

Не любишь DomainEvents из DDD?
Ты просто не умеешь их готовить!



**Денис
Цветцх**

DevBrothers

Domain Events

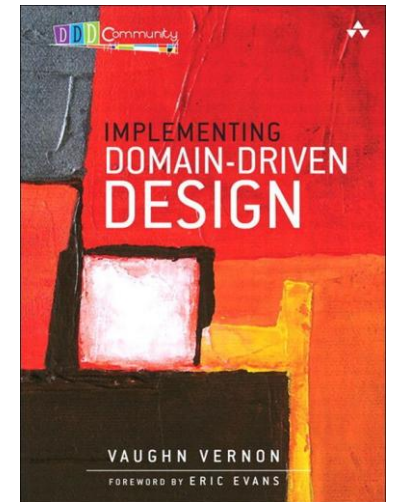
- Aggregate, Entity, ValueObject, **DomainEvent**
- Редко используются на практике
- Как делать – мало информации
- Зачем – еще меньше

О чем поговорим

- Что такое DomainEvents в разных книгах по DDD
- Мессаджи между Bounded Context
- Доменные ивенты внутри Bounded Context
- ЧАВО о DomainEvents
- ApplicationEvents и стратегии их обработки

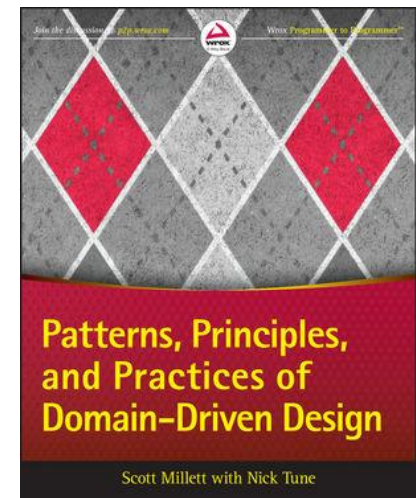
Что такое доменные ивенты (RedBook)

- Факт, имеющий значение для экспертов предметной области
 - Порождается командой
- События передаются во внешние службы
- Позволяют избежать двухфазной системы фиксации распределенных транзакций (как?)



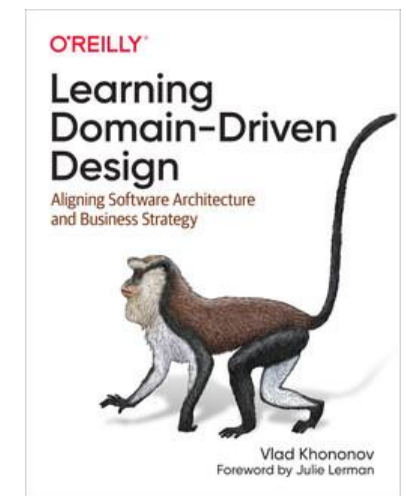
Миллет, DDD C#

- Однопоточное выполнение
- Внутри одного ограниченного контекста
- Могут быть асинхронными для интеграции контекстов
 - Противоречит второму пункту



Learning Domain-Driven Design, Vlad Khononov

```
01 public class Campaign
02 {
03     ...
04     List<DomainEvent> _events;
05     IMessageBus _messageBus;
06     ...
07
08     public void Deactivate(string reason)
09     {
10         for (l in _locations.Values())
11         {
12             l.Deactivate();
13         }
14
15         IsActive = false;
16
17         var newEvent = new CampaignDeactivated(_id, reason);
18         _events.Append(newEvent);
19         _messageBus.Publish(newEvent);
20     }
21 }
```

