polishchuk 0 15.6K 31.10.2019

Unit of Work – Паттерны Объектно-Реляционной логики (PoEAA)

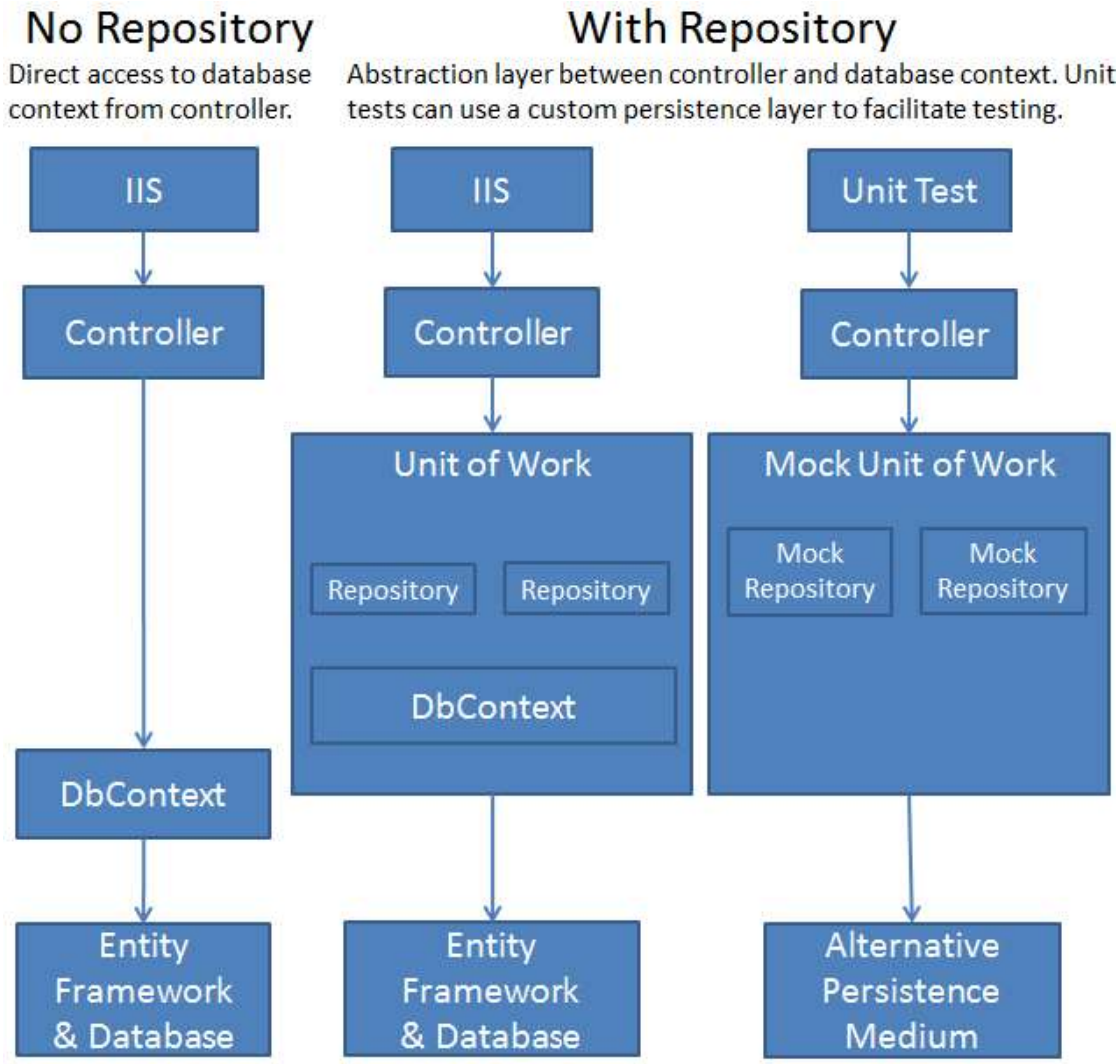
Категории: Программирование Типичные вопросы на собеседовании

Почему нужно использовать Unit Of Work и Repository

Навигация по статье

В приложениях часто юзается шаблон Repository для инкапсуляции логики работы с БД. Часто приходится оперировать множеством сущностей и моделей, для управления которыми создается также большое количество репозиториев. Паттерн Unit of Work помогает упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же DbContext.

