Generic Repository

Последнее обновление: 13.10.2016

Репозиторий представляет паттерн, задача которого заключается в управлении доступом к источнику данных. Класс, реализующий данный паттерн, не содержит бизнесс-логику, не управляет бизнес-процессами, он только содержит операции над данными. Как правило, репозиторий реализует CRUD-интерфейс, то есть представляет операции по извлечению, добавлению, редактированию и удалению данных.

Как правило, репозиторий привязан к одной конкретной сущности или модели, данными которой он управляет. Хотя это необязательно - в репозитории мы можем предусмотреть механизм для загрузки связанных данных из других таблиц, которые связаны с основной моделью, и ряд аналогичных операций. Но тем не менее, часто для управлению одной сущность создается свой репозиторий. Например, если у нас есть классы Phone и Company:

```
public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    public int CompanyId { get; set; }
    public Company Company { get; set; }
}
```

То для каждого из этих классов мы можем создать свой репозиторий.

Если репозитории используют одно и то же подключение, то нередко для организации доступа к одному подключению для всех репозиториев приложения используется другой паттерн - **Unit Of Work**. Класс, который реализует данный паттерн, как правило, содержит набор репозиториев и ряд некоторых общих для них функций.

Но если мы обратимся непосредственно к Entity Framework, то мы можем увидеть, то он уже реализует паттерны Unit Of Work и репозиторий. К примеру, контекст данных EF для выше обозначенных моделей мог бы выглядеть следующим образом:

```
using System.Data.Entity;

public class ApplicationContext : DbContext
{
    public DbSet<Phone> Phones { get; set; }
    public DbSet<Company> Companies { get; set; }
}
```

Фактически класс ApplicationContext представляет реализацию UnitOfWork - он содержит ряд репозиториев. Каждый репозиторий представлен объектом DbSet, с помощью функциональности которого мы можем получать, добавлять, удалять данные.

Возникает вопрос, а нужно ли нам вообще реализовывать паттерн репозиторий, если мы работаем с EF? Ответ зависит от конкретной ситуации. Если мы будем использовать только EF и больше ничего, то для управления доступа к данным нам не надо создавать никаких дополнительных репозиториев. К тому же для многих распространенных СУБД уже есть свои провайдеры для EF 6, поэтому при наличии одного и того же кода мы относительно легко сможем перейти от использования одной СУБД к другой. Основные изменения будут касаться прежде всего конфигурации проекта.

Однако если мы полагаемся на ряд СУБД, которые могут не иметь нормальных провайдеров для EF 6 и интерфейс работы которых сильно отличается от той функциональности, которую предоставляет нам EF, то чтобы привести все технологии работы с БД к общему знаменателю, мы можем реализовать паттерн репозиторий.

К примеру реализуем паттерн репозиторий для работы через EF 6. Вначале создадим интерфейс репозитория:

```
using System;
using System.Collections.Generic;

namespace GenRepApp
{
    public interface IGenericRepository<TEntity> where TEntity : class
    {
        void Create(TEntity item);
        TEntity FindById(int id);
        IEnumerable<TEntity> Get();
        IEnumerable<TEntity> Get(Func<TEntity, bool> predicate);
        void Remove(TEntity item);
        void Update(TEntity item);
    }
}
```

Если у нас несколько классов, функциональность работы с которыми совпадает, то мы можем реализовать Generic Repository, который может работать с разными сущностями.

Теперь создадим базовую реализацию для репозитория, которая применяет Entity Framework:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Entity;
using System.Linq.Expressions;

namespace GenRepApp
{
    public class EFGenericRepository<TEntity> : IGenericRepository<TEntity> where TEntity : class
    {
        DbContext _context;
        DbSet<TEntity> _dbSet;

        public EFGenericRepository(DbContext context)
        {
            _context = context;
            _dbSet = context.Set<TEntity>();
        }

        public IEnumerable<TEntity> Get()
        {
            return _dbSet.AsNoTracking().ToList();
        }

        public IEnumerable<TEntity> Get(Func<TEntity, bool> predicate)
        {
            return _dbSet.AsNoTracking().Where(predicate).ToList();
        }
```

```
        public TEntity FindById(int id)
        {
            return _dbSet.Find(id);
        }

        public void Create(TEntity item)
        {
            _dbSet.Add(item);
            _context.SaveChanges();
        }
        public void Update(TEntity item)
        {
            _context.Entry(item).State = EntityState.Modified;
            _context.SaveChanges();
        }
        public void Remove(TEntity item)
        {
            _dbSet.Remove(item);
            _context.SaveChanges();
        }
    }
}
```

Репозиторий хранит ссылку на контекст и набор DbSet для работы с текущей сущностью. Все методы репозитория фактически вызывают методы DbSet и контекста данных.

Стоит отметить, что в конце каждого метода на изменение данных вызывается метод `_context.SaveChanges()`. При реализации паттерна UnitOfWork этот метод, как правило, вызывается отдельно и реализуется в самом классе UnitOfWork.

Отдельно стоит сказать про загрузку связанных данных. Если у нас навигационные свойства помечены как виртуальные, то с помощью Lazy Loading связанные данные автоматически будут подгружаться к загруженным сущностям. Однако в примере с моделями выше навигационные свойства не виртуальные, и для загрузки данных следует использовать Eager Loading, то есть нам надо использовать метод `Include()`. Однако чтобы использовать этот метод, нам надо точно знать, какие навигационные свойства надо использовать для подгрузки связанных классах, что в случае с generic-реализацией маловероятно. Тем не менее мы можем это сделать.

Итак, добавим в класс репозитория следующие методы:

```
public IEnumerable<TEntity> GetWithInclude(params Expression<Func<TEntity, object>>[] includeProperties)
{
    return Include(includeProperties).ToList();
}

public IEnumerable<TEntity> GetWithInclude(Func<TEntity,bool> predicate,
    params Expression<Func<TEntity, object>>[] includeProperties)
{
    var query =  Include(includeProperties);
    return query.Where(predicate).ToList();
}

private IQueryable<TEntity> Include(params Expression<Func<TEntity, object>>[] includeProperties)
{
    IQueryable<TEntity> query = _dbSet.AsNoTracking();
    return includeProperties
                .Aggregate(query, (current, includeProperty) => current.Include(includeProperty));
}
```

Для загрузки связанных данных здесь определен вспомогательный метод `Include()`. Используя переданный в качестве параметра массив выражений Include и метод Aggregate, он составляет запрос в виде переменной query, которая возвращается в качестве результата.

Этот метод реализуется в методе `GetWithInclude()`, который возвращает массив объектов. Перегруженная версия метода GetWithInclude также добавляет дополнительное условие.

Применение подобных методов:

```
EFGenericRepository<Phone> phoneRepo = new EFGenericRepository<Phone>(new ApplicationContext());

//IEnumerable<Phone> phones = phoneRepo.GetWithInclude(p=>p.Company);
IEnumerable<Phone> phones = phoneRepo.GetWithInclude(x=>x.Company.Name.StartsWith("S"), p=>p.Company);
foreach (Phone p in phones)
{
        Console.WriteLine($"{p.Name} ({p.Company.Name}) - {p.Price}");
}
```

Назад **Содержание** **Вперед**