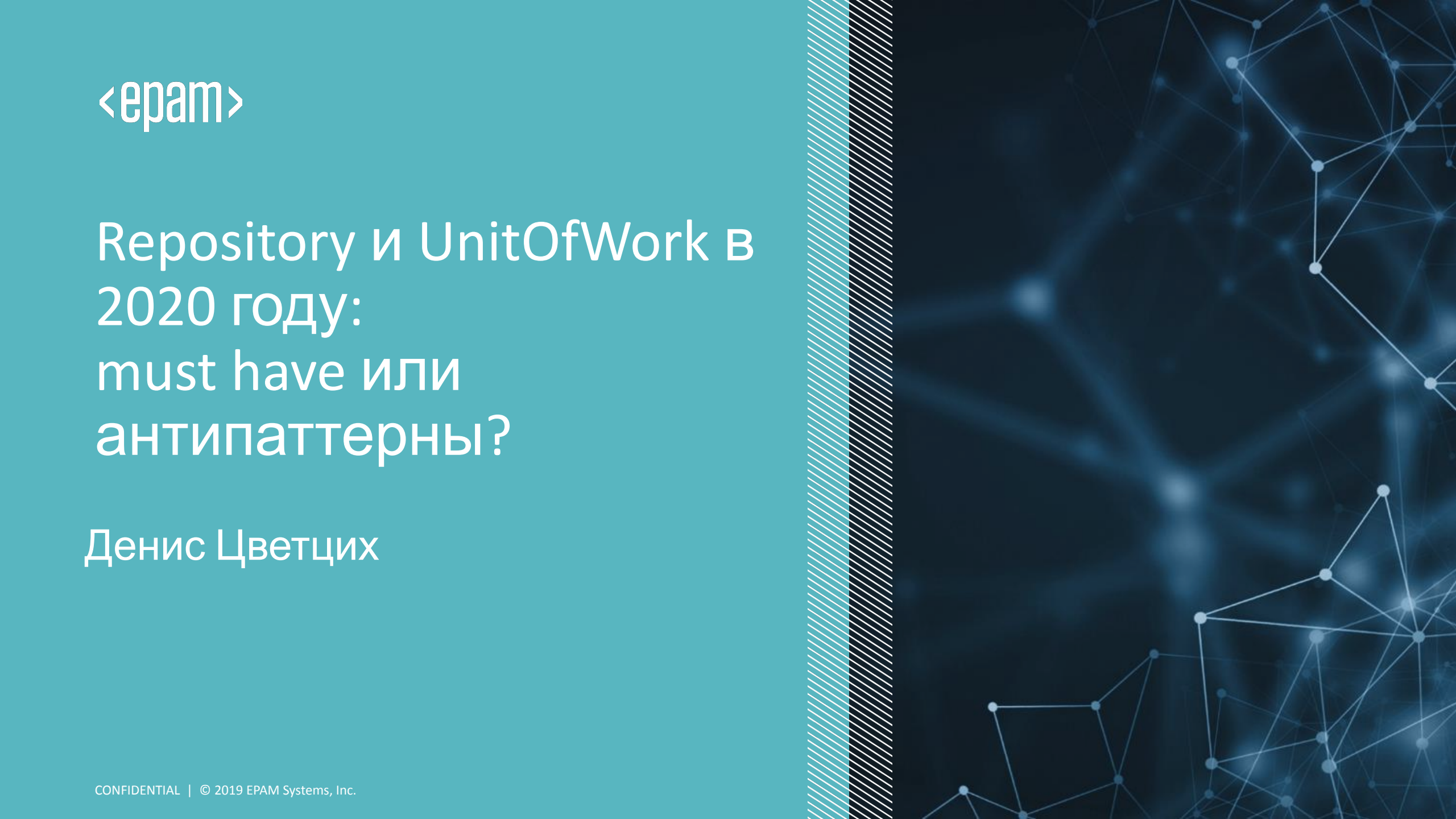


Repository и UnitOfWork в 2020 году: must have или антипаттерны?