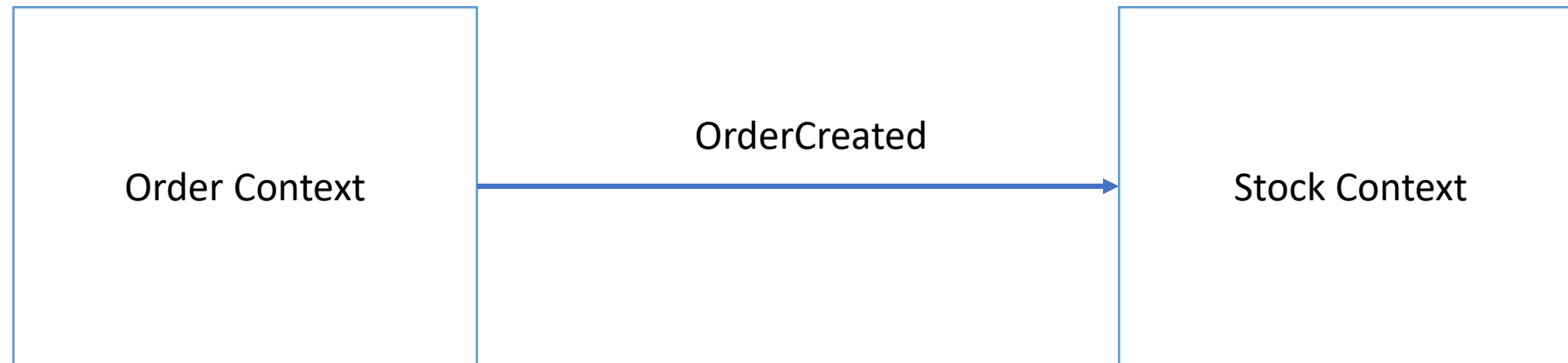


.NET Microservices architecture

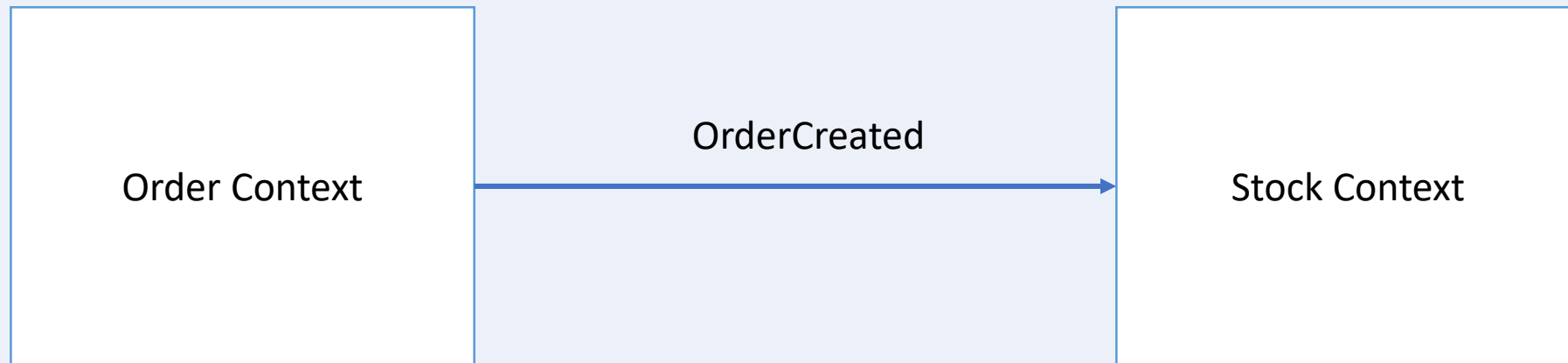
- Для взаимодействия агрегатов в одном ограниченном контексте
- Изменения для команд и ивентов сохраняются в одной транзакции
 - Transactional consistency, а не eventual consistency
- Похожи на триггеры в базе
- Могут быть синхронными и асинхронными
 - Это противоречит сохранению изменений в одной транзакции



