

# LINQ to SQL: паттерн Repository



В этой статье будет рассмотрен один из вариантов реализации паттерна **репозиторий** на базе LINQ to SQL.

Сегодня LINQ to SQL – это одна из технологий Microsoft, предназначенная для решения проблемы объектно-реляционного отображения (**object-relational mapping**). Альтернативная технология Entity Framework является более мощным инструментом, однако у LINQ to SQL есть свои преимущества – относительная простота и низкоуровневость.

Данная статья — это попытка продемонстрировать сильные стороны LINQ to SQL. Паттерн репозиторий отлично ложится на парадигму LINQ to SQL.

## Репозиторий

Для начала вспомним, что такое репозиторий.

```
public interface IRepository<T> where T: Entity
{
    IQueryable<T> GetAll();
    bool Save(T entity);
    bool Delete(int id);
    bool Delete(T entity);
}
```

\* This source code was highlighted with **Source Code Highlighter**.

Репозиторий – это фасад для доступа к базе данных. Весь код приложения за пределами репозитория работает с базой данных через него и только через него. Таким образом, репозиторий инкасулирует в себе логику работы с базой данных, это слой объектно-реляционного отображения в нашем приложении. Более точно, репозиторий, или хранилище, это интерфейс для доступа к данным одного типа – один класс модели, одна таблица базы данных в простейшем случае. Доступ к данным организуется через совокупность всех репозиторийей. Обратите внимание, что интерфейс репозитория задается в терминах модели приложения: *Entity* – базовый класс для всех классов модели приложения (**POCO**-объекты).

```
public abstract class Entity
{
    protected Entity()
```

```

{
    Id = -1;
}

public int Id { get; set; }

public bool IsNew()
{
    return Id == -1;
}
}

* This source code was highlighted with Source Code
Highlighter.

```

Вообще говоря, атрибут *Id* необходим только на уровне базы данных. На уровне модели приложения уникальность объектов может разрешаться без использования явного идентификатора. Таким образом, предлагаемое решение не совсем честное решение проблемы объектно-реляционного отображения с теоретической точки зрения. Однако на практике использование атрибута первичного ключа в модели приложения часто приводит к получению даже более гибких схем. Предлагаемое решение — компромисс между уровнем абстракции слоя базы данных и гибкостью архитектуры.

Методы интерфейса *IRepository* обеспечивают полный набор CRUD-операций.

**GetAll** – возвращает всю совокупность объектов данного типа, хранимых в БД. Фильтрация, сортировка и другие операции над выборкой объектов осуществляются на более высоком уровне, благодаря использованию интерфейса *IQueryable<T>*. Подробнее в разделе «Фильтры и конвейер».

**Save** – сохраняет объект модели в базе данных. В случае, если он новый, выполняется операция INSERT, иначе – UPDATE.

**Delete** – удаляет объект из базы данных. Предусмотрены два варианта вызова функции: с параметром *id* удаляемой записи и с параметром объектом класса модели приложения.

## Реализация

Пусть у нас есть БД, состоящая из одной таблицы Customers.

```

CREATE TABLE dbo.Customers
(
    [Id] int IDENTITY(1,1) NOT NULL PRIMARY

```

```

KEY,
    [Name] nvarchar(200) NOT NULL,
    [Address] nvarchar(1000) NULL,
    [Balance] money NOT NULL
)
--

```

\* This source code was highlighted with Source Code Highlighter.

Для начала добавим в проект файл *dbml*, в котором будут задаваться классы объектов модели базы данных и свойства их отображения. Для этого надо воспользоваться контекстным меню Solution Explorer (*New Item...->Data->LINQ to SQL Classes*) в Visual Studio. После появления окна дизайнера следует открыть Server Explorer и перетащить таблицу Customers в окно дизайнера. Вот что должно получиться:



В результате, Visual Studio сгенерирует класс *Customer* модели базы данных. Модель самого приложения в общем случае отличается от модели базы данных, но в данном примере они практически совпадают. Ниже приведено описание класса *Customer* модели приложения:

```

public class Customer : Entity
{
    public string Name { get; set; }
    public string Address { get; set; }
    public decimal Balance { get; set; }
}
--

```

\* This source code was highlighted with Source Code Highlighter.

