DOTNEXT 2021 PITER

Денис Цветцих

Invent

9 способов улучшить Enterprise архитектуру при помощи CQRS и Vertical Slices

Обо мне

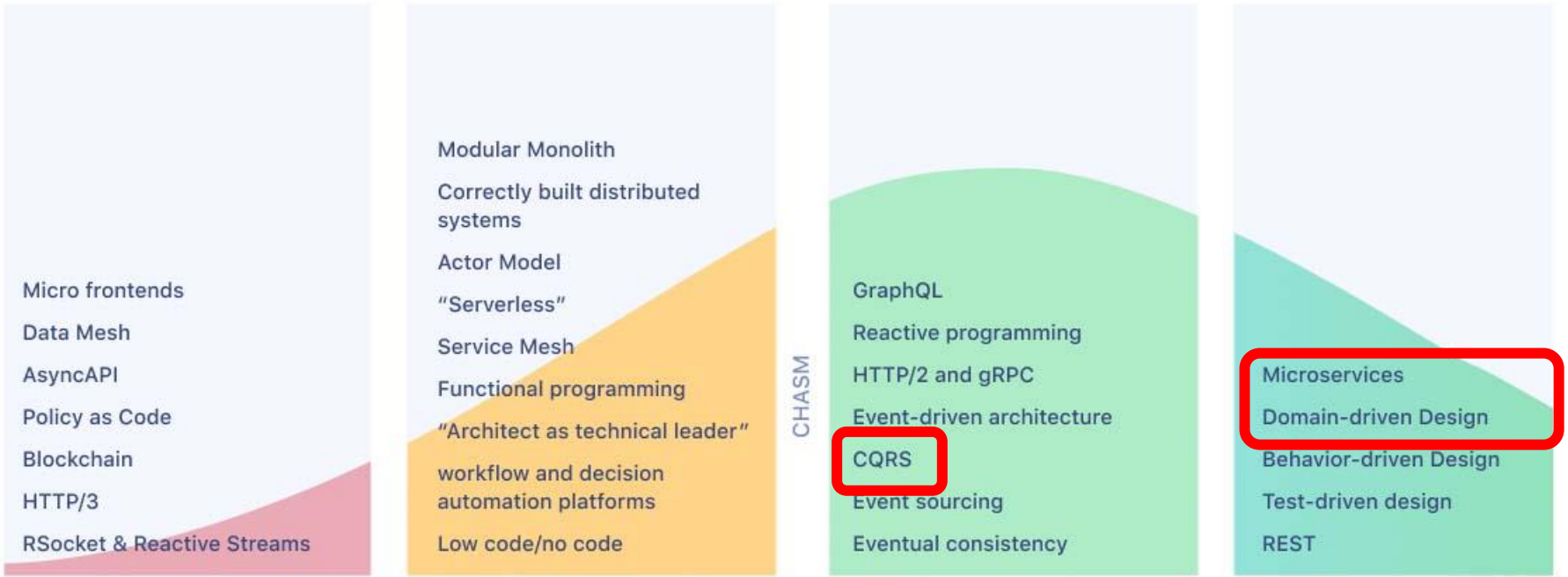
- Реализовал при помощи хендлеров/слайсов серию проектов
- За последний год отрефакторил два проекта со слоев на хендлеры
- Поделюсь полученным опытом ☺

О чем поговорим

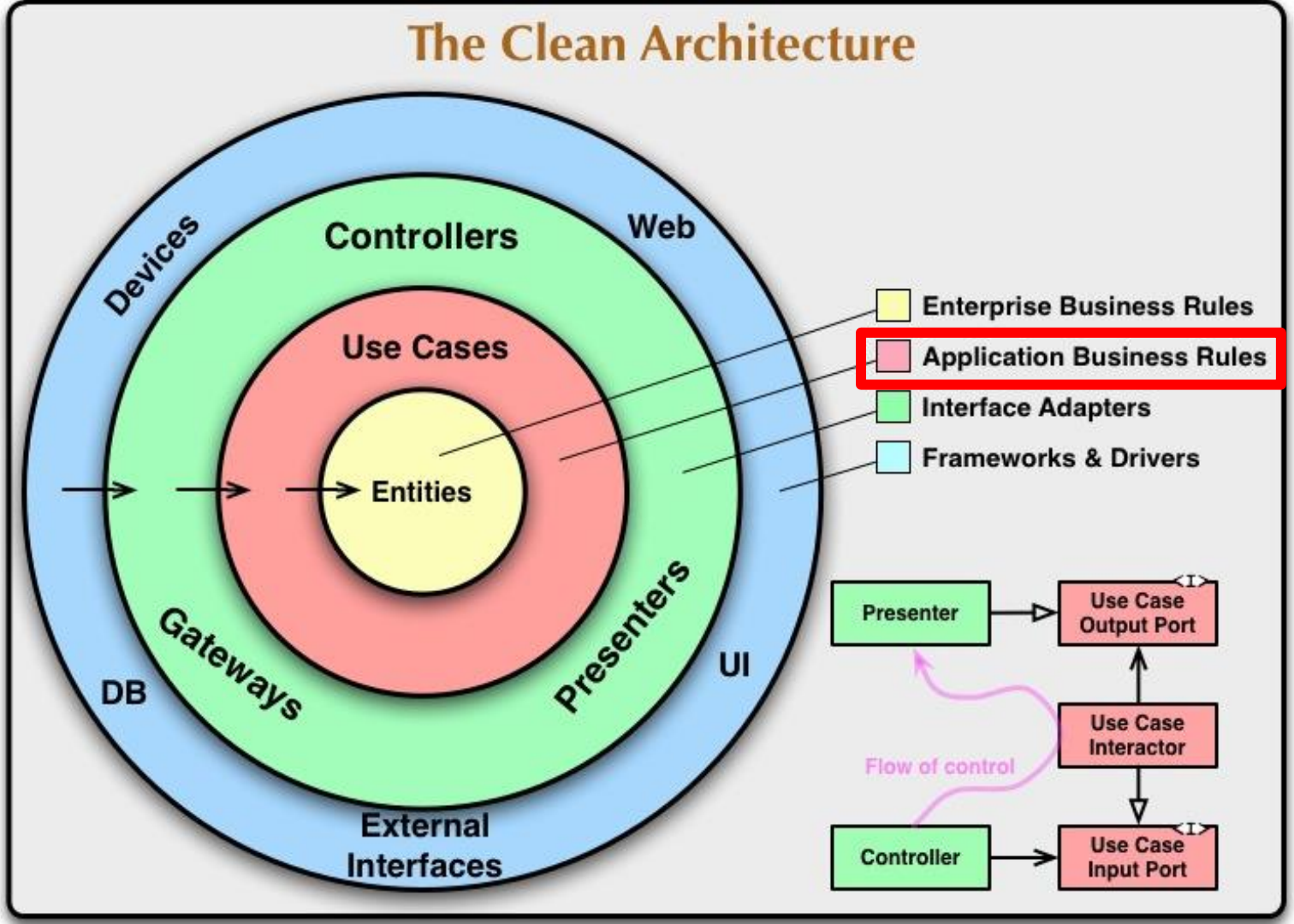
- Что такое CQRS-хендлер, Vertical Slice
- 9 достоинств хендлеров перед слоями
- Мифы и практические советы по реализации
- Как перейти от сервисов к хендлерам/слайсам

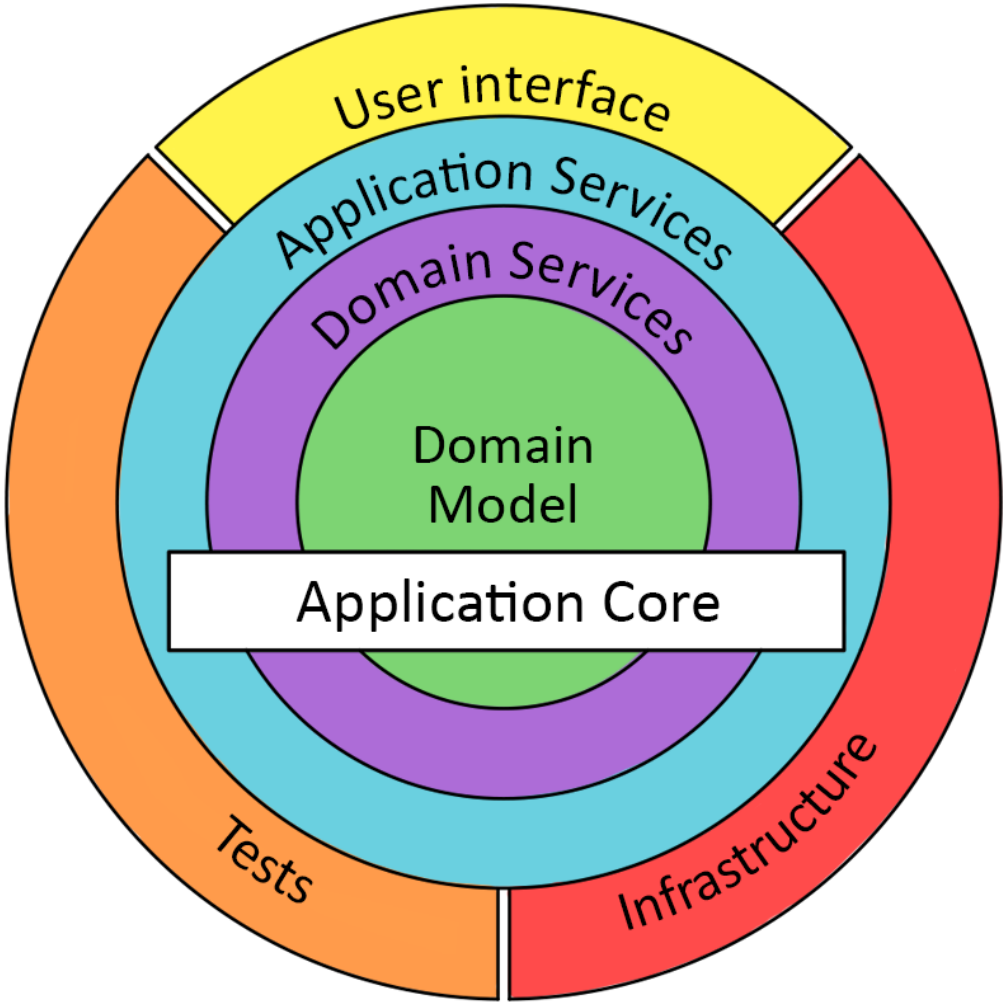
Software Development
Architecture and Design 2020 Q2 Graph
http://infoq.link/architecture-trends-2020

