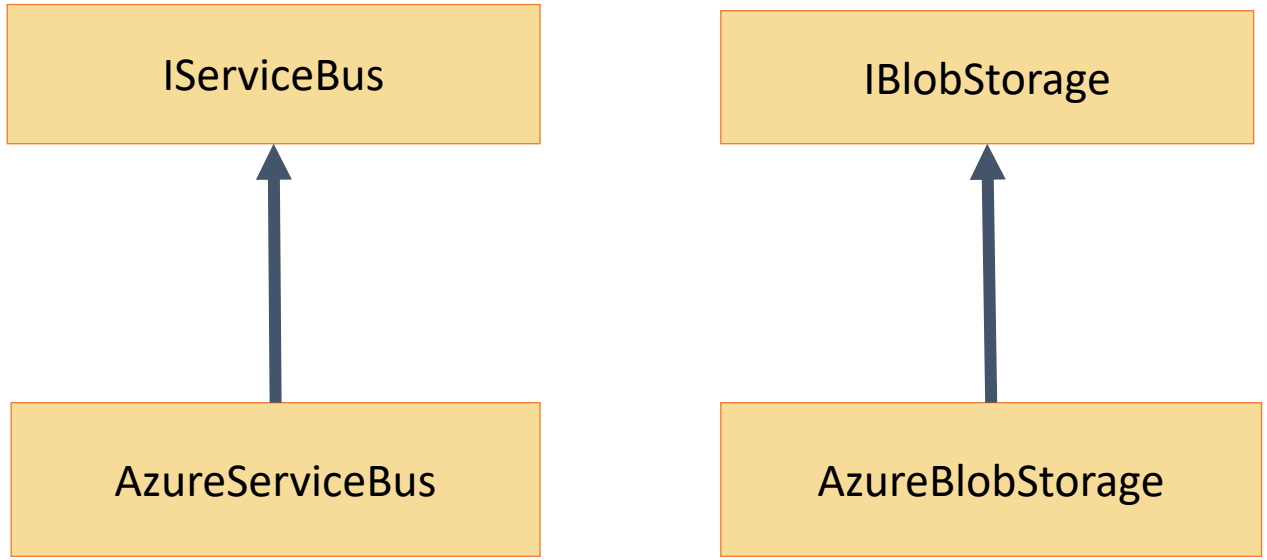


Инфраструктура

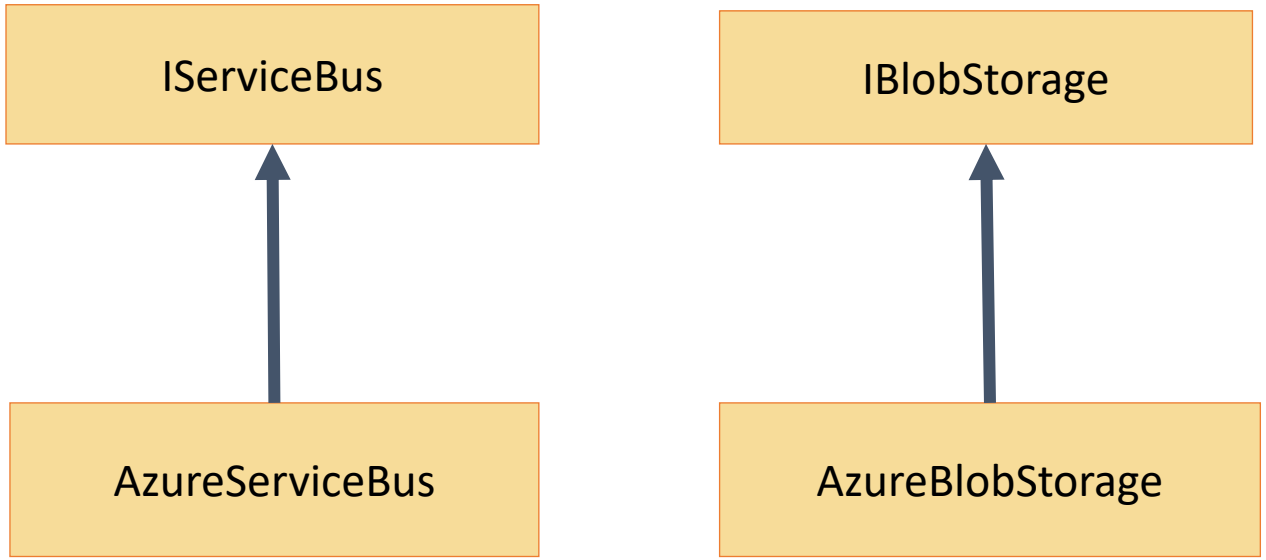
- Azure Service Bus
- Azure Blob Storage
- Интеграции (рассылка email)
- итд