Пришло время заняться реализацией *CustomersRepository* – репозитория объектов типа *Customer*. Для того, чтобы избежать дублирования кода при создании репозитория для других классов модели, большая часть функциональности вынесена в базовый класс.

```

public abstract class RepositoryBase<T, DbT> :
    IRepository<T>
    where T : Entity where DbT : class, IDbEntity,
    new()
{
    protected readonly DbContext context = new
    DbContext();
}

```

```
public IQueryable<T> GetAll()
{
    return GetTable().Select(GetConverter());
}

public bool Save(T entity)
{
    DbT dbEntity;

    if (entity.IsNew())
    {
        dbEntity = new DbT();
    }
    else
    {
        dbEntity = GetTable().Where(x => x.Id ==
entity.Id).SingleOrDefault();
        if (dbEntity == null)
        {
            return false;
        }
    }

    UpdateEntry(dbEntity, entity);

    if (entity.IsNew())
    {
        GetTable().InsertOnSubmit(dbEntity);
    }

    context.SubmitChanges();

    entity.Id = dbEntity.Id;
    return true;
}

public bool Delete(int id)
{
    var dbEntity = GetTable().Where(x => x.Id ==
id).SingleOrDefault();

    if (dbEntity == null)
```

```

        {
            return false;
        }

        GetTable().DeleteOnSubmit(dbEntity);

        context.SubmitChanges();
        return true;
    }

    public bool Delete(T entity)
    {
        return Delete(entity.Id);
    }

    protected abstract Table<DbT> GetTable();
    protected abstract Expression<Func<DbT, T>>
GetConverter();
    protected abstract void UpdateEntry(DbT
dbEntity, T entity);
}

* This source code was highlighted with Source Code
Highlighter.

```

Все классы модели LINQ to SQL имеют общий интерфейс *IDbEntity*:

```

public interface IDbEntity
{
    int Id { get; }
}

* This source code was highlighted with Source Code
Highlighter.

```

К сожалению, средства визуального дизайнера не позволяют указать базовый класс для объектов LINQ to SQL. Для этого необходимо открыть файл *dbml* в редакторе XML (Open with...) и указать атрибут *EntityBase* у элемента *Database*:

```

<Database EntityBase="Data.Db.IDbEntity" ...>

* This source code was highlighted with Source Code

```

Highlighter.

Далее приведено описание класса *CustomersRepository*.

```
public class CustomersRepository :
RepositoryBase<Customer, Db.Entities.Customer>
{
    protected override Table<Db.Entities.Customer>
GetTable()
    {
        return context.Customers;
    }

    protected override
Expression<Func<Db.Entities.Customer, Customer>>
GetConverter()
    {
        return c => new Customer
        {
            Id = c.Id,
            Name = c.Name,
            Address = c.Address,
            Balance = c.Balance
        };
    }

    protected override void
UpdateEntry(Db.Entities.Customer dbCustomer, Customer
customer)
    {
        dbCustomer.Name = customer.Name;
        dbCustomer.Address = customer.Address;
        dbCustomer.Balance = customer.Balance;
    }
}
```

\* This source code was highlighted with Source Code  
Highlighter.

**Фильтры и конвейер**

Метод *GetAll* репозитория возвращает объект, реализующий интерфейс *IQueryable<T>*. Это позволяет применять к выборке объектов операции фильтрации (метод *Where*), сортировки и любые другие операции, определенные над *IQueryable<T>*.

Для удобства часто употребляемые операции могут быть вынесены в extension-методы. Например, фильтрация по имени клиента.

```
public static IQueryable<Customer> WithNameLike(this
IQueryable<Customer> q, string name)
{
    return q.Where(customer =>
customer.Name.StartsWith(name));
}

* This source code was highlighted with Source Code
Highlighter.
```

Теперь мы можем использовать репозиторий следующим образом.

```
IRepository<Customer> rep = new CustomersRepository();
foreach (var cust in
rep.GetAll().WithNameLike("Google").OrderBy(x => x.Name)) { ...
}

* This source code was highlighted with Source Code
Highlighter.
```

Неважно, какой сложности фильтры или другие операции, которые мы используем. Неважно сколько их. В результате будет выполнен ровно один запрос к базе данных. Этот принцип называется отложенным выполнением запросов (*deferred execution*) – итоговый SQL-запрос генерируется и исполняется только в момент, когда требуется получить итоговую выборку. В данном случае, это происходит непосредственно перед выполнением первой итерации цикла *foreach*.