Денис Цветцих

Что такое репозиторий

- Абстракция от хранилища данных
- Интерфейс для добавления и удаления аналогичный коллекциям
- Поиск объектов по декларативным запросам

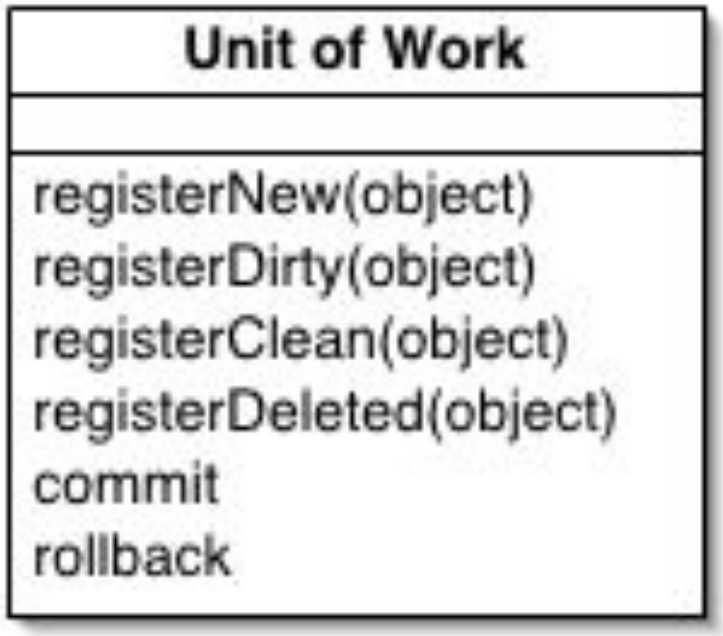
Реализация (Вон Вернон)

```
public class HibernateCalendarEntryRepository
    implements CalendarEntryRepository {

    @Override
    public void add (CalendarEntry aCalendarEntry) {
        try {
            this.session().saveOrUpdate(aCalendarEntry);
        }
        catch ( ConstraintViolationException e ) {
            throw new IllegalStateException("CalendarEntry is not unique.", e);
        }
    }
}
```

UnitOfWork

Управляет *записью изменений*
для *набора объектов*,
изменяемых в *одной бизнес-транзакции*



Опрос

- Кто считает, что эти паттерны актуальны (нужна своя реализация)?
- Кто считает, что нет?

Какие плюсы дает репозиторий:

1. Изоляция доступа к данным в одном месте
2. Работа с зависимыми сущностями через репозиторий агрегата
3. Инкапсуляция специфики SQL для конкретной базы
4. Простота тестирования

Пример многие-ко-многим: модель

```
public class Product
{
    public ICollection<ProductCategory> ProductCategories { get; set; }
}

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

    public Product Product { get; set; }
    public Category Category { get; set; }
}
```

Удаление

```
var product = await _uow.ProductRepository
    .GetWithCategoriesAsync(request.ProductId);
if (product == null)
    throw new NotFoundException("Deleted product not found.");

// Delete categories before product
_uow.ProductCategoryRepository.RemoveRange(product.ProductCategories);
_uow.ProductRepository.Remove(product);
await _uow.SaveChangesAsync();
```

Удаление, чего хочется

```
var product = await _uow.ProductRepository.Remove(request.ProductId);

await _uow.SaveChangesAsync();
```

Корень агрегации: обновление

```
var product = await _uow.ProductRepository
    .GetWithProductCategories(request.ProductId);

// Delete not exists categories
// Add new categories

await _uow.SaveChangesAsync();
```

Удаление категорий

```
//delete not existing in DTO categories
foreach (var category in product.ProductCategories
    .Where(x => !newCategoryIds.Contains(x.CategoryId)))
{
    product.ProductCategories.Remove(category);
}
```

Добавление новых категорий

```
//new categories
foreach (var categoryId in
          newCategoryIds.Except(currentCategoryIds))
{
    product.ProductCategories.Add(new ProductCategory
    {
        CategoryId = categoryId,
        ProductId = product.Id
    });
}
```

Обновление, чего хочется

```
var product = await _uow.ProductRepository.Update(request.Product);

await _uow.SaveChangesAsync();
```

Абстракция ;)

```
public interface IAppUnitOfWork : IUnitOfWork
{
    AppDbContext Context { get; }
}
```