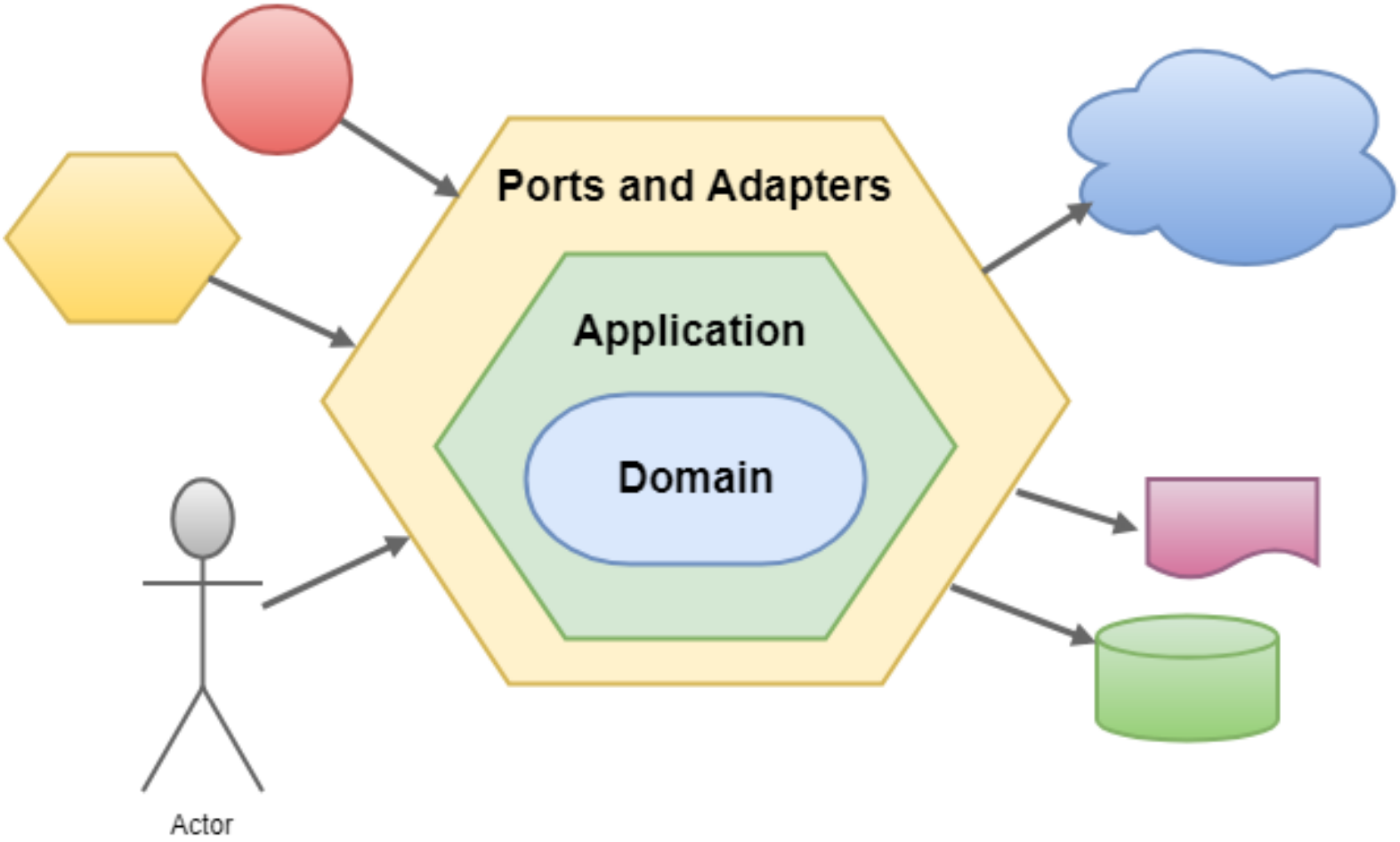
InfoQ

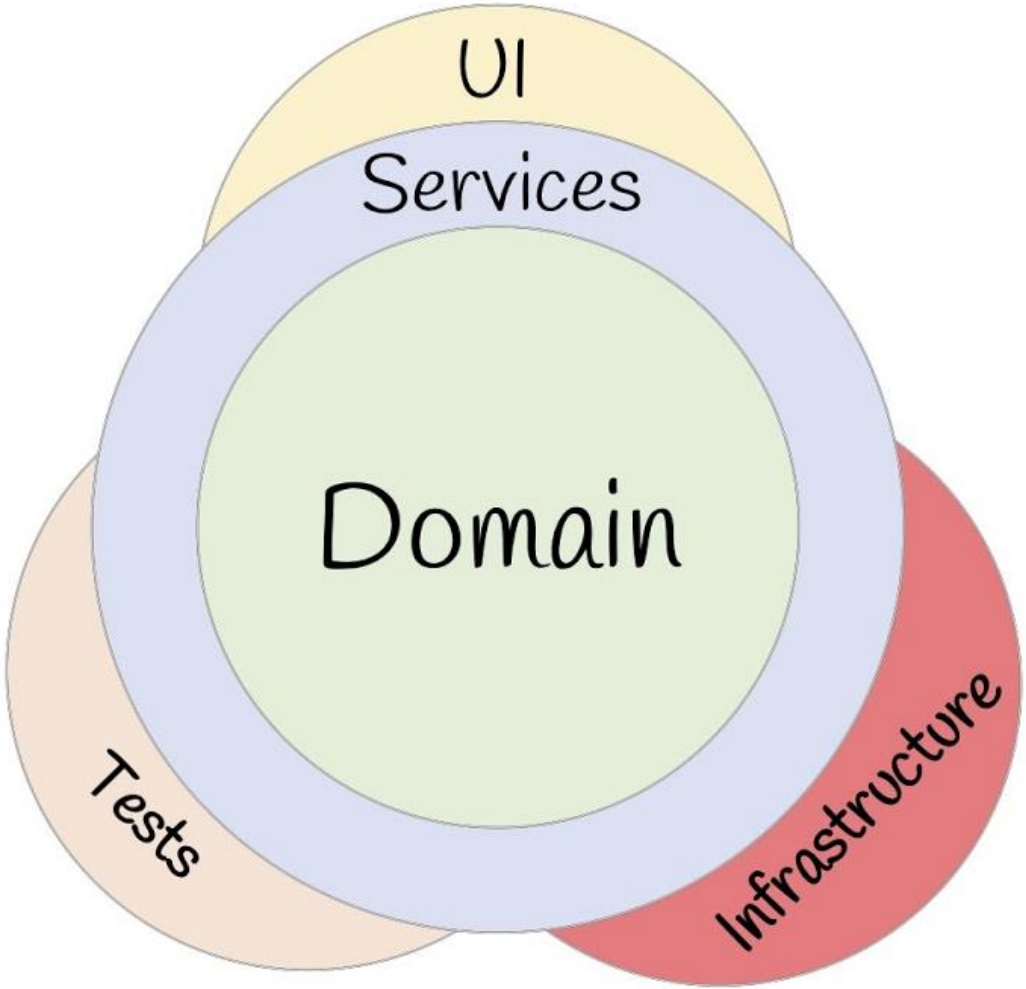


CQRS был изобретен для того, чтобы найти более эффективные способы создания сложных систем


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port


User interface
Application Services
Domain Services
Domain
Model
Application Core
Tests
Infrastructure


Ports and Adapters
Application
Domain
Actor


UI
Services
Domain
Tests
Infrastructure

Везде есть Application

- Называется по-разному
  - ApplicationServices
  - UseCases (Interactor)

- В Application больше всего бардака
  - DataAccess – ORM
  - Controllers – тонкие
  - Domain – голые правила без инфраструктуры

- Так что как реализовать Application – очень важный вопрос!