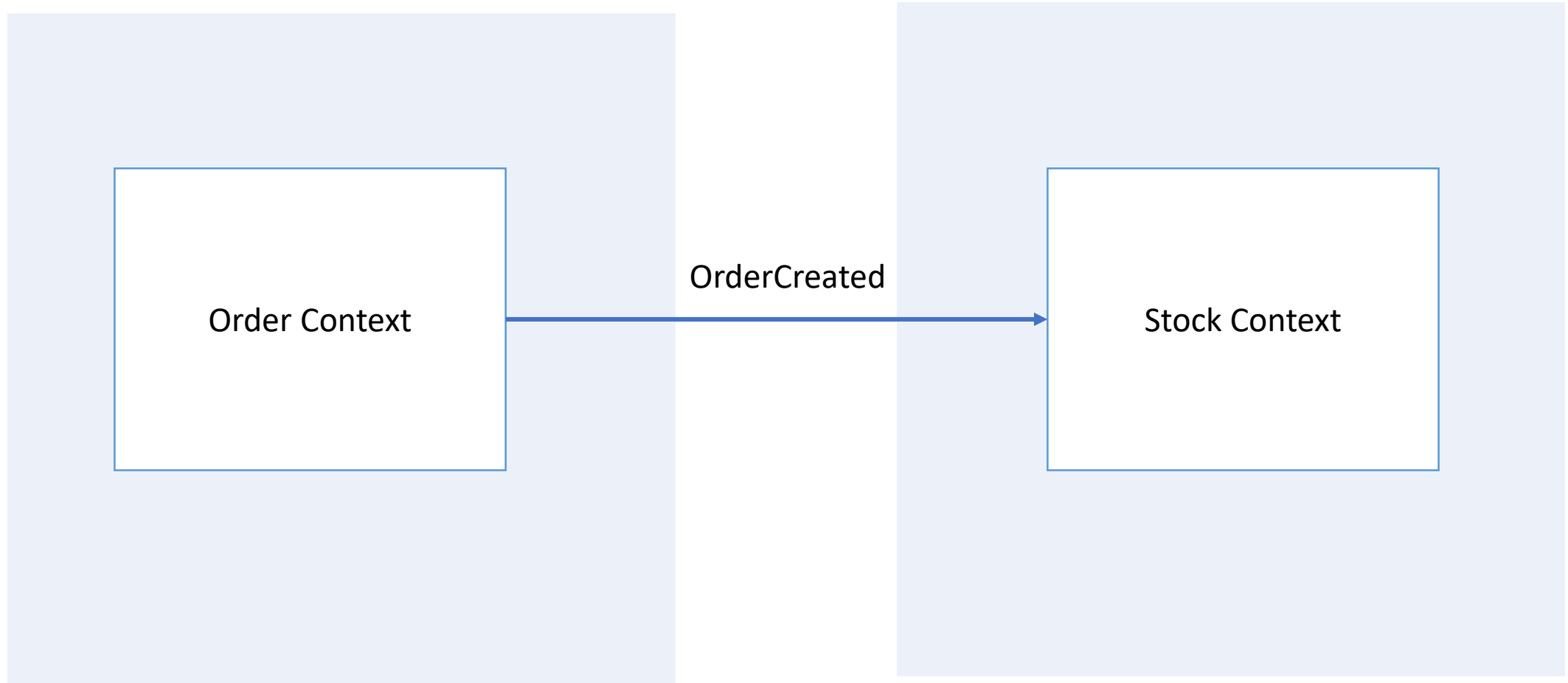
Два Bounded Context



Один процесс



Разные процессы



Бизнес-модель и контракт

- Entity -> DTO
- Domain Event -> Message

Два Bounded Context



Транспорт: InMemory или MessageBroker

- DotNetCore.CAP <https://github.com/dotnetcore/CAP>
- MassTransit <https://github.com/MassTransit/MassTransit>

DomainEvents vs IntegrationMessages

- DomainEvents

- Деталь реализации
- В одном Bounded Context (и в одном процессе)
- Уведомляют другие агрегаты (тактический уровень)
- Общаются синхронно (Mediator)
- Результаты в одной транзакции (transactional consistency)