Часто стремятся к такому

```
public interface IUnitOfWork
{
    IRepository<User> UsersRepository { get; }

    IProductRepository<Product> ProductsRepository { get; }

    Task<int> SaveChangesAsync();
}
```

Но получается примерно так

```
public interface IUnitOfWork
{
   IRepository<User> UsersRepository { get; }
   IProductRepository<Product> ProductsRepository { get; }

   Task<int> SaveChangesAsync();

   DbSet<User> Users { get; }
   DbSet<Product> Products { get; }
}
```

Или вот так

```
public interface IAvailableRepository<TEntity> : IRepository<TEntity>
{
    Task<IEnumerable<TEntity>> GetAllAvailableAsync();

    IQueryable<TEntity> AllAvailable { get; } // DbSet
}
```

И скатывается к вот-такому

```
public interface IUnitOfWork
{
    IQueryable<User> Users { get; }

    IQueryable<Product> Products { get; }

    Task<int> SaveChanges();
}
```

Или даже такому

```
public interface IUnitOfWork
{
    DbSet<User> Users { get; }

    DbSet<Product> Products { get; }

    Task<int> SaveChanges();
}
```

Что имеем на практике

1. Изоляция: CRUD код, который должен быть в репозитории, протекает в вызывающий код
2. Создаются репозитории не только для агрегатов
3. SQL ~~никто~~ мало кто пишет
4. Для EF Core есть InMemory (для остальных – SQLite::memory)

Реализация Repository

```
public class Repository<T> where T : class
{
    protected readonly DbSet<T> DbSet;

    public Repository(AppDbContext context)
    {
        DbSet = context.Set<T>();
    }
}
```

Еще одна реализация Repository

```
public class Repository<T> where T : class
{
    protected readonly AppDbContext Context;

    public Repository(AppDbContext context)
    {
       DbContext = context;
    }
}
```

И так тоже бывает ☺️

```
public abstract class AbstractRepository {
    protected readonly AppDbContext Context;

    protected AbstractRepository(AppDbContext context) {
        Context = context;
    }
}

public class Repository<T> : AbstractRepository  {
    protected readonly DbSet<T> DbSet;

    public Repository(AppDbContext context) : base(context) {
        DbSet = context.Set<T>();
    }
}
```

Взгляд из угла ORM

A DbContext instance represents a combination of the Unit Of Work and Repository patterns

Плюсы реализации Repository поверх ORM

Репозиторий и ORM создает сложность

- Нужно писать дополнительный инфраструктурный уровень
- Нужно думать как сделать выборки с Include
- Нужно думать как отключить ChangeTraching для запросов
- Нужно думать как сделать универсальные выборки чтобы не плодить много методов
  - что возвращать из репозитория: IQueryable или Ienumerable
- Нужно думать что использовать в контроллерах, репозитории или сервисы
  - А если сервис только пробрасывает методы репозитория?

Ну может хотя бы запросы?

```
public class Product
{
    public int Quantity { get; set; }
    public bool IsAvailable { get; set;}
    public string Name { get; set; }
}
```

Репозиторий

```
public class ProductRepository
{
    public async Task<List<Product>> GetProductsByName(string name)
    {
        return await _context.Products
            .Where(x => x.IsAvailable && x.Quantity > 0 && // дубль
                        x.Name.Contains(name))
            .ToListAsync();
    }

    public async Task<List<Product>> GetAllProducts()
    {
        return await _context.Products
            .Where(x => x.IsAvailable && x.Quantity > 0) // дубль
            .ToListAsync();
    }
}
```

Метод для инкапсуляции запроса

```
protected IQueryable<Product> GetAvailableProducts()
{
    return _context.Products.Where(x => x.IsAvailable && x.Quantity > 0);
}
```

Его использование в других запросах

```
public async Task<List<Product>> GetProductsByName(string name)
{
    return await GetAvailableProducts()
        .Where(x => x.Name.Contains(name))
        .ToListAsync();
}

public async Task<List<Product>> GetAllProducts()
{
    return await GetAvailableProducts().ToListAsync();
}
```

Спецификация – классика

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T obj);
}
```

Universal.Autofilter спецификация