Какие есть варианты

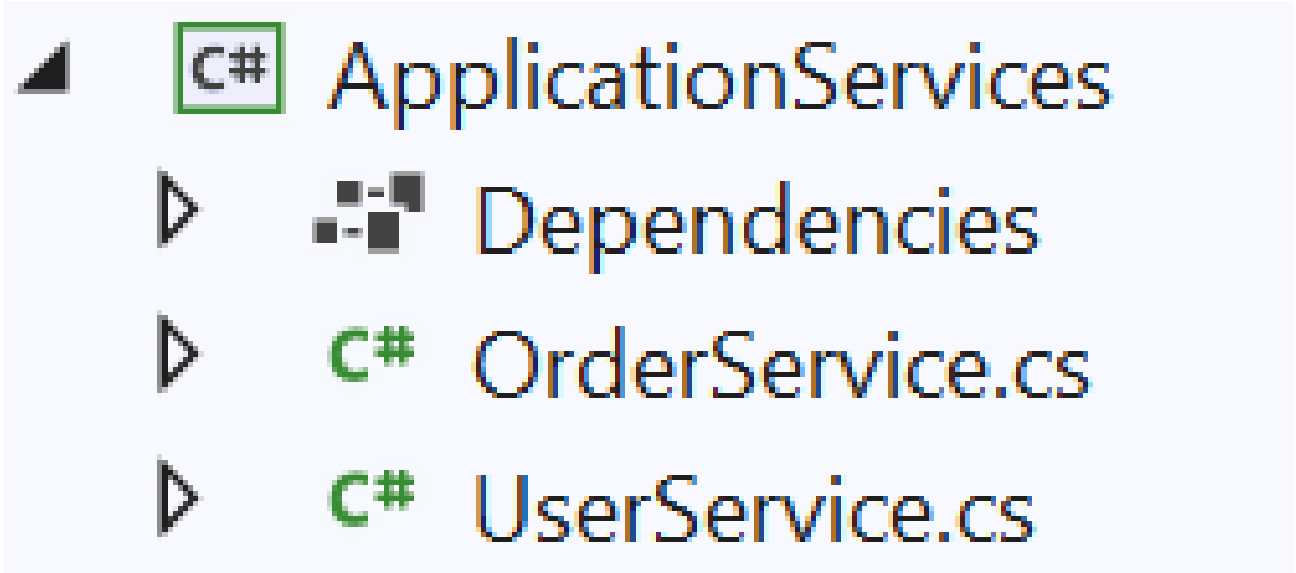
- Горизонтальный
  - Services
- Вертикальный
  - CQRS-хендлеры
  - Vertical Slices

Сервис для каждого агрегата

```
public class OrderService
{
    public Order CreateOrder()
    {
    }

    public List<Order> GetOrders()
    {
        return _dbContext.Orders;
    }
}
```

Компонент с сервисами

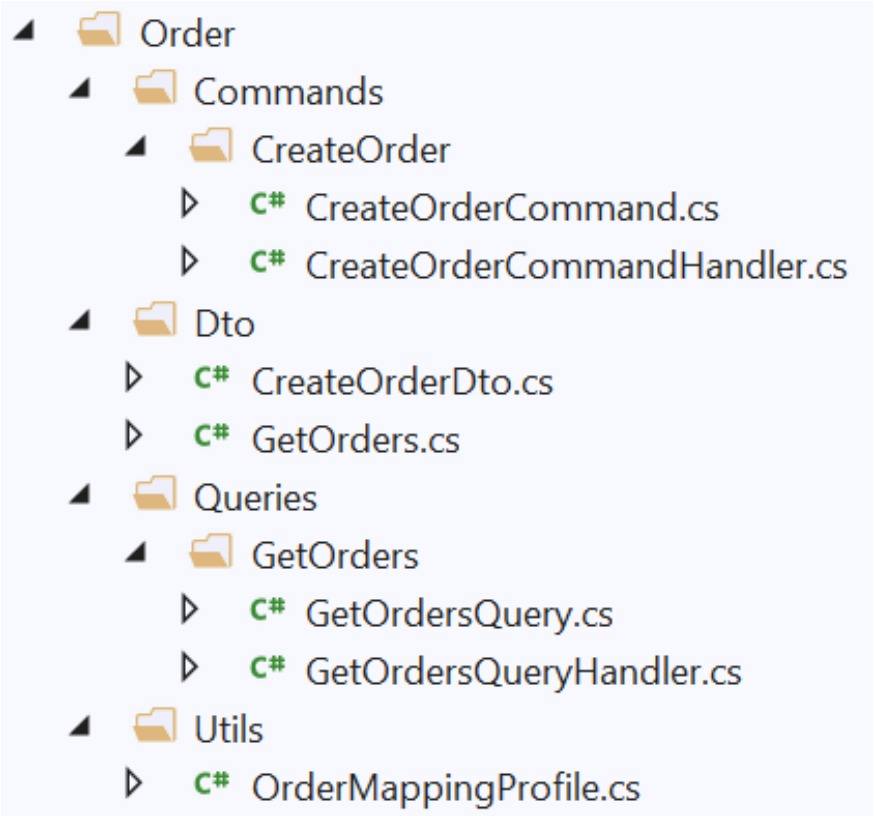


Хендлер для каждого юскейса

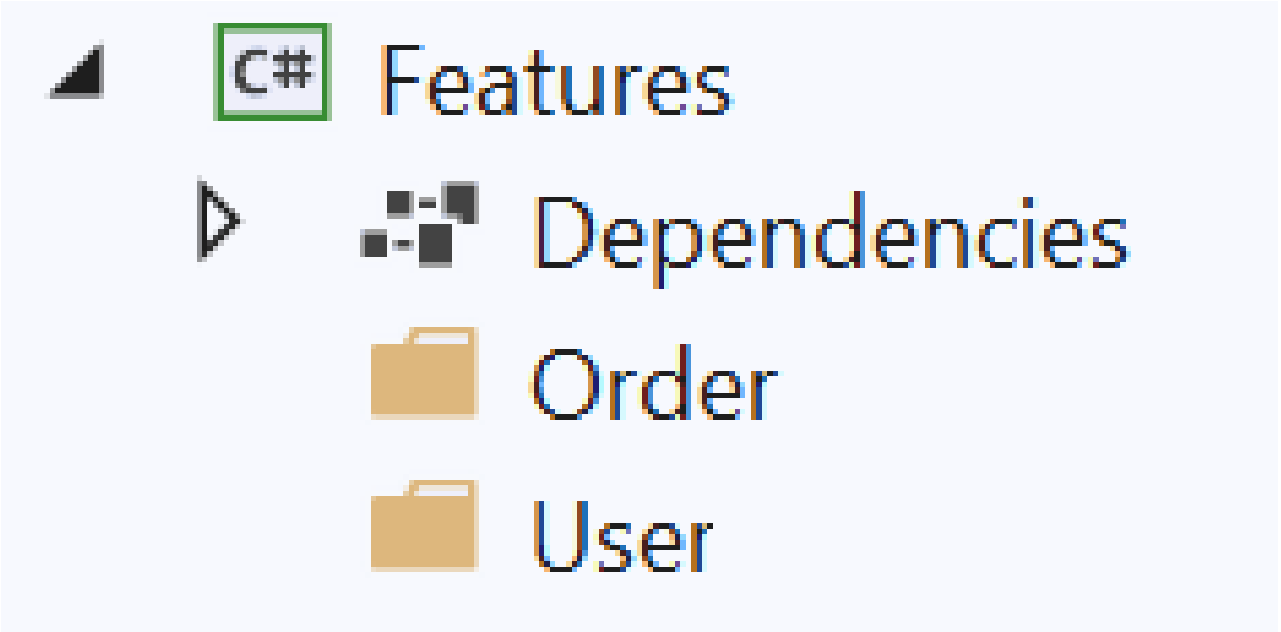
```
public class GetOrdersQuery
{
}

public class GetOrdersQueryHandler
{
    public List<Order> Handle(GetOrdersQuery request)
    {
        return _dbContext.Orders;
    }
}
```

Юскейсы для агрегата



Компонент с юскейсами



Некоторых не отпускает наследие сервисов ☺

```
public class OrderService :IGetOrdersRequestHandler,
                           IGetOrderRequestHandler
{
    public List<Order> Handle(GetOrdersRequest request)
    {
        return _dbContext.Orders;
    }
    public Order Handle(GetOrderRequest request)
    {
        return _dbContext.Orders.Find(request.Id);
    }
}
```

Достоинства хендлеров: 3 + 2 + 2 +2 = 9

- 3 очевидных
- 2 требуют определенных условий
- 2 для переиспользования юскейсов
- 2 для cross-cutting concerns

Очевидные

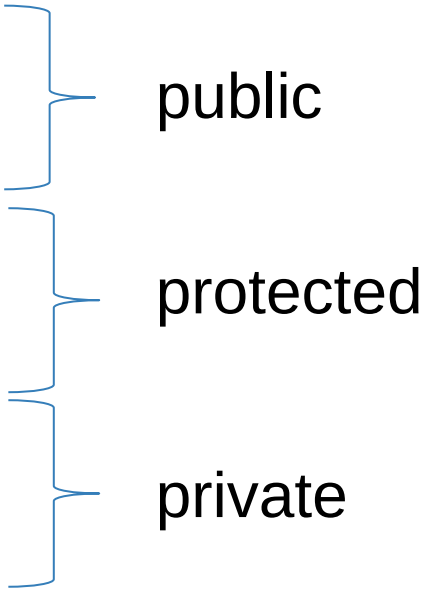
1. Код юскейса инкапсулирован в хендлере
2. В хендлере нет лишних зависимостей
3. В хендлерах нет лишних Generic-параметров

1. Код юскейса в одном классе

- Весь код юскейса в одном хендлере
- В хендлере нет кода других юскейсов

В сервисе юскейсы фрагментируются

```
public class OrderService {
    public void CreateOrder(CreateDto dto) {
    }
    public void UpdateOrder(UpdateDto dto) {
    }
    protected void ValidateOrder(Order order) {
    }
    protected void OrderBelongsToUser() {
    }
    private void OrderCreated() {
    }
    private void OrderUpdated() {
    }
}
```



Юскейсы реализованы в разных сервисах

- Клонирование жирного агрегата – в 7 сервисах
- Один сервис вызывал другие сервисы

Размер имеет значение

- Самый жирный хендлер – около 500 строк кода. Обычно в разы меньше
- Сервис – от сотен до тысяч строк кода
- Это кажется очевидным
- Но поддерживать маленькие хендлеры с четкой ответственностью намного проще

Single Responsibility для класса и системы

- Хендлер соответствует SRP, про сервис сложно это сказать
- Архитектура на базе хендлеров расширяется добавлением новых классов, а не переписыванием сервисов

2. В хендлере нет лишних зависимостей

- Только зависимости, нужные для этого юскейса
- Больше 5 зависимостей в хендлере не видел
- Для сервиса 10 зависимостей - не предел
- Проще писать тесты на хендлеры, меньше зависимостей мокать

Сервис, 11 зависимостей

```
public ScenarioService(
    IScenarioRepository scenarioRepository,
    IRigScheduleRepository rigScheduleRepository,
    IPopScheduleRepository popScheduleRepository,
    IExecutionContextAccessor executionContextAccessor,
    IConfigurationProvider mappingConfiguration,
    IWellRepository wellRepository,
    IProjectRepository projectRepository,
    IMeterStationRepository meterStationRepository,
    ITrancheRepository trancheRepository,
    IUnitRepository unitRepository,
    IUserService userService)
```

