# BLToolkit. Intro

Уже достаточно давно меня интересует тема CQRS, но пока что все останавливалось на изучении документов, примеров и самому написать что-то используя подход CQRS не доводилось. В последние дни работа в этом направлении активизировалась с новой силой, и пошлел процесс по исследованию инструментов более всего пригодных для реализации этой концепции. Наверно так повлияла конференция Patterns'n'Practices, когда было объявлено что к концу года команда выпустит гайдлайны по реализации CQRS на практике.

Одной из ключевых вещей является непосредственное получение информации из базы данных без промежуточных программных слоев. Такое получение данных должно быть быстрое и простое. Простое и быстрое. По многочисленным источникам и по результатам сайта ORM Battle был выбран для экспериментов BLToolkit.

Насчет скорости работы есть диаграмма сравнения с другими фреймворками на 30 июля 2011 года:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | 1000 item sequence | |
| Test name | Minimum | Maximum | SqlClient | Entity Framework | LINQ to SQL | BLToolkit | DataObjects.Net | LinqConnect | NHibernate | OpenAccess | Subsonic | Unit |
| **CRUD Performance:** | | | | | | | | | | | | |
| Fetch | 0 | | 20 808 | 5 316 | 8 855 | 11 663 | 8 086 | 12 007 | 6 070 | 5 493 | 8 332 | op/s |
| Single Operation: | | | | | | | | | | | | |
| Create Instance | 0 | | 17 048 | 3 619 | 3 409 | 11 047 | 6 737 | 8 478 | 3 599 | 8 914 | 3 269 | op/s |
| Update Instance | 0 | | 16 738 | 4 128 | 1 723 | 10 730 | 8 770 | 10 405 | 683 | 8 531 | 3 788 | op/s |
| Remove Instance | 0 | | 17 666 | 4 781 | 1 950 | 11 193 | 9 144 | 11 130 | 1 149 | 9 870 | 4 004 | op/s |
| CUD Average | 0 | | 17 142 | 4 105 | 2 163 | 10 986 | 8 036 | 9 853 | 1 148 | 8 972 | 3 602 | op/s |
| Multiple Operations: | | | | | | | | | | | | |
| Create Instance | 0 | | 39 342 | 5 383 | 5 124 | 19 329 | 19 086 | 14 929 | 12 417 | 14 443 | 3 711 | op/s |
| Update Instance | 0 | | 41 532 | 6 571 | 2 101 | 39 244 | 21 465 | 21 876 | 13 413 | 13 729 | 3 788 | op/s |
| Remove Instance | 0 | | 25 463 | 7 026 | 2 244 | 60 092 | 25 306 | 27 844 | 14 416 | 24 660 | 3 982 | op/s |
| CUD Average | 0 | | 33 750 | 6 246 | 2 683 | 31 926 | 21 519 | 19 968 | 13 277 | 16 421 | 3 738 | op/s |
| | | | | | | | | | | | | |
| **Data Access Performance:** | | | | | | | | | | | | |
| Query: | | | | | | | | | | | | |
| LINQ Query | 0 | | n/a | 563 | 743 | 8 064 | 1 414 | 2 980 | 1 008 | 3 298 | 231 | queries/s |
| Compiled LINQ Query | 0 | | n/a | 6 586 | 9 575 | 12 601 | 9 029 | 9 732 | n/a | 3 383 | n/a | queries/s |
| Native Query | 0 | | 18 017 | 8 758 | n/a | 15 374 | 10 898 | n/a | 3 945 | 6 174 | 7 656 | queries/s |
| Paging (LINQ only): | | | | | | | | | | | | |
| Get Small Page (10 items) | 0 | | n/a | 4 600 | 7 779 | 10 608 | 6 769 | 6 912 | 707 | 3 057 | 221 | pages/s |
| Get Average Page (20 items) | 0 | | n/a | 3 708 | 7 050 | 9 174 | 5 246 | 5 290 | 631 | 2 805 | 212 | pages/s |
| Get Large Page (50 items) | 0 | | n/a | 2 348 | 5 373 | 6 494 | 3 198 | 3 151 | 496 | 2 235 | 191 | pages/s |
| Get Huge Page (100 items) | 0 | | n/a | 1 467 | 3 910 | 4 538 | 1 929 | 1 890 | 325 | 1 671 | 189 | pages/s |
| Materialization: | | | | | | | | | | | | |
| LINQ Materialize | 0 | | n/a | 125 922 | 455 497 | 549 480 | 215 973 | 208 281 | 37 802 | 268 046 | 79 624 | objects/s |
| Native Materialize | 0 | | 1 186 661 | 201 328 | n/a | 888 651 | 281 056 | n/a | 49 879 | 344 317 | 124 055 | objects/s |
| | | | | | | | | | | | | |
| **File sizes (in bytes):** | | | | | | | | | | | | |
| **Runtime libraries (.dlls)** | | | n/a | n/a | n/a | 1 584 128 | 4 345 856 | 911 872 | 2 481 844 | 4 755 968 | 285 184 | bytes |
| | | | | | | | | | | | | |
| **Color bar** | | | | | | | **Worst result** | | | | | **Best result** |
| | | | | | | | | | | | | |
| **Units:** | | | | | | | | | | | | |
| **op/s** | operations per second, more is better | | | | | | | | | | | |
| **queries/s** | queries per second, more is better | | | | | | | | | | | |
| **pages/s** | pages per second, more is better | | | | | | | | | | | |
| **objects/s** | # of materialized objects per second, more is better | | | | | | | | | | | |