Так же использование шаблона **Репозиторий** и **UoW** позволяет создать правильную структуру для развертывания приложения и внедрения DI практик которые в том числе помогают в тестировании проекта:



Паттерн Unit of Work как правило не является полностью самостоятельным, обычно тесно связан с паттерном Identity Map и Metadata Mapping, которые реализованы внутри DbContext'a (если вы используете Entit Framework). Вдаваться в подробности реализации контекста не буду. Опишу пару слов про эти шаблоны PoeAA:

Identity Map — реализует сохранение карты созданных объектов, взятых из стореджа с тем чтобы гарантировать что одна единица информации из хранилища представлена ровно одним экземпляром объекта данных в приложении. Это помогает избежать конфликтов изменений т.к. не допускает ситуации когда два объекта, представляющих один и тот же элемент данных в хранилище, изменены по-разному (по сути реализует уровень изоляции во избежание коллизий данных)

Metadata Mapping — Поскольку для вычисления разницы (и определения того что и каким образом что и где должно быть изменено в хранилище (Tracking Changes)) необходимо знать какие данные и как именно хранятся в объектах - как правило это реализуется с помощью MetaData Mapping паттерна, описывающего связь между содержимым БД (к примеру таблицами и столбцами базы данных) и классами / свойствами объектов.

Реализация паттерна Unit of Work на C#

Прежде чем рассмотреть реализацию UoW, нам нужно рассмотреть реализацию репозитория. Паттерн Репозиторий и его реализация описана тут, по этому не буду повторятся и приведу лишь интерфейс:

```
/// <summary>
 /// Generic repository provide all base needed methods (CRU
 /// </summary>
 public interface IGenericRepository<T> where T : class
 {
     /// <summary>
     /// Persists all updates to the data source
     /// </summary>
     void SaveChanges();

     /// <summary>
     /// Persists all updates to the data source async
```

```
        /// </summary>
        Task SaveChangesAsync();

        /// <summary>
        /// Get first entity by predicate
        /// </summary>
        /// <param name="predicate">LINQ predicate</param>
        /// <returns>T entity</returns>
        T First(Expression<Func<T, bool>> predicate);

        /// <summary>
        /// Get first entity by predicate
        /// </summary>
        /// <param name="predicate"></param>
        /// <returns>T entity</returns>
        T FirstOrDefault(Expression<Func<T, bool>> predicate);

        /// <summary>
        /// Get first entity
        /// </summary>
        /// <returns>T entity</returns>
        T FirstOrDefault();

        /// <summary>
        /// Get first entity async
        /// </summary>
        /// <returns>T entity</returns>
        Task<T> FirstOrDefaultAsync();

        /// <summary>
        /// Get all queries
        /// </summary>
        /// <returns>IQueryable queries</returns>
        IQueryable<T> GetAll();

        /// <summary>
        /// Find queries by predicate (where logic)
        /// </summary>
```

```
        /// <param name="predicate">Search predicate (LINQ)</pa
        /// <returns>IQueryable queries</returns>
        IQueryable<T> FindBy(Expression<Func<T, bool>> predicat

        /// <summary>
        /// Find queries by predicate
        /// </summary>
        /// <param name="predicate">Search predicate (LINQ)</pa
        /// <returns>IQueryable queries</returns>
        bool Any(Expression<Func<T, bool>> predicate);

        /// <summary>
        /// Find entity by keys
        /// </summary>
        /// <param name="keys">Search key</param>
        /// <returns>T entity</returns>
        T Find(params object[] keys);

        /// <summary>
        /// Add new entity
        /// </summary>
        /// <param name="entity">Entity object</param>
        void Add(T entity);

        /// <summary>
        /// Add new entities
        /// </summary>
        /// <param name="entities">Entity collection</param>
        void AddRange(IEnumerable<T> entities);

        /// <summary>
        /// Remove entity from database
        /// </summary>
        /// <param name="entity">Entity object</param>
        void Delete(T entity);

        /// <summary>
        /// Remove entities from database
```

```
        /// </summary>
        /// <param name="entity">Entity object</param>
        void DeleteRange(IEnumerable<T> entity);

        /// <summary>
        /// Update entity
        /// </summary>
        /// <param name="entity">Entity object</param>
        void Update(T entity);

        /// <summary>
        /// Order by
        /// </summary>
        IOrderedQueryable<T> OrderBy<K>(Expression<Func<T, K>>

        /// <summary>
        /// Order by
        /// </summary>
        IQueryable<IGrouping<K, T>> GroupBy<K>(Expression<Func<

        /// <summary>
        /// Remove range of given entities
        /// </summary>
        void RemoveRange(IEnumerable<T> entities);
    }
```