Хендлер после рефакторинга, 3 зависимости

```
public CloneScenarioCommandHandler(
    IDbContext dbContext,
    IExecutionContextAccessor accessor,
    IMapper mapper)
```

3. В хендлере нет лишних Generic-параметров

- Только generics, нужные для этого юскейса
- Больше 3 generics в хендлере не видел
- Для сервиса бывает 10 и более generics

Сервис, 10 generics

```
public abstract class ModelRunsService<
    TModelRunDto,
    TRunInputDto,
    TRunInfoDto,
    TCreateRunDto,
    TRunEntity,
    TRunInputEntity,
    TRunRequestMessage,
    TRunConfiguration,
    TRunnerInputDto,
    TKillRunRequestMessage>
```

Хендлер, 3 generics

```
public abstract class UpdateRunCommandHandler<
    TCommand,
    TMessage,
    TRunEntity>
```

Проявляются при условиях

4. Нет операций, которые не определены для сущности
5. Проще создать дополнительный уровень абстракции

4. Нет лишних операций

```
public abstract class BaseService<T>
{
    public virtual void Add(T entity) { }

    public virtual void Edit(T entity) { }
}
```

Новая сущность – нельзя создавать, можно только клонировать и редактировать

Сервис для новой сущности

```
public class EntityService : BaseService<Entity>
{
    public void Clone(Entity entity) { }

    public override void Edit(Entity entity) { }

    public override void Add(Entity entity)
        { throw new NotSupportedException(); }

    //Add в базовом классе
    //Но эта операция не имеет смысла для Entity. Путаница
}
```

В случае хендлеров

Для каждой операции

- Базовый и специфичный класс команды
- Базовый и специфичный класс хендлера

- Нет копипаста
- Нет операций, не определенных для сущностей
- Дизайн системы лучше

Команды

```
public abstract class EditCommand
{
}

public class EntityEditCommand : EditCommand
{
}
```

Базовый хендлер

```
public abstract class EditCommandHandler<TCommand>
    where TCommand : EditCommand
{
    public virtual void Handle(TCommand command)
    {
        // Common code
    }
}
```

Специфичный хендлер

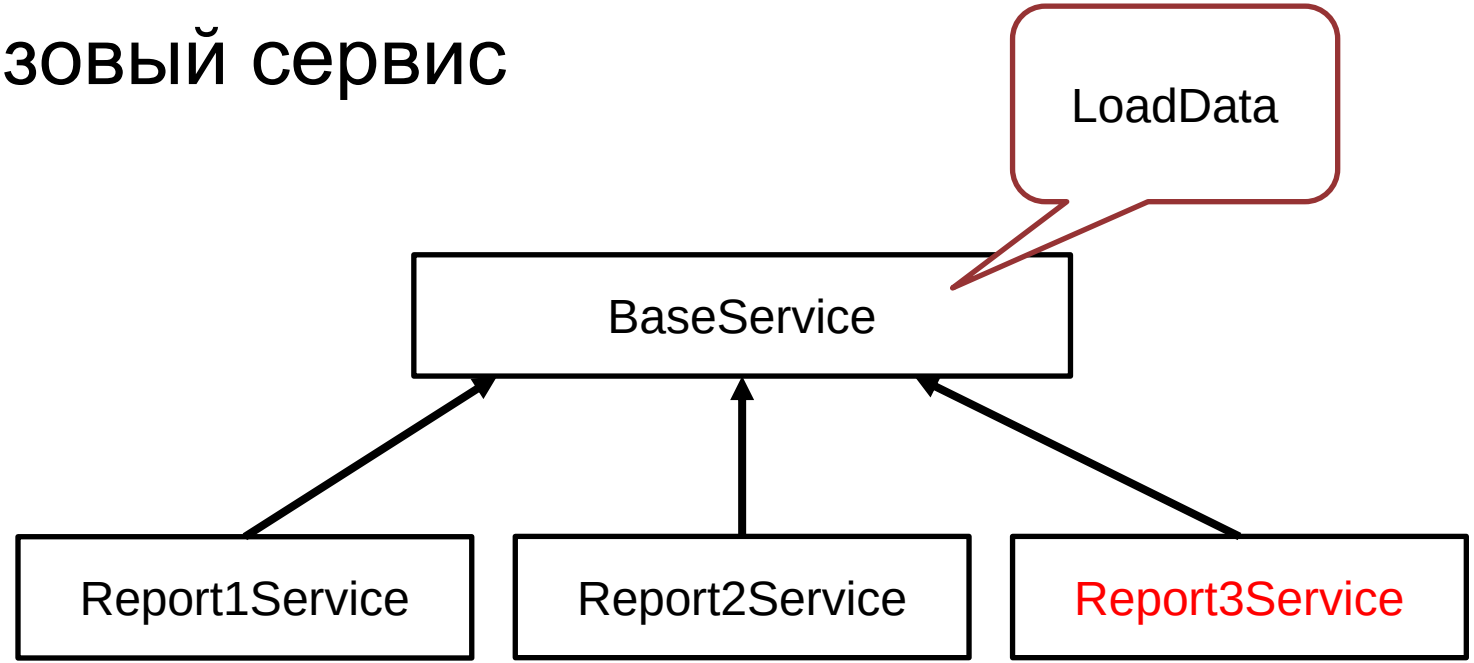
```
public class EntityEditCommandHandler :
    EditCommandHandler<EntityEditCommand>
{
    public override void Handle(EntityEditCommand command)
    {
        // Common code
        base.Handle(command);

        // Specific code
    }
}
```

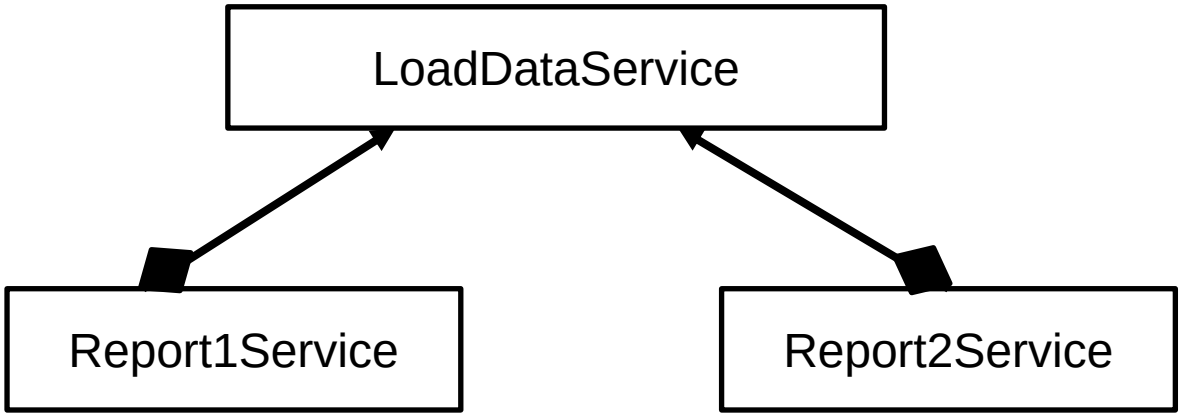
5. Новый уровень абстракции

- Несколько отчетов, построить каждый отчет – отдельный юскейс
- Общий выбор данных для всех отчетов
- Юскейсы дорабатывают выбранные данные
- Куда положить выбор данных?
- В общем случае – куда девать общую логику для разных сервисов/хендлеров?