Не могу не вспомнить фразу с презентации на NDC2011 по поводу Kill your ORM, где ведущий высказался в таком духе: «Linq2Sql вышел слишком простым и быстрым, поэтому придумали EF. Больше абстракций!».

Первой фазой я решил проверить легкость использования и различные нюансы построения классов для маппинга, так как обычно уже на этой фазе начинают вылезать разные неприятные моменты использования.

На официальном сайте есть что-то в духе вводного курса, но как обычно это и бывает, многие вещи писатели держали в уме и считали само-собой разумеющееся. Разобраться можно, но последовательность повествования надо поменять, что я и попробую сейчас сделать, надеюсь, что получится яснее.

## Возможности BLToolkit

Сами разработчики относят свое детище к легковесным ORM, т.е. таким которые управляют только получением/сохранением данных, но не поддерживают связи между сущностями базы данных. По умолчанию не поддерживает. Хотя можно указать маппинг на загрузку, но тогда классы становятся несколько перегруженными аннотациями. Впрочем, в разрезе CQRS меня не очень интересуют возможности подгрузки связанных данных.

Связывание данных делается «по соглашениям», и реже «по контракту» то, что называется convention over contract, хотя возможность указать явно таблицу или процедуру существует. Так же есть возможность задать правило, по которому будут конструироваться имена процедур.

Библиотека предоставляет базовые средства для аудита и логирования, поддерживает linq, рефлексия, построение типов и что-то еще, что пока-что меня никак не интересует, так как свой ORM писать в планы не входит никоим образом.

## Подготовка к работе

По результатам прочтения документации и примеров получается, что самым простым способом для получения данных будет использование атрибутов с текстом запроса. Но обо всем по порядку. Начнем с создания проекта.