Реализацию Unit of Work я позаимствовал у юзера Arch в гитхабе. И немножко улучшил его.

Давайте рассмотрим для начала интерфейсы.

IUnitOfWork и UnitOfwork‹TContext›

```
/// <summary>
  /// Defines the interface(s) for generic unit of work.
  /// </summary>
```

```
public interface IUnitOfWork<TContext> : IUnitOfWork where
{
    /// <summary>
    /// Gets the db context.
    /// </summary>
    /// <returns>The instance of type <typeparamref name="
    TContext DbContext { get; }

    /// <summary>
    /// Saves all changes made in this context to the datab
    /// </summary>
    /// <param name="ensureAutoHistory"><c>True</c> if save
    /// <param name="unitOfWorks">An optional <see cref="IU
    /// <returns>A <see cref="Task{TResult}"/> that represe
    Task<int> SaveChangesAsync(bool ensureAutoHistory = fa
}
/// <summary>
/// Defines the interface(s) for unit of work.
/// </summary>
public interface IUnitOfWork : IDisposable
{
    /// <summary>
    /// Changes the database name. This require the databas
    /// </summary>
    /// <param name="database">The database name.</param>
    /// <remarks>
    /// This only been used for supporting multiple databas
    /// </remarks>
    void ChangeDatabase(string database);

    /// <summary>
    /// Gets the specified repository for the <typeparamre
    /// </summary>
    /// <param name="hasCustomRepository"><c>True</c> if pr
    /// <typeparam name="TEntity">The type of the entity.</
    /// <returns>An instance of type inherited from <see cr
    IGenericRepository<TEntity> GetRepository<TEntity>(boo
```

```
/// <summary>
/// Saves all changes made in this context to the datab
/// </summary>
/// <param name="ensureAutoHistory"><c>True</c> if say
/// <returns>The number of state entries written to the
int SaveChanges(bool ensureAutoHistory = false);

/// <summary>
/// Asynchronously saves all changes made in this unit
/// </summary>
/// <param name="ensureAutoHistory"><c>True</c> if save
/// <returns>A <see cref="Task{TResult}"/> that represe
Task<int> SaveChangesAsync(bool ensureAutoHistory = fa

/// <summary>
/// Executes the specified raw SQL command.
/// </summary>
/// <param name="sql">The raw SQL.</param>
/// <param name="parameters">The parameters.</param>
/// <returns>The number of state entities written to da
int ExecuteSqlCommand(string sql, params object[] para

/// <summary>
/// Uses raw SQL queries to fetch the specified <typepa
/// </summary>
/// <typeparam name="TEntity">The type of the entity.</
/// <param name="sql">The raw SQL.</param>
/// <param name="parameters">The parameters.</param>
/// <returns>An <see cref="IQueryable{T}"/> that conta
IQueryable<TEntity> FromSql<TEntity>(string sql, params

/// <summary>
/// Uses TrakGrap Api to attach disconnected entities
/// </summary>
/// <param name="rootEntity"> Root entity</param>
/// <param name="callback">Delegate to convert Object's
```

```
        void TrackGraph(object rootEntity, Action<EntityEntryGr
    }
```