```
public class Product
{
    public static Spec<Product> AvailableSpec =
        new Spec<Product>(x => x.IsAvailable && x.Quantity > 0);

    public static Spec<Product> ByNameSpec(string name)
    {
        return new Spec<Product>(x => x.Name.Contains(name));
    }
}
```

https://github.com/denis-tsv/AutoFilter

Все продукты

```
public class ProductController
{
    public async Task<List<Product>> GetAllProducts()
    {
        return await _context.Products
            .Where(Product.AvailableSpec)
            .ToListAsync();
    }
}
```

Комбинация спецификаций

```
public async Task<List<Product>> GetProductsByName(string name)
{
    return await _context.Products
        .Where(Product.AvailableSpec && Product.ByNameSpec(name))
        .ToListAsync();
}
```

LinqSpec – класс для каждой спецификации

```
public abstract class Specification<T>
{
    public abstract Expression<Func<T, bool>> ToExpression();
}
```

Реализация LinqSpec

```
public class AvailableProductSpecification : Specification<Product> {

    public override Expression<Func<Product, bool>> ToExpression()     {
        return x => x.IsAvailable && x.Quantity > 0;
    }
}

public class ProductByNameSpecification : Specification<Product> {

    private string _name;

    public ProductByNameSpecification(string name) {
        _name = name;
     }

    public override Expression<Func<Product, bool>> ToExpression()    {
        return x => x.Name.Contains(_name);
    }
}
```

Репозиторий

- Не нужен как абстракция источника данных
- Не нужен для избавления от дублирования в запросах

Чистая архитектура

Дядя Боб: ORM – это инфраструктура, он которой нужно абстрагироваться.

Ага, может быть новая роль Repository и UnitOfWork – абстракция для ORM, а не базы?

Получение списка через ORM

```
internal class GetProductsQueryHandler
{
    private readonly AppDbContext _context;

    public GetProductsQueryHandler(AppDbContext context)
    {
        _context = context;
    }

    public async Task<List<Product>> HandleAsync(GetProductsQuery query)
    {
        return await _context.Products.ToListAsync();
    }
}
```

Получение списка через репозиторий

```
public interface IUnitOfWork
{
    IQueryable<Product> Products { get; }
}

public class EFUnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;

    public EFUnitOfWork(AppDbContext context)
    {
        _context = context;
    }

    public IQueryable<Product> Products => _context.Products;
}
```

Хендлер с UnitOfWork вместо контекта

```
internal class GetProductsQueryHandler
{
    private readonly IUnitOfWork _uow;

    public GetProductsQueryHandler(IUnitOfWork uow)
    {
        _uow = uow;
    }

    public async Task<List<Product>> HandleAsync(GetProductsQuery query)
    {
        return await _uow.Products.ToListAsync();
    }
}
```

Не все так просто ☹️

```
using Microsoft.EntityFrameworkCore;

internal class GetProductsQueryHandler
{
    public async Task<List<Product>> HandleAsync(GetProductsQuery query)
    {
        return await _uof.Products.ToListAsync();
    }
}
```

Как переопределить Extension Method?

- Новый класс QueryableExecutor
- Service Locator

QuerableExecutor

```
public interface IQueryableExecutor
{
    Task<List<T>> ToListAsync<T>(IQueryable<T> source);
    //SingleAsync
    //и остальные асинхронные методы по необходимости
}

public class QueryableExecutor : IQueryableExecutor
{
    public async Task<List<T>> ToListAsync<T>(IQueryable <T> source)
    {
        return EntityFrameworkQueryableExtensions.ToListAsync(source);
    }
}
```

Хендлер с IQueryableExecutor

```
internal class GetProductsQueryHandler
{
    public GetProductsQueryHandler(IUnitOfWork uow,
        IQueryableExecutor executor)
    {
        _uow = uow;
        _executor = executor;
    }

    public async Task<List<Product>> HandleAsync(GetProductsQuery query)
    {
        var query = _uof.Products;
        return await _executor.ToListAsync(query);
    }
}
```

Не забываем про тестирование

```
public class InMemoryQueryableExecutor : IQueryableExecutor
{
    public async Task<List<T>> ToListAsync<T>(IQueryable<T> source)
    {
        return source.ToList();
    }