Обычно – дело вкуса

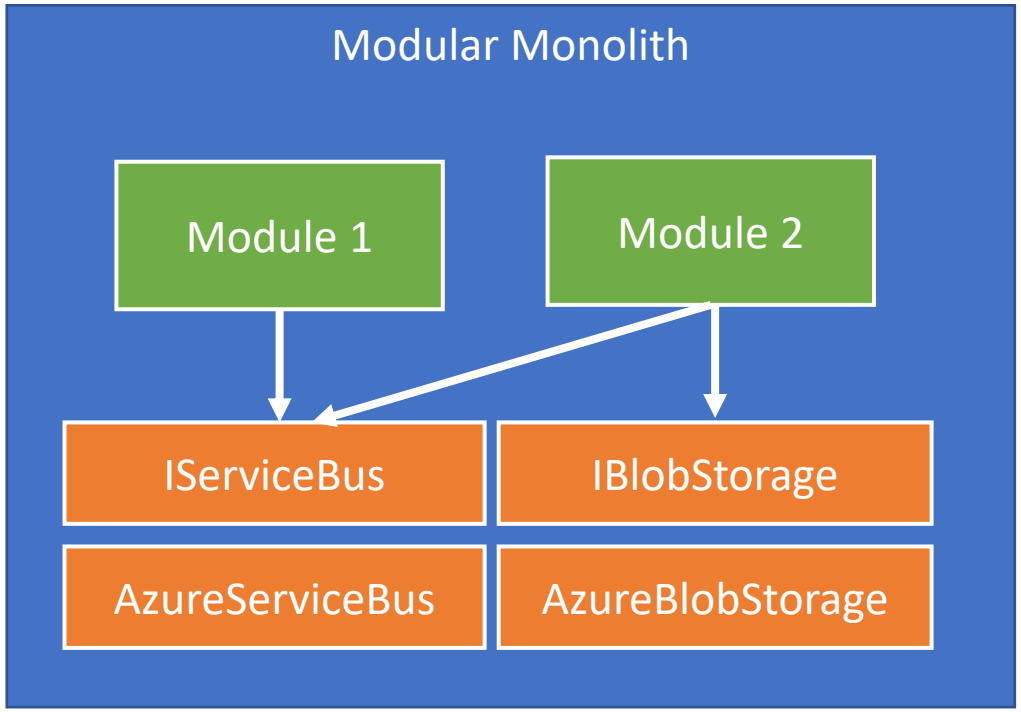




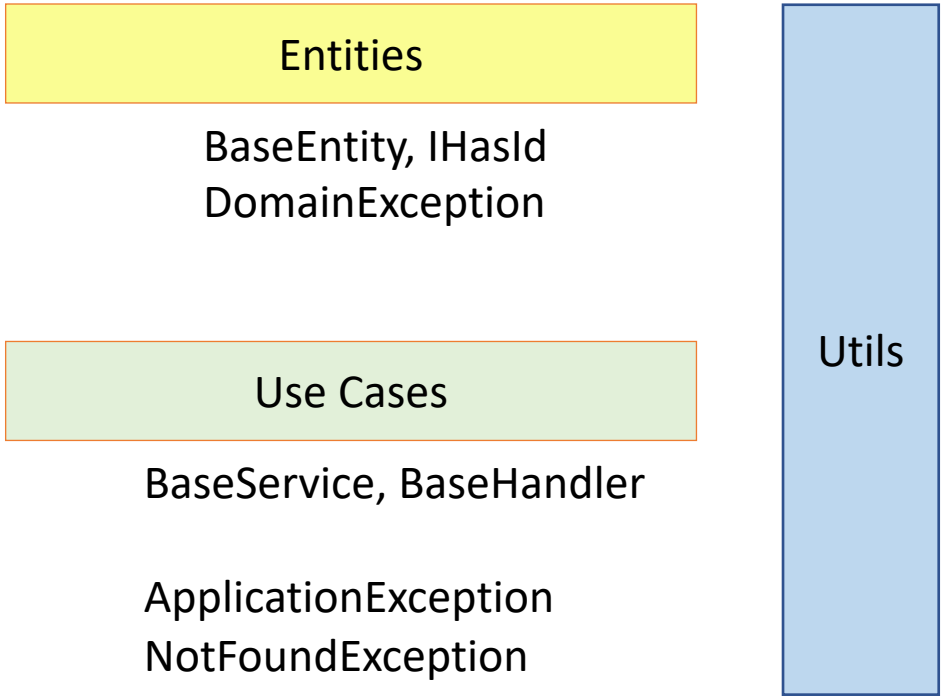
Модульный монолит



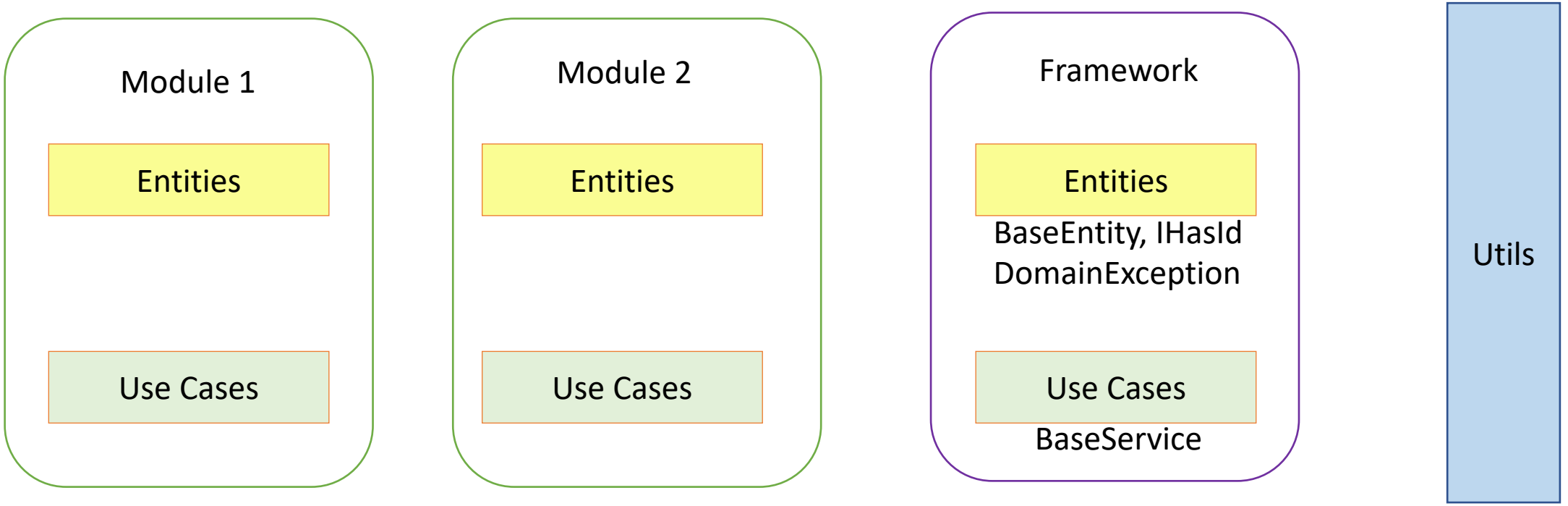
Разным модулям разная инфраструктура



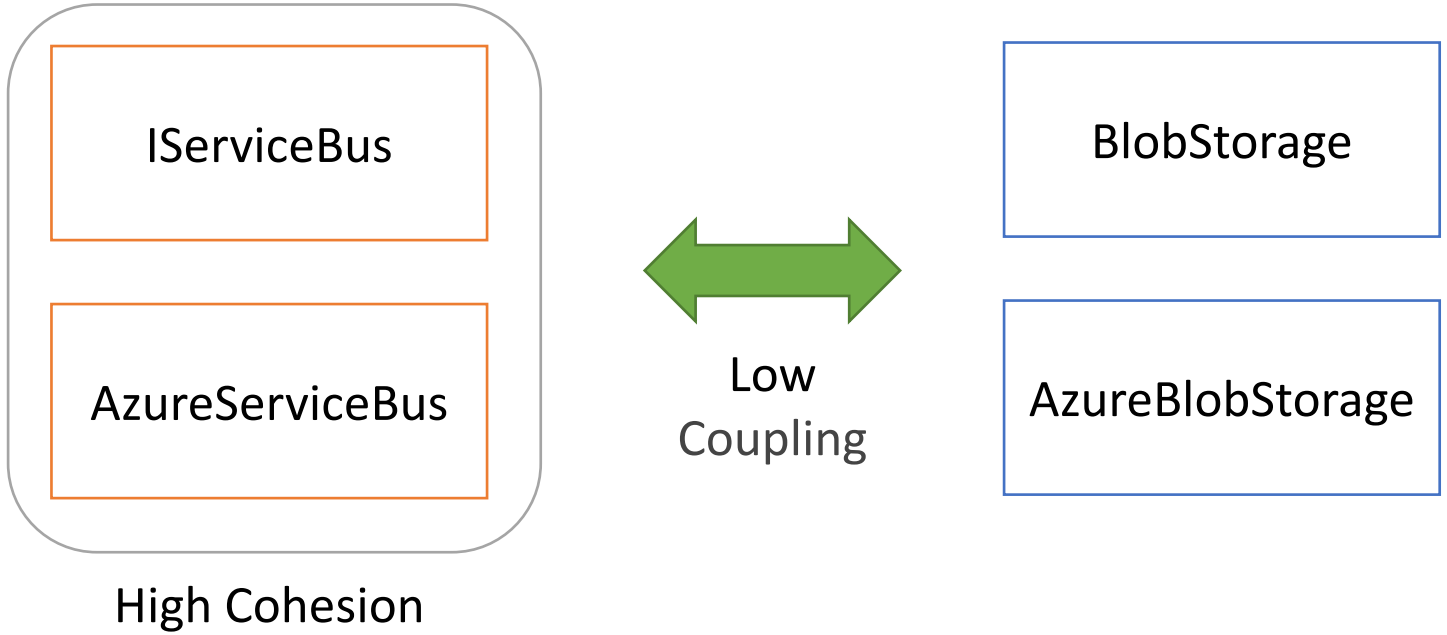
Монолит


Entities
BaseEntity, IHasId
DomainException
Use Cases
BaseService, BaseHandler
ApplicationException
NotFoundException
Utils

Модульный Монолит



Много маленьких компонентов



Изменения

- Отдельные модули для инфраструктуры
  - Проектов в солюшене будет больше!
- Отдельные модули для фреймворка
  - Проектов в солюшене еще больше!!
- Каждый модуль – Entities, DomainServices, UseCases
  - Проектов в солюшене намного больше!!!
- Архитектурные тесты на ссылки между проектами!
  - Рекорд мало поставить, надо удержать

- Не зря ли все это?

В каждом модуле

- Entities
- Application

- Их меньше
- Проще понимать
- Проще изменять и сопровождать
- Конечная цель всех рефакторингов достигнута

Достоинства модульных монолитов

- Мало инфраструктурного кода
  - транзакции
- Логические границы между модулями
  - Их проще нащупывать и менять
- В этом помогают
  - Простота отладки
  - Простота деплоя

О чем поговорили

- Когда рулит монолит
  - Стартап
  - Enterprise без претензии на HighLoad
  - Не Enterprise
- Когда нужны модули
  - Растет проект и его команда
  - Выносим части проекта в отдельные процессы для производительности
  - Нащупываем границы модулей в новой предметной области