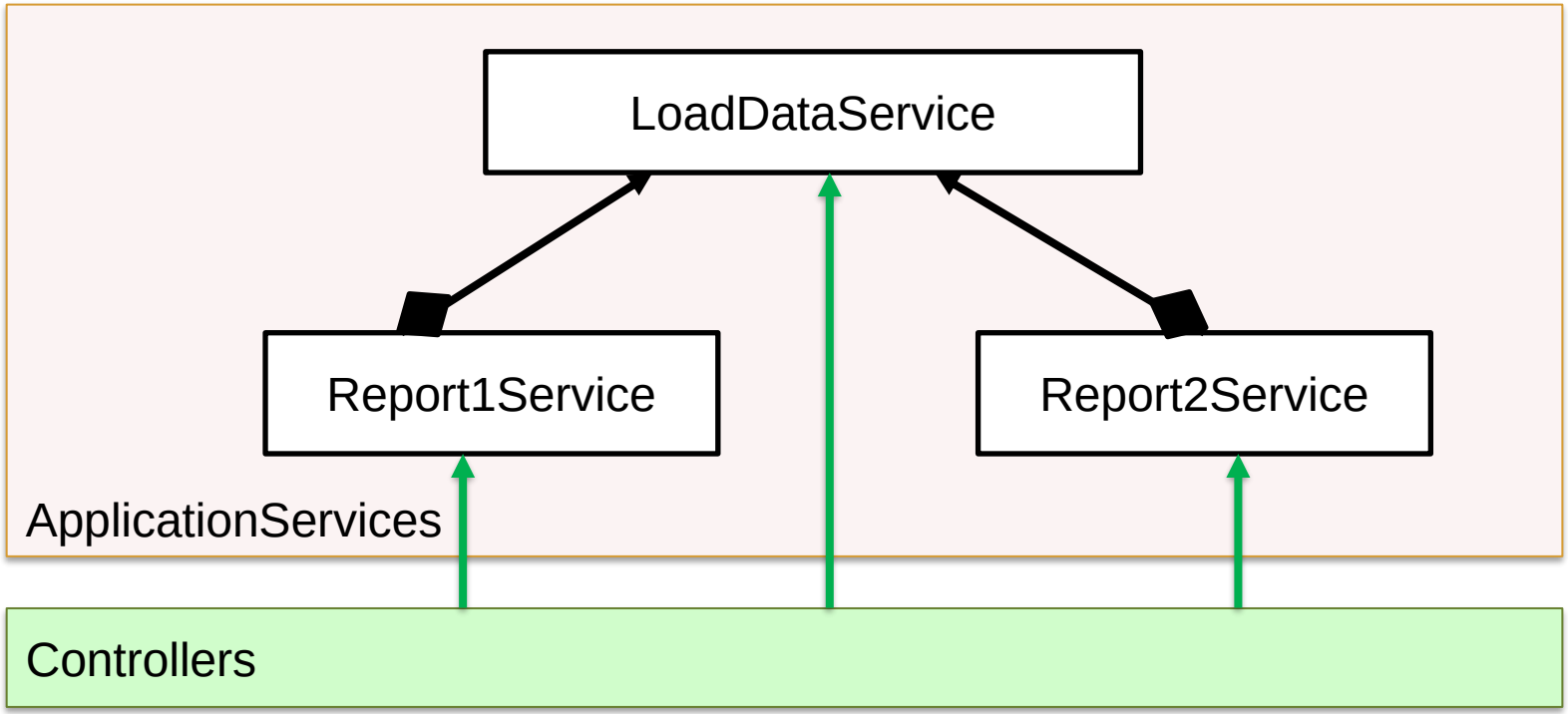
Базовый сервис



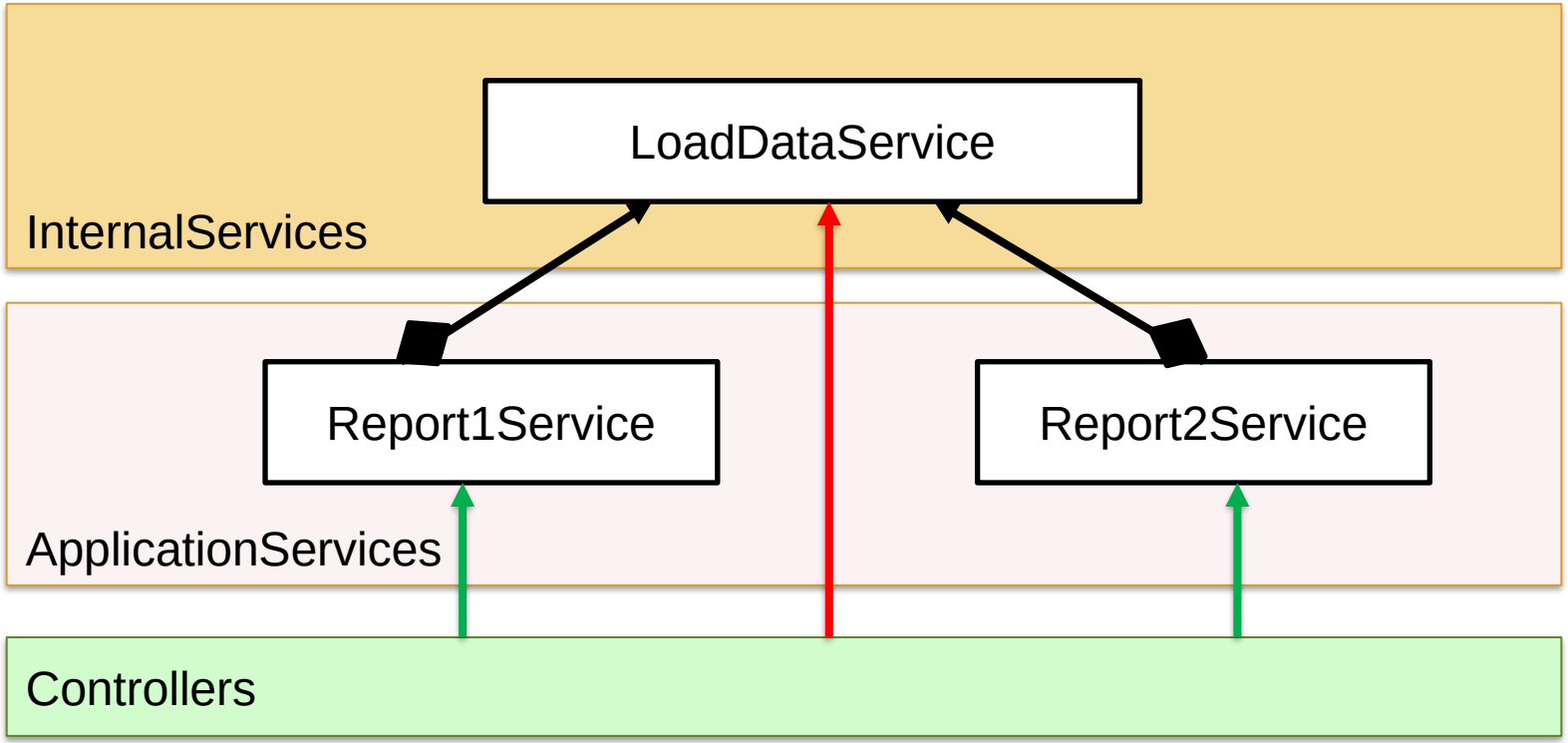
Отдельный сервис



Контроллеры могут вызывать новый сервис?



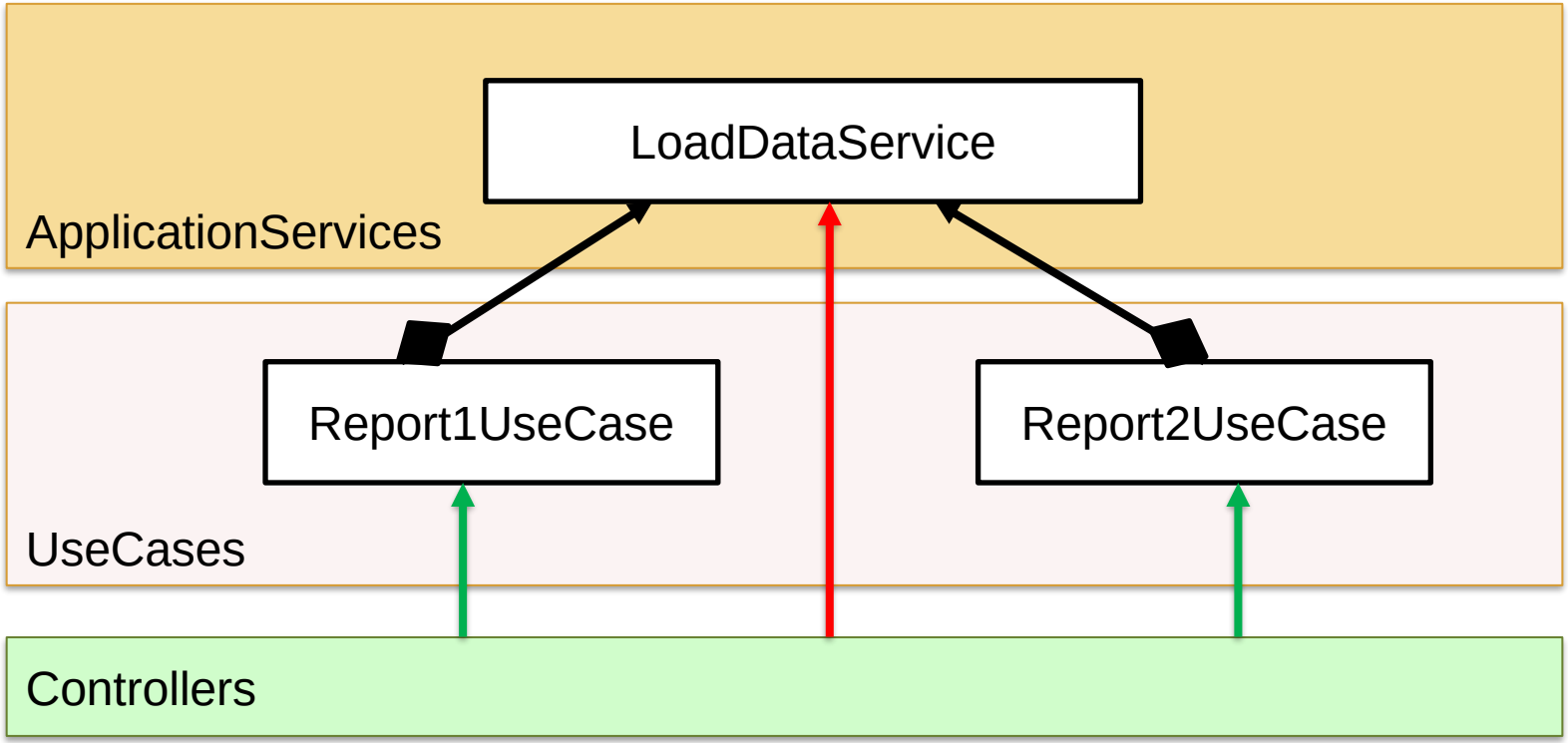
Новый слой


LoadDataService
InternalServices
Report1Service
Report2Service
ApplicationServices
Controllers

Как назвать новый слой?

- Infrastructure - нет
  - Это не абстракция для базы, ServiceBus, BlobStorage итд
- Domain – нет
  - Оперирует не Entities, а агрегированными данными
  - Использует инфраструктуру
- Application – нет
  - Уже есть ☺
  - Этот слой не может использоваться контроллерами
- Фактически Application делится на две части

В случае хендлеров нет дилеммы



Может ли она появиться в будущем?

- В теории – да
  - Количество слоев абстракции не ограничено ☺
- На практике – нет
  - UseCases, ApplicationServices, DomainServices всегда было достаточно

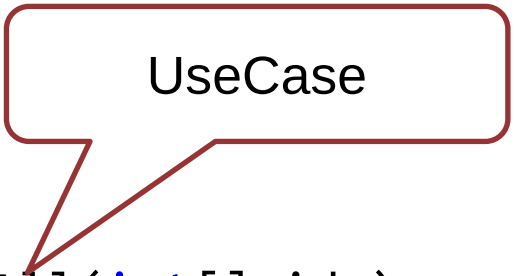
Переиспользование юскейсов

6. Явный вызов юскейса из юскейса
7. Нет проблемы циклов между сервисами

6. Явный вызов юскейса из юскейса

- В CQRS среде есть холивар: можно вызывать юскейс из юскейса или нет
- В слоистой среде его нет. Почему?