- IntegrationMessages

- Часть контракта
- Между Bounded Contexts (процесс один или разные)
- Уведомляют другие BoundedContext (стратегический уровень)
- Общаются асинхронно (MessageBroker)
- Результаты в разных транзакциях (eventual consistency)

Товар

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

    public bool IsDeleted { get; set; }
}
```

Категория

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;

    public List<ProductCategory> Products { get; set; } = null!;
}
```

Многие ко многим

```
public class ProductCategory
{
    public int ProductId { get; set; }
    public int CategoryId { get; set; }

    public Product Product { get; set; } = null!;
    public Category Category { get; set; } = null!;
}
```

Бизнес-правила

1. Товары удаляются «мягко»
2. При удалении товара он убирается из всех категорий
3. Категории без товаров удаляются

Удаление товара

BEGIN TRANSACTION

UPDATE Products SET IsDeleted=1 WHERE Id=@ProductId

DELETE FROM ProductCategories WHERE ProductId=@ProductId

DELETE FROM c FROM Categories c
WHERE NOT EXISTS (SELECT 1 FROM ProductCategories AS pc WHERE c.Id = pc.CategoryId)

COMMIT TRANSACTION

Удаление товара EF Core

```
public async Task Handle(DeleteProductCommand request, CancellationToken ct)
{
    await using var transaction = await _dbContext.BeginTransactionAsync(ct);

    await _dbContext.Products
        .Where(x => x.Id == request.ProductId)
        .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsDeleted, true), ct);

    await _dbContext.ProductCategories
        .Where(x => x.ProductId == request.ProductId)
        .ExecuteDeleteAsync(ct);

    await _dbContext.Categories
        .Where(x => !x.Products.Any())
        .ExecuteDeleteAsync(ct);

    await transaction.CommitAsync(ct);
}
```

Убрать товар из категории

BEGIN TRANSACTION

```
DELETE FROM ProductCategories WHERE ProductId=@ProductId AND CategoryId=@CategoryId
```

```
DELETE FROM c FROM Categories c  
WHERE c.Id=@CategoryId AND  
      NOT EXISTS (SELECT 1 FROM ProductCategories AS pc WHERE c.Id = pc.CategoryId)
```

COMMIT TRANSACTION

Убрать товар из категории EF Core

```
public async Task Handle(RemoveProductFromCategoryCommand request, CancellationToken ct)
{
    await using var transaction = await _dbContext.BeginTransactionAsync(ct);

    await _dbContext.ProductCategories
        .Where(x => x.ProductId == request.ProductId && x.CategoryId == request.CategoryId)
        .ExecuteDeleteAsync(ct);

    await _dbContext.Categories
        .Where(x => x.Id == request.CategoryId && !x.Products.Any())
        .ExecuteDeleteAsync(ct);

    await transaction.CommitAsync(ct);
}
```

Особенности

- Максимальная производительность
- SQL на C# (низкая читаемость)
- Нет переиспользования кода

Удаление товара -> команда для категории

```
await using var transaction = await _dbContext.BeginTransactionAsync(ct);
```

```
await _dbContext.Products  
    .Where(x => x.Id == request.ProductId)  
    .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsDeleted, true));
```

```
var categoryIds = await _dbContext.ProductCategories  
    .Where(x => x.ProductId == request.ProductId)  
    .Select(x => x.CategoryId)  
    .Distinct()  
    .ToListAsync(cancellationToken);  
  
foreach (var categoryId in categoryIds)  
{  
    await _sender.Send(new RemoveProductFromCategoryCommand(request.ProductId, categoryId));  
}
```

```
await transaction.CommitAsync(ct);
```