Важное преимущество архитектуры – фильтры, как и всё приложение за исключением слоя репозитория, работают над моделью приложения, а не над моделью базы данных.

## Анализ

Далее проводится анализ генерируемых LINQ to SQL запросов к базе данных при выполнении той или иной операции над репозиторием.

*GetAll*. В случае примера:

```
rep.GetAll().WithNameLike("Google").OrderBy(x => x.Name)
```

\* This source code was highlighted with Source Code Highlighter.

Делается единственный запрос:

```
exec sp_executesql N'SELECT [t0].[Name], [t0].[Address],  
[t0].[Balance], [t0].[Id]  
FROM [dbo].[Customers] AS [t0]  
WHERE [t0].[Name] LIKE @p0  
ORDER BY [t0].[Name]',N'@p0 nvarchar(7)',@p0=N'Google%'
```

\* This source code was highlighted with Source Code Highlighter.

Метод *Save* для нового объекта выполняет единственный запрос INSERT. Например:

```
exec sp_executesql N'INSERT INTO [dbo].[Customers]([Name],  
[Address], [Balance])  
VALUES (@p0, @p1, @p2)  
SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]',N'@p0  
nvarchar(6),@p1 nvarchar(3),@p2  
money',@p0=N'Google',@p1=N'USA',@p2=$10000.0000
```

\* This source code was highlighted with Source Code Highlighter.

В случае вызова *Save* для существующего объекта или *Delete* выполняются два запроса. Первый – извлечение записи из базы данных. Например:

```
exec sp_executesql N'SELECT [t0].[Id], [t0].[Name], [t0].  
[Address], [t0].[Balance]  
FROM [dbo].[Customers] AS [t0]  
WHERE [t0].[Id] = @p0',N'@p0 int',@p0=29
```

\* This source code was highlighted with Source Code Highlighter.



Второй запрос – непосредственное выполнение операций UPDATE или DELETE, соответственно. Пример для DELETE:

```
exec sp_executesql N'DELETE FROM [dbo].[Customers] WHERE
([Id] = @p0) AND ([Name] = @p1) AND ([Address] = @p2) AND
([Balance] = @p3)', N'@p0 int, @p1 nvarchar(6), @p2
nvarchar(3), @p3
money', @p0=29, @p1=N'Google', @p2=N'USA', @p3=$10000.0000

* This source code was highlighted with Source Code
Highlighter.
```

В случае UPDATE и DELETE первый запрос является избыточным, однако без него не удастся сохранить или удалить объект, используя стандартные средства LINQ to SQL.

Один из вариантов избавления от ненужного запроса – использование хранимых процедур.

## Заключение

Основная цель статьи – дать общее представление о паттерне репозиторий и его реализации на LINQ to SQL. Рассмотренный пример применения подхода слишком прост. В реальных приложениях возникает множество проблем при реализации данной архитектуры. Вот некоторые из них.

- Преобразование между объектом модели базы данных и объектом модели приложения может быть значительно более сложным. В таких случаях, невозможно реализовать фильтры над моделью приложения так, чтобы итоговый запрос можно было транслировать в SQL.
- Часто в качестве результата выборки необходимо получить результат соединения (JOIN) нескольких таблиц, а не данные лишь одной таблицы.
- Не все SQL-операции и функции имеют свой эквивалент в LINQ.

Большинство проблем решаемы, но эти вопросы выходят за рамки данной статьи.

[Исходный код к статье](#) (проект ASP.NET MVC).

## Ссылки по теме

[Паттерн Repository \(Martin Fowler\)](#)

[Сравнение LINQ to SQL с Entity Framework](#)


Storefront MVC (screencasts):

Repository Pattern

Pipes and Filters

 **habrahabr.ru**

 <http://habrahabr.ru/blogs/net/52173/>

 <http://goo.gl/EJfIq>