Типичный сервис

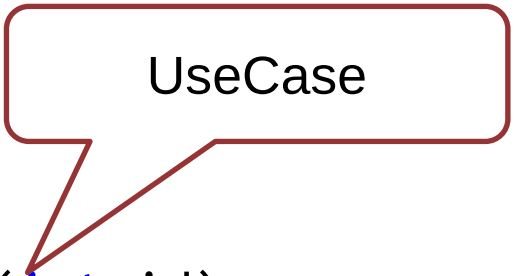



```
public class Service
{
    public void DeleteAll(int[] ids)
    {
        ids.ForEach(id => Delete(id));
    }
}
```

Контроллер вызывает сервис

```
public class Controller
{
    [HttpDelete]
    public void DeleteAll(int[] ids)
    {
        service.DeleteAll(ids);
    }
}
```

По методу не видна его область видимости



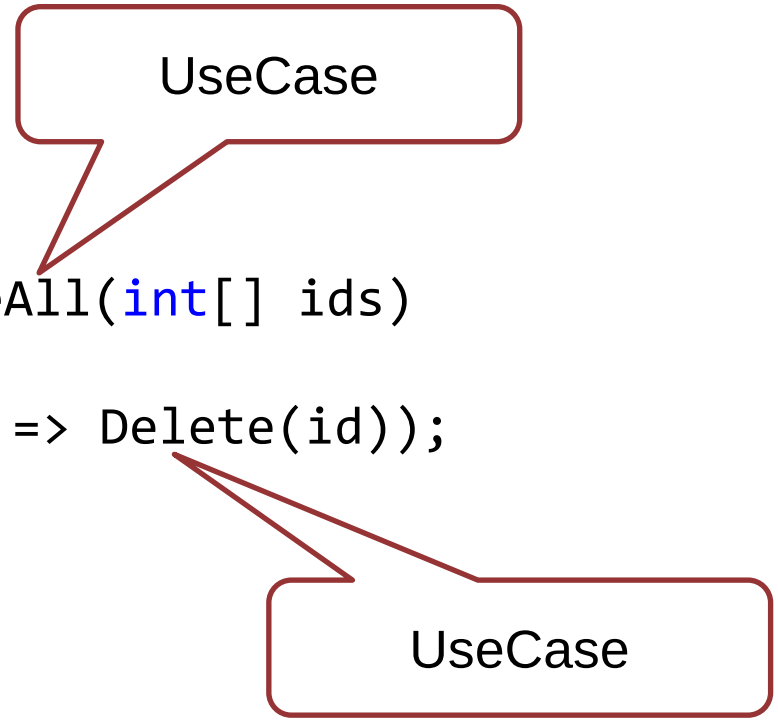

```
public class Service
{
    public void Delete(int id)
    {
    }
}
```

Контроллер вызывает сервис

```
public class Controller
{
    [HttpDelete]
    public void DeleteAll(int[] ids)
    {
        service.DeleteAll(ids);
    }
    [HttpDelete]
    public void Delete(int id)
    {
        service.Delete(id);
    }
}
```

Вызов юскейса из юскейса

```
public class Service
{
    public void DeleteAll(int[] ids)
    {
        ids.ForEach(id => Delete(id));
    }
}
```



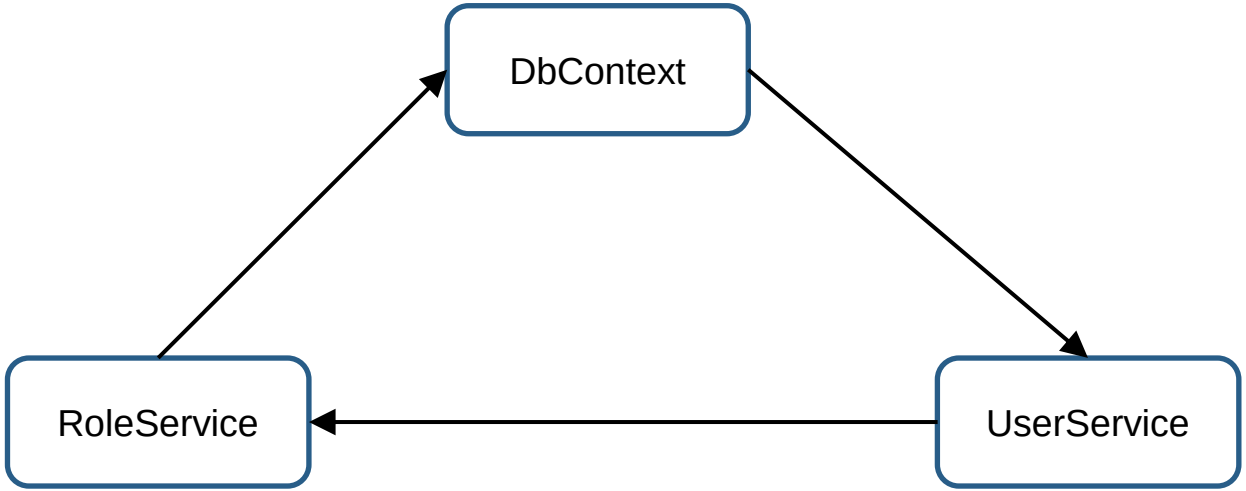
В случае с хендлерами не перепутаешь

```
public class DeleteAllCommandHandler
{
    public void Handle(DeleteAllCommand command)
    {
        command.Ids.ForEach(
            id => mediator.Send(new DeleteCommand(id)));
    }
}
```

В случае сервисов поможет именование

```
public class Service
{
    public void DeleteAll_UseCase(int[] ids)
    {
        ids.ForEach(id => Delete_UseCase(id));
    }
}
```

7. Нет проблемы циклов между сервисами



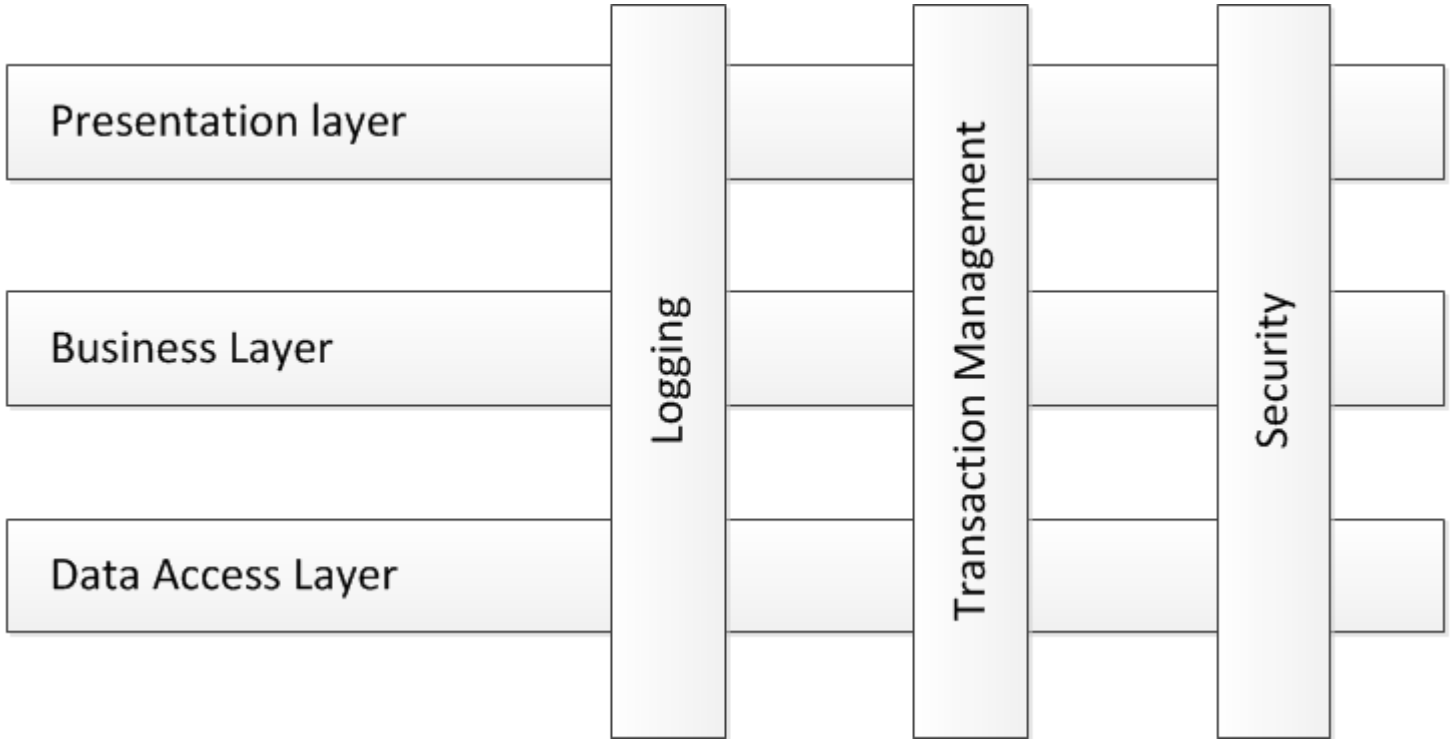
Хендлер не зависит от другого хендлера

```
public class EntityEditCommandHandler
{
    public void Handle(EntityEditCommand command)
    {
        mediator.Send(new SomeOtherCommand());
    }
}
```

Cross-cutting concerns

8. Cross-cutting concerns без магии
9. Cross-cutting concerns на уровне Application

8. Cross-cutting concerns без магии



В случае сервисов

- Dynamic Proxy
  - Castle
  - DI (Autofac, CastleWindsor, Unity)