Вложенные транзакции

```
await using var transaction = await _dbContext.BeginTransactionAsync(ct);
```

```
await _dbContext.Products  
    .Where(x => x.Id == request.ProductId)  
    .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsDeleted, true));
```

```
var categoryIds = await _dbContext.ProductCategories  
    .Where(x => x.ProductId == request.ProductId)  
    .Select(x => x.CategoryId)  
    .Distinct()  
    .ToListAsync(cancellationToken);
```

```
foreach (var categoryId in categoryIds)  
{  
    await _sender.Send(new RemoveProductFromCategoryCommand(request.ProductId, categoryId));  
}
```

```
await transaction.CommitAsync(ct);
```

Транзакции в Behavior

```
public class TransactionPipelineBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
    where TRequest : ITransactionRequest
{
    public async Task<TResponse> Handle
        (TRequest request, RequestHandlerDelegate<TResponse> next, CancellationToken ct)
    {
        if (_dbContext.IsTransactionStarted) return await next();

        await using var transaction = await _dbContext.BeginTransactionAsync(ct);

        var result = await next();

        await transaction.CommitAsync(ct);

        return result;
    }
}
```

Вызов сервиса внутри транзакции

```
//begin transaction
```

```
await _externalService.LongOperationAsync(ct);
```

```
await _dbContext.Products  
    .Where(x => x.Id == request.ProductId)  
    .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsDeleted, true));
```

```
//commit transaction
```

Неконсистентное состояние

```
//begin transaction
```

```
_dbContext.Products.Add(product);
```

```
_dbContext.Messages.Add(message);
```

```
await _dbContext.SaveChangesAsync(); //no commit
```

```
await _broker.SendMessage(message);
```

```
//commit transaction
```

Особенности

- Нет дублирования кода
- Производительность ниже
 - Отдельно обрабатываем каждую категорию
- Нужно избавиться от вложенных транзакций

Базовый класс с ивентами

```
public abstract class BaseEntity
{
    protected List<INotification> Notifications = new();

    public List<INotification> FetchNotifications()
    {
        var result = Notifications.ToList();

        Notifications.Clear();

        return result;
    }
}
```

Tovar

```
public record ProductDeletedNotification(int ProductId) : INotification;
```

```
public class Product : BaseEntity  
{  
    public int Id { get; set; }  
    public bool IsDeleted { get; set; }  

```

```
    public void Delete()  
    {  
        IsDeleted = true;  
  
        Notifications.Add(new ProductDeletedNotification(Id));  
    }  

```

```
}
```

Категория

```
public class Category : BaseEntity
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;

    public List<ProductCategory> Products { get; set; } = null!;

    public void RemoveProduct(int productId)
    {
        Products.RemoveAll(x => x.ProductId == productId);
    }
}
```

Удаление товара

```
var product = await _dbContext.Products.FindAsync(request.ProductId);  
  
product!.Delete();  
  
await _dbContext.SaveChangesAsync();
```

Генерация ивентов

```
public class AppDbContext : DbContext, IDbContext
{
    public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken)
    {
        while(true)
        {
            var notifications = ChangeTracker.Entries<BaseEntity>()
                .SelectMany(x => x.Entity.FetchNotifications())
                .ToList();

            if (!notifications.Any()) break;

            foreach (var notification in notifications)
            {
                await _publisher.Publish(notification, cancellationToken);
            }
        }

        return await base.SaveChangesAsync(cancellationToken);
    }
}
```

Ивент хендлер

```
public class ProductDeletedNotificationHandler : INotificationHandler<ProductDeletedNotification>
{
    public async Task Handle(ProductDeletedNotification notification, CancellationToken ct)
    {
        var categories = await _dbSets.Categories
            .Where(x => x.Products.Any(p => p.ProductId == notification.ProductId))
            .Include(x => x.Products)
            .ToListAsync(cancellationToken);

        foreach (var category in categories)
        {
            _categoryService.RemoveProductFromCategory(category, notification.ProductId);
        }
    }
}
```