Самое главное

- Можно без микросервисов
  - Изоляция модулей
  - Несколько команд
- Зачем нужен модульный монолит
  - Сохранить достоинства монолита
  - Добавить изоляцию модулей
- Как перейти к модульному монолиту
  - Отдельные схемы на уровне базы
  - Отдельные модули на уровне кода
  - Сохраняем бд транзакции и ссылки между таблицами

Хорошие источники доп. информации

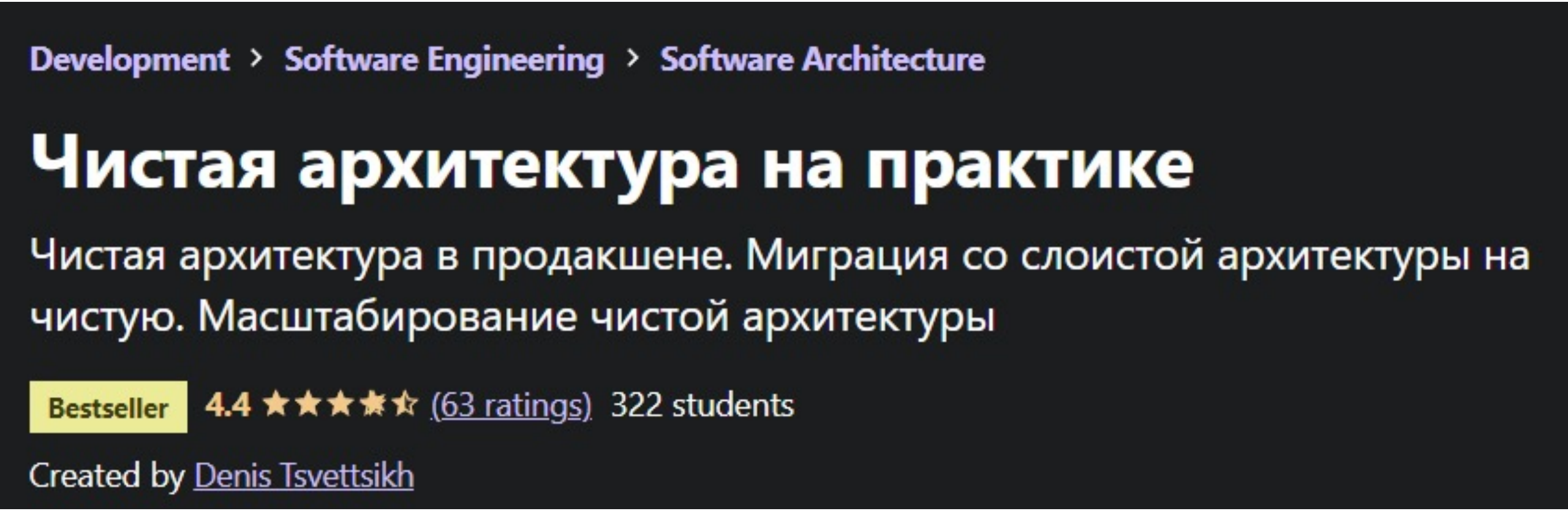
- Рихтер о микросервисах
  https://youtu.be/fiwlcb6a8EQ
- Пример кода на .NET
  https://github.com/denis-tsv/ModularMonolith
- Как защитить чистоту архитектуры

  https://youtu.be/eZnuzpktT4Q

Еще источники информации

- Simon Brown про модульный монолит (введение в тему)
  https://youtu.be/5OjqD-ow8GE
- Majestic Modular Monoliths by Axel Fontaine (много идей, мало фактов)
  https://youtu.be/BOvxJaklcr0
- Пример «Как не надо делать модульный монолит»
  https://github.com/kgrzybek/modular-monolith-with-ddd
  (я открыл несколько issues с архитектурными проблемами)

https://www.udemy.com/course/clean-architecture-csharp-ru/



DOTNEXT

Спасибо ☺
Вопросы?

Денис Цветцих

denis.tsvettsih@yandex.ru
@den_tsvettsikh
https://github.com/denis-tsv