- MSIL хаки
  - Fody
  - PostSharp

Да, но

- Fody и PostSharp платные
- Fody MethodDecorator – нет DI
- Fody MethodDecorator и Castle Interceptor синхронные
  - AsyncInterceptor давно в бете

Interceptor

```
public class TransactionScoper : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        using (var tr = new TransactionScope())
        {
            invocation.Proceed();
            tr.Complete();
        }
    }
}
```

Прокси

```
var generator = new ProxyGenerator();
var foo = new Foo();
var proxy = generator.
    CreateInterfaceProxyWithTarget(foo, TransactionScoper);
```

Fody MethodDecorator

```
public class InterceptorAttribute : Attribute,
    IMethodDecorator
{
     public void OnEntry() { }

     public void OnExit() { }

     public void OnException(Exception exception) { }
}
```

Как его использовать

```
public class Sample
{
    [Interceptor]
    public void Method()
    {
        Debug.WriteLine("Your Code");
    }
}
```

В случае хендлеров

- Унификация обработки Command и Query

- Настройка пайплайна обработки запроса
  - PreProcessor
  - Behavior
  - PostProcessor

Базовый Request и Handler

```
public interface IRequest<TResponse>
{
}

public interface IRequestHandler<TRequest, TResponse>
   where TRequest : IRequest<TResponse>
{
   Task<TResponse> Handle(TRequest request);
}
```

Behavior

```
public class LoggingBehavior<TRequest, TResponse> :
    IPipelineBehavior<TRequest, TResponse>
{
    public async Task<TResponse> Handle(TRequest request,
        RequestHandlerDelegate<TResponse> next)
    {
        // before
        var response = await next();
        // after

        return response;
    }
}
```

Чем удобно

- Dependency Injection
- Асинхронность
- Нет проблем с отладкой
- Проще для понимания

8. Cross-cutting concerns на уровне логики

- ASP.NET тоже умеет Cross-cutting concerns
  - Middleware
  - Filters

Middleware

```
public class ExceptionHandlerMiddleware {
    public async Task Invoke(HttpContext httpContext) {
        try {
            await next(httpContext);
        }
        catch (EntityNotFoundException ex) {
            // 404 NotFound HTTP Code
        }
    }
}
```

Filter

```
public class SimpleResourceFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
    }
}
```

Использование фильтра

```
[SimpleResourceFilter]
public class HomeController : Controller
{
    [SimpleResourceFilter]
    public IActionResult Index()
    {
        return View();
    }
}
```

Интеграционные тесты (TestServer)

```
public class BasicTests :
    IClassFixture<WebApplicationFactory<Startup>>
{
    [Theory, InlineData("/Index")]
    public async Task Get_EndpointsReturnSuccess(string url) {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
    }
}
```

В случае хендлеров

- Магия Generic для добавления Behavior
- Любой DI кроме стандартного ASP.NET Core ☺

Behavior

```
public class LoggingBehavior<TRequest, TResponse> :
    IPipelineBehavior<TRequest, TResponse>
    where TRequest : ILoggingRequest
{
    public async Task<TResponse> Handle(TRequest request,
        RequestHandlerDelegate<TResponse> next)
    {
        // before
        var response = await next();
        // after

        return response;
    }
}
```

DTO

LoggingRequest

```
public class EditEntityCommand : ILoggingRequest
{
    public int Id { get; set; }

    public EditDto Dto { get; set; }
}
```

Нестандартный DI контейнер

Container / Feature	ASP.NET Core DI	Autofac	DryIoc	DryIocZero	LightInject	Ninject	SimpleInjector
Request Handler	✅	✅	✅	✅	✅	✅	✅
Void Handler	✅	✅	✅	✅	✅	✅	✅
Pipeline Behavior	✅	✅	✅	✅	✅	✅	✅
Pre-Processor	✅	✅	✅	✅	✅	✅	✅
Post-Processor	✅	✅	✅	✅	✅	✅	✅
Constrained Post-Processor	❌	✅	✅	✅	✅	❌	✅
Ordered Behaviors	❌	✅	✅	✅	✅	✅	✅

https://github.com/jbogard/MediatR/wiki/Container-Feature-Support

Тесты через MediatR

```
[Fact]
public async Task Test() {
    // Arrange
    var mediatr = CreateMediatr(); // using DI Container

    // Act
    var result = await mediatr.Send(new EditEntityCommand{Id = 1});

    // Assert
    Assert.Equal(1, result.Id);
}
```

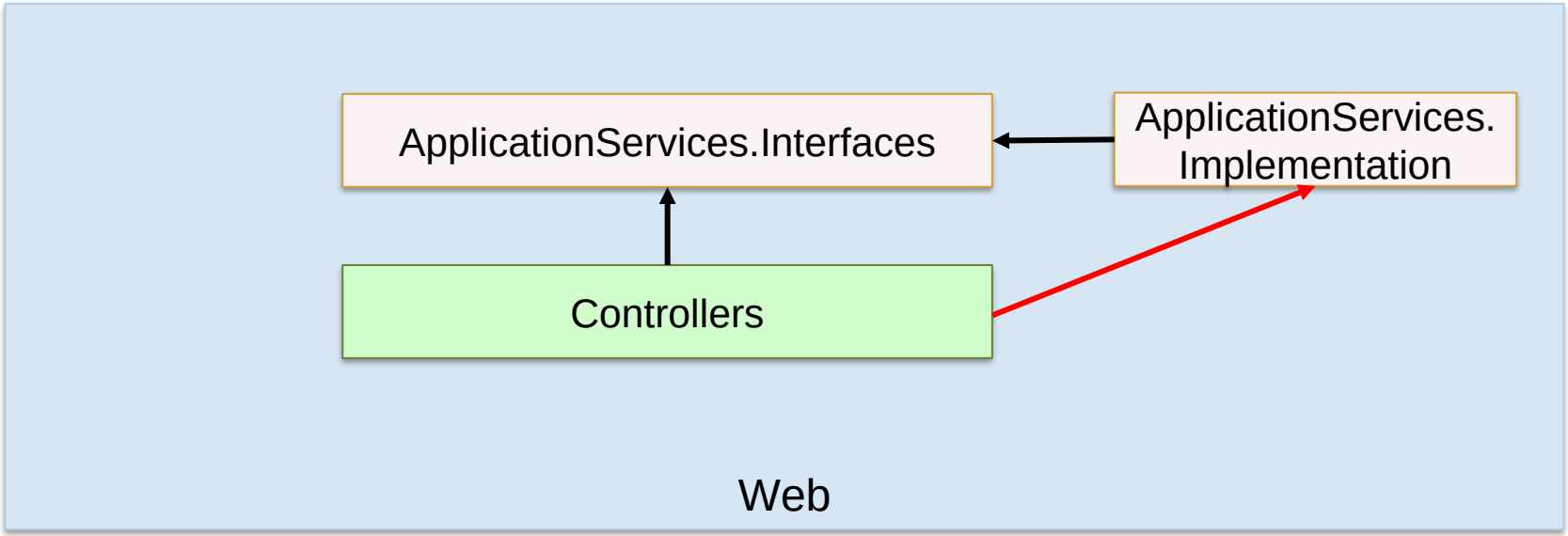
Чем удобнее

- Cross-cutting concerns не зависят от веб-фреймворка
  - DTO вместо HttpContext
- Проще тестирование
  - Не нужен TestServer

Недостаток

- Command/Query+DTO и Handlers – один компонент
- Можно вызвать команду прямо из контролера

Контроллеры – отдельный компонент от хоста

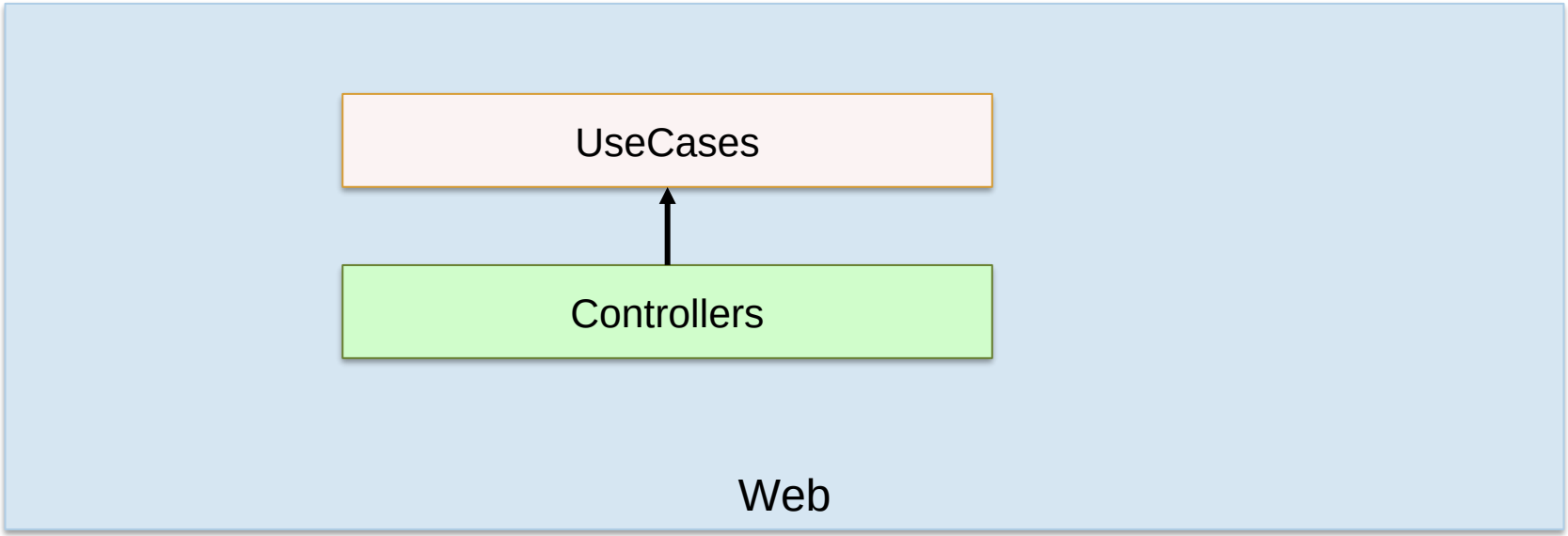


Нельзя вызвать из сервис из контроллера

```
public class OrderController : Controller
{
    public IActionResult GetOrders()
    {
        return new OrderService().GetOrders(); //CE

        return orderService.GetOrders(); //IOrderService
    }
}
```