Реализация `IUnitOfWork<TContext>` в данном случае нужна для поддержки нескольких БД. Если в вашем случае это не нужно, можно упустить его реализацию

Реализация UnitOfWork выглядит следующим образом:

```
    /// <summary>
    /// Represents the default implementation of the <see cref=
    /// </summary>
    /// <typeparam name="TContext">The type of the db context.<
    public class UnitOfWork<TContext> : IRepositoryFactory, IUn
    {
        private bool _disposed;
        private Dictionary<Type, object> _repositories;

        /// <summary>
        /// Initializes a new instance of the <see cref="UnitO
        /// </summary>
        /// <param name="context">The context.</param>
        public UnitOfWork(TContext context)
        {
            DbContext = context ?? throw new ArgumentNullExcep
        }

        /// <summary>
        /// Gets the db context.
        /// </summary>
        /// <returns>The instance of type <typeparamref name="
        public TContext DbContext { get; }

        /// <summary>
        /// Changes the database name. This require the databa
        /// </summary>
```

```
        /// <param name="database">The database name.</param>
        /// <remarks>
        /// This only been used for supporting multiple databas
        /// </remarks>
        public void ChangeDatabase(string database)
        {
            var connection = DbContext.Database.GetDbConnectior
            if (connection.State.HasFlag(ConnectionState.Open))
            {
                connection.ChangeDatabase(database);
            }
            else
            {
                var connectionString = Regex.Replace(connectior
                connection.ConnectionString = connectionString;
            }

            // Following code only working for mysql.
            var items = DbContext.Model.GetEntityTypes();
            foreach (var item in items)
            {
                if (item.Relational() is RelationalEntityTypeAr
                {
                    extensions.Schema = database;
                }
            }
        }

        /// <summary>
        /// Gets the specified repository for the <typeparamre
        /// </summary>
        /// <param name="hasCustomRepository"><c>True</c> if pr
        /// <typeparam name="TEntity">The type of the entity.</
        /// <returns>An instance of type inherited from <see cr
        public IGenericRepository<TEntity> GetRepository<TEntit
        {
            if (_repositories == null)
            {
```

```
            _repositories = new Dictionary<Type, object>()
        }

        // what's the best way to support custom reposity?
        if (hasCustomRepository)
        {
            var customRepo = DbContext.GetService<IGenericR
            if (customRepo != null)
            {
                return customRepo;
            }
        }

        var type = typeof(TEntity);
        if (!_repositories.ContainsKey(type))
        {
            _repositories[type] = new GenericRepository<TEn
        }

        return (IGenericRepository<TEntity>)_repositories[t
    }

    /// <summary>
    /// Executes the specified raw SQL command.
    /// </summary>
    /// <param name="sql">The raw SQL.</param>
    /// <param name="parameters">The parameters.</param>
    /// <returns>The number of state entities written to da
    public int ExecuteSqlCommand(string sql, params object

    /// <summary>
    /// Uses raw SQL queries to fetch the specified <typepa
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</
    /// <param name="sql">The raw SQL.</param>
    /// <param name="parameters">The parameters.</param>
    /// <returns>An <see cref="IQueryable{T}" /> that conta
    public IQueryable<TEntity> FromSql<TEntity>(string sql,
```

```
/// <summary>
/// Saves all changes made in this context to the datab
/// </summary>
/// <returns>The number of state entries written to the
public int SaveChanges(bool ensureAutoHistory = false)
{
    return DbContext.SaveChanges();
}

/// <summary>
/// Asynchronously saves all changes made in this unit
/// </summary>
/// <returns>A <see cref="Task{TResult}"/> that represe
public async Task<int> SaveChangesAsync(bool ensureAuto
{
    return await DbContext.SaveChangesAsync();
}

/// <summary>
/// Saves all changes made in this context to the datab
/// </summary>
/// <param name="ensureAutoHistory"><c>True</c> if save
/// <param name="unitOfWorks">An optional <see cref="IU
/// <returns>A <see cref="Task{TResult}"/> that represe
public async Task<int> SaveChangesAsync(bool ensureAuto
{
    using (var ts = new TransactionScope())
    {
        var count = 0;
        foreach (var unitOfWork in unitOfWorks)
        {
            count += await unitOfWork.SaveChangesAsync(
        }

        count += await SaveChangesAsync(ensureAutoHisto

        ts.Complete();
```

```
            return count;
        }
    }

    /// <summary>
    /// Performs application-defined tasks associated with
    /// </summary>
    public void Dispose()
    {
        Dispose(true);

        GC.SuppressFinalize(this);
    }

    /// <summary>
    /// Performs application-defined tasks associated with
    /// </summary>
    /// <param name="disposing">The disposing.</param>
    protected virtual void Dispose(bool disposing)
    {
        if (!_disposed)
        {
            if (disposing)
            {
                // clear repositories
                _repositories?.Clear();

                // dispose the db context.
                DbContext.Dispose();
            }
        }

        _disposed = true;
    }

    public void TrackGraph(object rootEntity, Action<Entity
    {
```

```
            DbContext.ChangeTracker.TrackGraph(rootEntity, cal
        }
    }
```