Абстракция без транзакций

```
public interface IDbSets
{
    DbSet<Product> Products { get; }
    DbSet<Category> Categories { get; }
    DbSet<ProductCategory> ProductCategories { get; }
}

public interface IDbContext : IDbSets
{
    bool IsTransactionStarted { get; }
    Task<IDbContextTransaction> BeginTransactionAsync(CancellationTokens ct);
    Task<int> SaveChangesAsync(CancellationTokens ct);
}
```

Уведомления не влияют на транзакции

```
public class ProductDeletedNotificationHandler :  
    INotificationHandler<ProductDeletedNotification>  
{  
    private readonly IDbSets _dbSets;  
    private readonly CategoryService _categoryService;  
  
    public ProductDeletedNotificationHandler(IDbSets dbSets, CategoryService service)  
    {  
        _dbSets = dbSets;  
        _categoryService = service;  
    }  
}
```



Архитектурный тест (нет IDbContext в ctor)

```
[Fact]
public void NotificationHandlersShouldNotDependOnDbContext()
{
    var notificationHandlers = Types.InAssembly(
        Assembly.GetAssembly(typeof(ProductDeletedNotificationHandler)))
        .That().ImplementInterface(typeof(INotificationHandler<>))
        .GetTypes()
        .ToArray();

    var dependOnDbContext = notificationHandlers
        .Where(x => x.GetConstructors().Single()
            .GetParameters()
            .Any(p => p.ParameterType == typeof(IDbContext)))
        .ToArray();

    Assert.Empty(dependOnDbContext);
}
```

Общий код в сервисе

```
public class CategoryService
{
    private readonly IDbSets _dbSets;

    public CategoryService(IDbSets dbSets)
    {
        _dbSets = dbSets;
    }

    public void RemoveProductFromCategory(Category category, int productId)
    {
        category.RemoveProduct(productId);

        if (!category.Products.Any())
        {
            _dbSets.Categories.Remove(category);
        }
    }
}
```

Убирание товара из категории

```
var category = await _dbContext.Categories
    .Include(x => x.Products) //can't include filter because need to check products count
    .FirstOrDefaultAsync(x => x.Id == request.CategoryId, cancellationToken);

_categoryService.RemoveProductFromCategory(category!, request.ProductId);

await _dbContext.SaveChangesAsync(cancellationToken);
```

Где что лежит

- DomainEvents.Entities
 - Dependencies
 - Models
 - _BaseEntity.cs
 - Category.cs
 - Product.cs
 - ProductCategory.cs
 - Notifications
 - ProductDeletedNotification.cs
- DomainEvents.Infrastructure.Interfaces
 - Dependencies
 - IDbContext.cs
 - IDbSets.cs
- DomainEvents.Infrastructure.MsSql
 - Dependencies
 - Migrations
 - AppDbContext.cs

- DomainEvents.UseCases
 - Dependencies
 - Categories
 - Commands
 - RemoveProductFromCategoryCommand.cs
 - NotificationHandlers
 - ProductDeletedNotificationHandler.cs
 - Products
 - Services
 - CategoryService.cs

Особенности

- Хендлеры максимально чистые
- Вызов хендлера для ивента неявный
- Общий код в сервисе
- Нет проблем с вложенными транзакциями
- Производительность ниже
 - Полностью инициализированные агрегаты

Без DomainEvents

```
var product = await _dbContext.Products.FindAsync(request.ProductId);
```

```
product!.Delete();
```

```
var categories = await _dbContext.Categories
    .Where(x => x.Products.Any(a => a.ProductId == request.ProductId))
    .Include(x => x.Products)
    .ToListAsync(cancellationToken);

foreach (var category in categories)
{
    _categoryService.RemoveProductFromCategory(category, request.ProductId);
}
```

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

Особенности

- Есть переиспользование
- Производительность такая же
- Код не максимально чистый
- Код максимально понятный

Ивент о создании заказа

```
public class Order
{
    private readonly List<INotification> _domainEvents = new ();

    public int Id { get; set; }

    protected Order()
    {
    }

    public Order(int id)
    {
        Id = id;

        _domainEvents.Add(new OrderCreatedEvent(id));
    }
}
```