Request и Handler в одном компоненте



Можно вызвать хендлер из контроллера

```
public class OrderController : Controller
{
    public IActionResult GetOrders()
    {
        return new GetOrdersHandler().Handle();
    }
}
```

Зачем request и хендлер в одном компоненте

- Проще искать хендлер по request
- Go to implementation не работает ☺
- Положить в одну папку request и хендлер – профит!
  - Находим request, а хендлер рядом)

На практике недостаток не проявляется

- Проявляется при выделении Controlers в отдельный компонент
  - Иначе сервис из контроллера тоже можно создать
- Контроллеры отдельно (почти) никто не выделяет
- Теряем возможность настройки пайплайна
- Хендлеры можно сделать internal
- На практике ни разу не видел :)

Совет 1. Переиспользовать ли DTO?

```
public class CreateOrderDto
{
    // одинаковые данные
}

public class UpdateOrderDto
{
    // одинаковые данные
}
```

Совет 1. Переиспользовать ли DTO?

```
public class CreateOrderDto : OrderDto
{

}

public class UpdateOrderDto : OrderDto
{

}
```

Совет 1. Переиспользовать ли DTO?

```
public class OrderDto { }

public class OrderController : Controller
{
    [HttpPost]
    public IActionResult CreateOrder(OrderDto dto) { }

    [HttpPut("id")]
    public IActionResult UpdateOrder(int id, OrderDto dto) {
   }
}
```

Совет 1. Переиспользовать ли DTO?

- Да - в одном BFF
  - На каждый юскейс свой DTO
  - Базовый класс – это норм

Совет 1. Переиспользовать ли DTO?

- Нет - между агрегатами и BFF
- Редактирование данных пользователя
  - Из личного кабинета самим пользователем
  - Админом из админки
- Да, DTO надо будет скопипастить ☺
  - Прививка от перфекционизма
  - Копипаст лучше черезмерной связности
  - DTO – не место для объектно-ориентированного дизайна

Совет 2. MediatR или свой велик?

- Однозначно MediatR!
- Видел несколько своих великов
  - Про некоторые говорили, что они круче MediatR
- Их все в итоге заменили на MediatR

Совет 3. Request и DTO – отдельные классы?

```
public class EditEntityCommand : IRequest
{
   public string Name { get; set; }
}

public Task<IActionResult> CreateOrder([FromBody]
   EditEntityCommand command)
{
    return OK(mediator.Send(command));
}
```

Совет 3. Request и DTO – отдельные классы?

```
public class EditEntityDto
{
    public string Name { get; set; }
}

public class EditEntityCommand
{
    public EditEntityDto Dto { get; set; }
}
```

Совет 3. Request и DTO – отдельные классы?

```
public Task<IActionResult> CreateOrder([FromBody]
   EditEntityDto dto)
{
    return OK(mediator.Send(
      new EditEntityCommand { Dto = dto }));
}
```

Совет 3. Request и DTO – отдельные классы?

- Request и DTO отдельно!
- Разные иерархии наследования
- Request может содержать больше данных, чем DTO
  - Id для PUT и PATCH
  - Данные, передаваемые между этапами пайплайна

От сервиса к хендлерам

```
public class OrderService
{
    public Order CreateOrder()
    {
    }

    public List<Order> GetOrders
    {
        return _dbContext.Orders;
    }
}
```

CreateOrderCommand
CreateOrderCommandHandler

GetOrdersQuery
GetOrdersQueryHandler

Метод сервиса

```
public void CreateOrder(CreateOrderDto dto)
{
    var order = _mapper.Map<Order>(dto);
    _dbContext.Orders.Add(order);
    _dbContext.SaveChanges();
}
```

Хендлер. Параметры – свойства команды

```
public void Handle(CreateOrderCommand command)
{
    var order = _mapper.Map<Order>(command.Dto);
    _dbContext.Orders.Add(order);
    _dbContext.SaveChanges();
}
```

Как перейти от сервисов к хендлерам

- Потренироваться на одном аргегате/веб-джобе/микросервисе
- Рутинная работа. Полчаса в день всей командой. Нет отрыва от бизнес-задач
- Быстро – это медленно, но постоянно. Но нужен тот, кто не позволит заби(ы)ть
- В любой момент можно приостановиться, при этом не будет незавершенной работы
- Решарпер хорошо генерит инфраструктурный код
  - Конструкторы с параметрами

Собираем все вместе

- Очевидные достоинства
  1. Инкапсуляция юскейсов в хендлерах
  2. В хендлерах нет лишних зависимостей
  3. В хендлерах нет лишних generics
- Видные при определенных условиях достоинства
  4. Нет операций, не определенных для сущности
  5. Проще выделять дополнительный слой абстракции
- Переиспользование юскейсов
  6. Явный вызов юскейсов из юскейсов
  7. Нет проблемы циклов между сервисами
- Cross-cutting concerns
  8. Cross-cutting concerns без магии
  9. Cross-cutting concerns на уровне логики, а не фреймворка

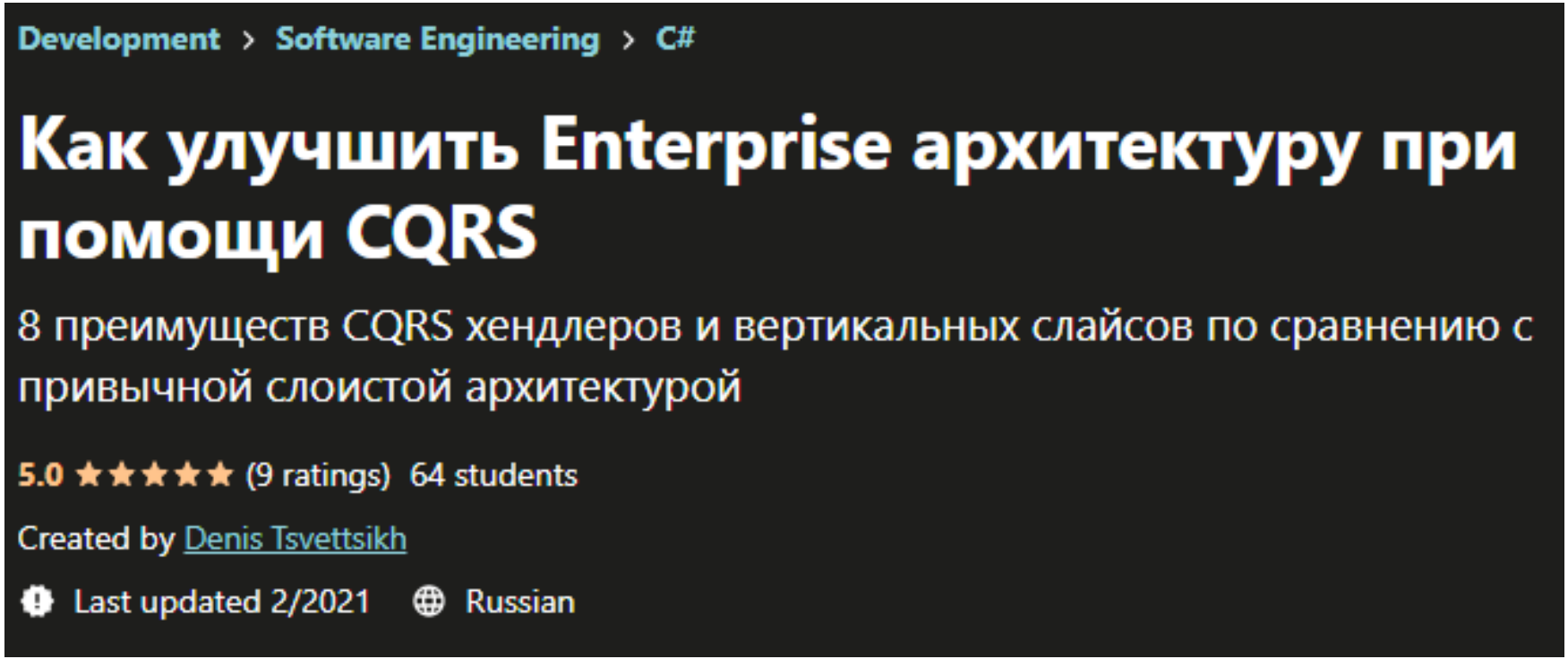
Продолжаем собирать

- Советы
  - Переиспользование DTO
    - Каждому юскейсу свой DTO, возможно с базовым классом
    - Между аргератами и BFF – копипаст
  - MediatR вместо своих великов
  - Не использовать Request как DTO
- Как перейти от сервисов к хендлерам
  - Медленно, но постоянно и всей командой. Не забивая)
  - Решарпер поможет

Материалы по теме

- Александр Бындю. CQRS на практике
  https://blog.byndyu.ru/2014/07/command-and-query-responsibility.html
- Jimmy Bogard. Vertical Slice Architecture
  https://youtu.be/SUiWfhAhgQw
- Максим Аршинов. Быстрорастворимое проектирование
  https://habr.com/ru/company/jugru/blog/447308/

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



Призыв

Попробуйте. Догадки часто не соответствуют реальности

Первое впечатление может быть неверным

- Нет переиспользованию
- Больше кода – больше багов

Спасибо за внимание

Денис Цветцих
Lead .NET Developer
Invent

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