   //SingleAsync итд
}
```

ServiceLocator и новые extension методы

```
public static class QueryableExtensions
{
    //Инициализация в конфигурации приложения или тесте
    public static IQueryableExecutor QueryableExecutor { get; set; }

    public static Task<List<T>> ToListAsync<T>(this IQueryable<T> source)
    {
        return QueryableExecutor.ToListAsync(source);
    }

   //SingleAsync итд
}
```

Хендлер без QuerableExecutor

```
using Infrastructure.Interfaces;

internal class GetProductsQueryHandler
{
    private readonly IUnitOfWork _uof;

    public GetProductsQueryHandler(IUnitOfWork uof)
    {
        _uof = uof;
    }

    public async Task<List<Product>> HandleAsync(GetProductsQuery query)
    {
        return await _uof.Products.ToListAsync();
    }
}
```

При миграции на другой ORM

```
//EF 6
context.Blogs
    .Include(b => b.Posts.Select(p => p.Comments))
    .ToList();

//EF Core
context.Blogs
    .Include(b => b.Posts).ThenInclude(p => p.Comments)
    .ToList();

//EF 6 и EF Core
 context.Blogs.Include(“Posts.Comments”).ToList();
```

Миграция на другой ORM

- Надо учитывать API тех ORM, между которыми хотим заложить возможность перехода
- Надо продать эти задачи менеджеру/заказчику ;)

Итого UnitOfWork для абстракция ORM

- Не нужна, если не планируется переход на другой ORM
- Переход на другой ORM не планируется никогда ☺️

Как сделать DAL без Repository и UnitOfWork

- Сборка Infrastructure.Interfaces - интерфейс IDbContext
  - Нет реализации OnModelCreating, зависимой от базы
  - По возможности чистая архитектура (без EFCore.MsSql)
  - Все нужные свойства и методы EF контекста (ChangeTracker, DbSet)
- Сборка DataAccess.MsSql (Postgres, …) – то, что зависит от базы
  - AppDbContext - реализация интерфейса IDbContext
  - Миграции, так их проще добавлять
- Дублирующиеся запросы – спецификации и их комбинации

Infrastructure.Interfaces

```
public interface IDbContext
{
    DbSet<Product> Products { get; }

    Task<int> SaveChangesAsync
              (CancellationToken cancellationToken = default);
}
```

Или два контекста для Read и CUD

```
public interface IReadDbContext
{
    DbSet<Product> Products { get; }
}

public interface IDbContext : IReadDbContext
{
    ChangeTracker ChangeTracker  { get; }

    Task<int> SaveChangesAsync();
}
```

DataAccess.MsSql

```
public class AppDbContext : IDbContext
{
    DbSet<Product> Products { get; set; }

    protected override void OnModelCreating
          (ModelBuilder builder)
    {
        //
    }
}
```

Если нужны EF.Functions (полнотекстовый поис

Нужна ли поддержка нескольких баз одновременно?

- Да
  - Делаем абстракции и свои реализации для каждой базы
- Нет
  - Обходимся без оберток ☺️
  - При переходе на другую базу – переписываем

Если дублируется логика сохранения

- Инициализация ChangedAt+ChangedBy
  - Перегрузка SaveChanges у контекста
  - Пост-процессор в пайплайне обработки запроса (MediatR)

Модель данных

```
public class AuditableEntity
{
    public DateTime CreatedAt { get; set; }
    public int CreatedBy { get; set; }

    public DateTime? ModifiedAt { get; set; }
    public int? ModifiedBy { get; set; }
}

public class Entity : AuditableEntity
{
}
```

Перегрузка SaveChanges у контекста

```
public override Task<int> SaveChangesAsync() {
    foreach (var entry in ChangeTracker.Entries<AuditableEntity>())     {
        switch (entry.State)
        {
            case EntityState.Added:
                entry.Entity.CreatedBy = _currentUserService.UserId;
                entry.Entity.CreatedAt = _dateTime.Now;
                break;

            case EntityState.Modified:
                entry.Entity.ModifiedBy = _currentUserService.UserId;
                entry.Entity.ModifiedAt = _dateTime.Now;
                break;
        }
    }