Без ивента в конструкторе

```
public class Order
{
    private readonly List<INotification> _domainEvents = new ();
    public int Id { get; set; }
    protected Order()
    {
    }
    internal Order(int id)
    {
        Id = id;
        _domainEvents.Add(new OrderCreatedEvent(id));
    }
    internal void AddDomainEvent(INotification evt)
    {
        _domainEvents.Add(evt);
    }
}
```

Фабрика добавляет ивент

```
public class OrderFactory
{
    public Order CreateOrder(int id)
    {
        var result = new Order(id);

        result.AddDomainEvent(new OrderCreatedEvent(id));

        return result;
    }
}
```

ChangeTracker

```
public class AppDbContext : DbContext
{
    public DbSet<Order> Orders { get; set; } = default!;

    public override Task<int> SaveChangesAsync(CancellationToken cancellationToken)
    {
        var orderCreatedEvents = ChangeTracker.Entries<Order>()
            .Where(x => x.State == EntityState.Added)
            .Select(x => new OrderCreatedEvent(x.Entity.Id))
            .ToList();

        // publish events

        return base.SaveChangesAsync(cancellationToken);
    }
}
```

Удаление категории

```
public async Task Handle(DeleteCategoryCommand request)
{
    _dbContext.Categories.Remove(new Category { Id = request.Id });

    await _dbContext.SaveChangesAsync();
}
```

ChangeTracker

```
public class AppDbContext : DbContext
{
    public override Task<int> SaveChangesAsync(CancellationToken ct)
    {
        var categoryDeletedEvents = ChangeTracker.Entries<Category>()
            .Where(x => x.State == EntityState.Deleted)
            .Select(x => new CategoryDeletedEvent(x.Entity.Id))
            .ToList();

        // publish events

        return base.SaveChangesAsync(ct);
    }
}
```

Удаление категории

```
public async Task Handle(DeleteProductCommand request)
{
    _dbContext.Categories.Remove(new Category { Id = request.Id });

    _domainEventService.AddEvent(new CategoryDeletedEvent(request.Id));

    await _dbContext.SaveChangesAsync();
}
```

Получение ивентов

```
public class AppDbContext : DbContext
{
    public DbSet<Order> Orders { get; set; } = default!;

    public override Task<int> SaveChangesAsync(CancellationToken cancellationToken)
    {
        var events = _domainEventService.FetchEvents();

        // publish events

        return base.SaveChangesAsync(cancellationToken);
    }
}
```

Без ChageTracker

```
public async Task Handle(DeleteProductCommand request)
{
    await _dbContext.BeginTransactionAsync();

    await _dbContext.Categories
        .Where(x => x.Id == request.Id)
        .ExecuteDeleteAsync();

    _domainEventService.AddEvent(new CategoryDeletedEvent(request.Id));

    await _dbContext.CommitTransactionAsync();
}
```

Получение ивентов

```
public class AppDbContext : DbContext
{
    public Task CommitTransactionAsync()
    {
        var events = _domainEventService.FetchEvents();

        // publish events

        await _transaction.CommitAsync();
    }
}
```

Важен ли порядок вызова обработчиков

- Нет
- Если важен, то что-то пошло не так)
 - Объединить обработчики в один
 - Выстроить логику в нужном порядке



Можно ли генерить события в EventHandler?