Конфигурация проекта

Для того чтобы "завести" эту машину на нашем проекте, давайте добавим пару extension методов для использования их в Startup'е

```
    public static IServiceCollection AddUnitOfWork<TContext>(t
        {
            services.AddScoped<IRepositoryFactory, UnitOfWork<T
            services.AddScoped<IUnitOfWork, UnitOfWork<TContext
            services.AddScoped<IUnitOfWork<TContext>, UnitOfWor

            return services;
        }

    public static IServiceCollection AddCustomRepository<TEntity
            where TEntity : class
            where TRepository : class, IGenericRepository<TEnt
        {
            services.AddScoped<IGenericRepository<TEntity>, TRe

            return services;
        }
```

Теперь инициализируем наш `UnitOfWork` в Startup'е в методе `ConfigureServices`. Для тестирования я буду использовать InMemoryDb.

```
public void ConfigureServices(IServiceCollection services)
        {
            services
```

```
                .AddDbContext<BloggingContext>(opt => opt.UseIn
                .AddUnitOfWork<BloggingContext>()
                .AddCustomRepository<Blog, CustomBlogRepository

            services.AddMvc();
        }
```

Использование паттерна UnitOfWork в Asp.NET Core controller'e

Юзание UoW в контроллере будет выглядеть приблизительно так:

```
[Route("api/[controller]")]
    public class ValuesController : Controller
    {
        private readonly IUnitOfWork _unitOfWork;
        public ValuesController(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
            _logger = logger;
        }

        // GET api/values/4
        [HttpGet("{id}")]
        public async Task<Blog> Get(int id)
        {
            return await _unitOfWork.GetRepository<Blog>().Find

        }

        // POST api/values
        [HttpPost]
        public async Task Post([FromBody]Blog value)
        {
            var repo = _unitOfWork.GetRepository<Blog>();
            repo.Insert(value);
```

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

C# PoEAA Архитектура ПО Паттерны

Комментарии:

Пожалуйста авторизируйтесь, чтобы получить возможность оставлять комментарии