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

Интерфейс для отметки об изменениях

```
public interface IChangeDataRequest
{
}

public class ChangeEntityRequest : IReqiest, IChangeDataRequest
{
}

public class PostProcessor<TRequest, TResponse> :
        IRequestPostProcessor<TRequest, TResponse>
        where TRequest : IChangeDataRequest
{
}
```

Хендлер обновления Entity (MediatR)

```
public class ChangeEntityHandler :
    IRequestHandler<ChangeEntityRequest>
{

    public async Task Handle(ChangeEntityRequest request)
    {
        var entity = await _context.FindAsync<Entity>(request.Id);

        Mapper.Map(request, entity);

        //не вызываем SaveChanges

    }
}
```

Пост-процессор IChangeDataRequest запросов

```
public async Task Process(TRequest request, TResponse response)
{
    _context.ChangeTracker.Entries<AuditableEntity>().ToList()
        .ForEach(x =>  {
                if (x.State == EntityState.Added)
                {
                    x.Entity.CreatedBy = _currentUserService.UserId;
                    x.Entity.CreatedAt = _dateTime.Now;
                }
                if (x.State == EntityState.Modified)
                {
                    x.Entity.ModifiedBy = _currentUserService.UserId;
                    x.Entity.ModifiedAt = _dateTime.Now;
                }
            });

    await _context.SaveChangesAsync();
}
```

Итого, отказ от Repository и UnitOfWork

- Избавляет от мук выбора:
  - Использовать в контроллерах репозитории или сервисы
  - Возвращать из репозитория IQueryable или IEnumerable
  - Как сделать универсальные запросы вместо множества методов
  - Итд
- Избавляет от дополнительного слоя абстракций, который:
  - Протекает
  - Не привносит ничего полезного
  - Требует времени и сил на разработку

А что говорит Вон Вернон?

- От репозиториев будет польза только если у вас есть агрегаты
- Если нет агрегатов – используйте DAO (CRUD для таблиц)
  - Именно это делает ORM
- Логика типа каскадного удаления в репозитории – спорный вопрос ☺️
  - Автору больше нравится помещать ее туда
  - Но это его личный выбор!
- Полезный кейс – отношения 1-1 между таблицами
  - Не настроить каскадное удаление
  - На практике редко встречается

Мораль

- Не только пишем код по образцам дядек из умных книжек
- Но читаем комментарии к нему ;)
- И думаем своей головой!

Полезные ссылки по теме

Что такое репозиторий

- Фаулер , Эванс

Спецификация

- AutoFilter, не нужен отдельный класс для спецификации, есть в nuget
- LinqSpec, отдельный класс для спецификации, есть в nuget
- Доклад Максима Аршинова на DotNext про Linq в Enterprise

Cross-cutting concerns

- Перегрузка SaveChanges у контекста
- Аршинов, доклад Быстрорастворимое проектирование про декораторы
- MediatR – пайплайн путем цепочки вывозов методов, nuget пакет
- Cqrs In Practice – пример велика для пайплайна из декораторов

Холивар про репозиторий

- Нет – автор книги «EF Core in Action»
- Нет – автор EntityFramework.CommonTools
- Нет – Jason Taylor (он говорит, что автор MediatR тоже против)
- Да – Владимир Хориков, в блоге часто встречается репозиторий
- Да – ведущий разработчик Бындю софт

Пример проекта с репозиториями и без них

https://github.com/denis-tsv/DataAccessWithoutRepositoryAndUnitOfWork

http://bit.ly/no-repository

Опрос

- Кто изменил мнение и считает, что он Repoisitory и UnitOfWork больше не нужны?
- А кто остался при своем и думает что они нужны?

Вопрос на подумать ☺️

```
public class UpdateProductCommandHandler : IRequestHandler<UpdateProductCommand>
{
    protected override async Task Handle(UpdateProductCommand request)
    {
        var product = await _dbContext.Products.FindAsync(request.ProductId);

        _mapper.Map(request.ProductDto, product);

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

Спасибо ☺️

Денис Цветцих
https://vk.com/denistsv
den.tsvettsih@yandex.ru