- Лучше не надо)

DomainEvents и EventSourcing

- DomainEvents без EventSourcing - да
- EventSourcing без DomainEvents - да
- EventSourcing и DomainEvents - да

Application Events

История просмотра товаров

```
public class GetProductQueryHandler : IRequestHandler<GetProductQuery, ProductDto>
{
    public async Task<ProductDto> Handle(GetProductQuery request)
    {
        var result = await _dbContext.Products
            .Where(x => x.Id == request.Id)
            .ProjectTo<ProductDto>(_mapper.ConfigurationProvider)
            .FirstOrDefaultAsync(cancellationToken);
        if (result == null) throw new NotFoundException(request.Id);

        // update viewed products

        return result;
    }
}
```

История просмотра товаров

```
public class GetProductQueryHandler : IRequestHandler<GetProductQuery, ProductDto>
{
    public async Task<ProductDto> Handle(GetProductQuery request)
    {
        var result = await _dbContext.Products
            .Where(x => x.Id == request.Id)
            .ProjectTo<ProductDto>(_mapper.ConfigurationProvider)
            .FirstOrDefaultAsync(cancellationToken);
        if (result == null) throw new NotFoundException(request.Id);

        await _publisher.Publish(new ProductViewedEvent(request.Id, UserId));

        return result;
    }
}
```

Стратегии публикации событий

```
public enum PublishStrategy
{
    SyncContinueOnException = 0,
    SyncStopOnException = 1,
    Async = 2,
    ParallelNoWait = 3,
    ParallelWhenAll = 4,
    ParallelWhenAny = 5,
}
```

<https://github.com/jbogard/MediatR/tree/master/samples/MediatR.Examples.PublishStrategies>

ParallelNoWait

```
private Task ParallelNoWait(IEnumerable<NotificationHandlerExecutor> handlers,  
                             INotification notification, CancellationToken cancellationToken)  
{  
    foreach (var handler in handlers)  
    {  
        Task.Run(() => handler.HandlerCallback(notification, cancellationToken));  
    }  
  
    return Task.CompletedTask;  
}
```

История просмотра товаров

```
public class GetProductQueryHandler : IRequestHandler<GetProductQuery, ProductDto>
{
    public async Task<ProductDto> Handle(GetProductQuery request)
    {
        var result = await _dbContext.Products
            .Where(x => x.Id == request.Id)
            .ProjectTo<ProductDto>(_mapper.ConfigurationProvider)
            .FirstOrDefaultAsync(cancellationToken);
        if (result == null) throw new NotFoundException(request.Id);

        await _publisher.Publish(new ProductViewedEvent(request.Id, UserId),
            PublishStrategy.ParallelNoWait);

        return result;
    }
}
```

Outbox – не теряем ApplicationEvents

```
public class GetProductQueryHandler : IRequestHandler<GetProductQuery, ProductDto>
{
    public async Task<ProductDto> Handle(GetProductQuery request)
    {
        await using var transaction = await _dbContext.BeginTransactionAsync(ct);

        var result = await _dbContext.Products
            .Where(x => x.Id == request.Id)
            .ProjectTo<ProductDto>(_mapper.ConfigurationProvider)
            .FirstOrDefaultAsync(ct);
        if (result == null) throw new NotFoundException(request.Id);

        await _dbContext.Messages.Add(new ProductViewedEvent(request.Id, UserId));

        await transaction.CommitAsync(ct);

        return result;
    }
}
```

ApplicationEvents

- Могут быть не только в командах, но и в запросах
- Допускают разные стратегии выполнения кроме SyncStopOnException
- Outbox чтобы не терять

О чем поговорили

- Доменные ивенты
 - Внутри ограниченного контекста
 - Для изменения двух агрегатов в одной транзакции
- Позволяют писать чистый код
- Взаимодействие агрегатов неявное
- Производительность не высокая
- Хендлеры ивентов не должны
 - вызывать команды
 - стартовать или коммитать транзакцию

О чем поговорили

- Application ивенты
 - в командах и запросах
 - разные стратегии выполнения
 - Outbox чтобы не терять

Главные мысли

- DomainEvents – два агрегата в одной транзакции
- Не только домен, но и инфраструктура

Полезные ссылки

- Исходный код примера
<https://github.com/denis-tsv/DomainEvents>

Спасибо за внимание!

Есть вопросы?

Денис Цветчих
DevBrothers



@den_tsvettsikh



den.tsvettsih@yandex.ru



<https://github.com/denis-tsv>

DOTNEXT