Для создания нам понадобится любая редакция VisualStudio 2010, а так же SQL Server не ниже 2005. По умолчанию в поставку BLToolkit как минимум входят драйвера для MS SQL 2005, 2008, Microsoft Access. Впрочем, все чудесно работает и с MS SQL Denali

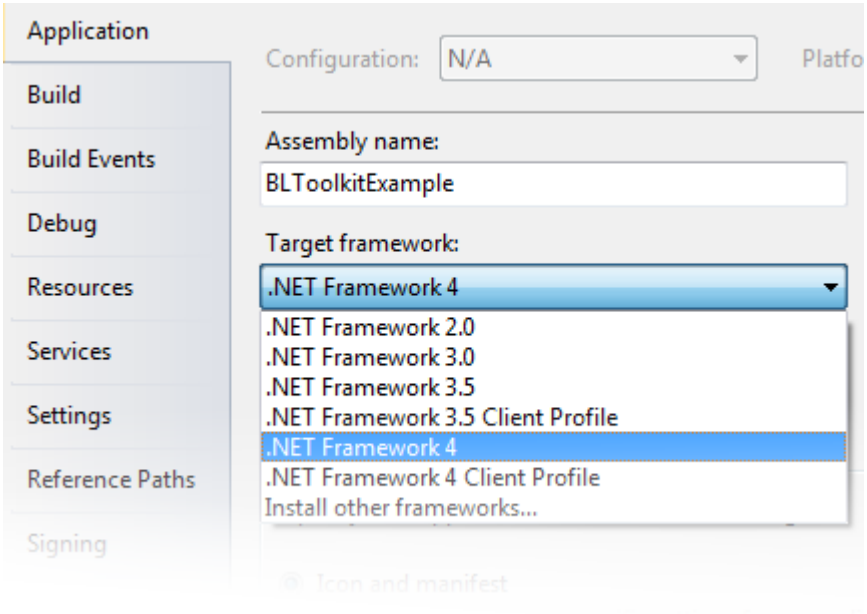
Скрипт для базы

```
create table Person(
        PersonId uniqueidentifier primary key default newid()
       ,Name nvarchar(100) not null
       ,Birth datetime
       ,Resident bit default 0
       ,Gender char default 'm'
       ,Weight int
       ,Height decimal (3,2)

)
go

insert into Person values(newid(), 'Adam', '01-01-1980', 0, 'm', 80, 1.80)
insert into Person values(newid(), 'Amy', '04-07-1986', 1, 'f', 52, 1.65)
```

Тестов на скорость не делаем, поэтому хватит и двух записей для разнообразия.

## Добавление BLToolkit

Основой для экспериментов будет служить консольное приложение. Создаем консольное приложение, обязательно указываем, что работать будем с полным четвертым фреймворком.

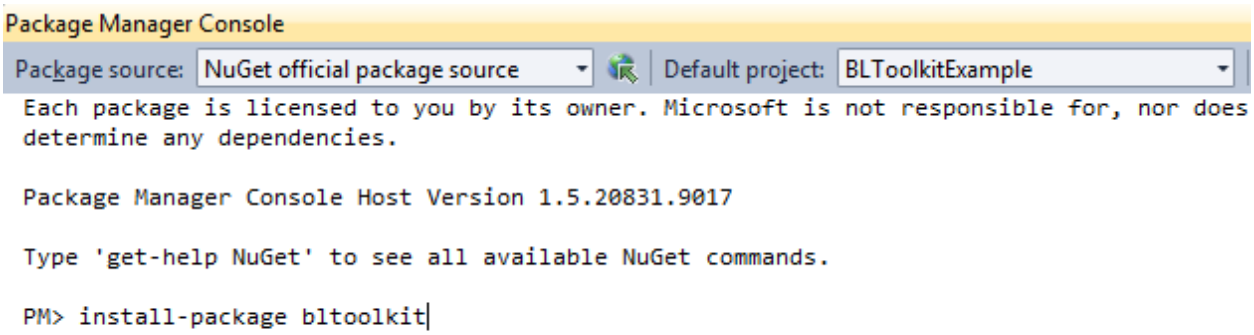



При создании по умолчанию проект будет создан для 4го клиентского профиля, что нам не подойдет совсем. Если не поменять фреймворк с ".Net Framework 4 Client Profile", на ".Net Framework 4", то во время компиляции получим такие сообщения:

```
The referenced assembly "BLToolkit.4" could not be resolved because it has a dependency on
"System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
The referenced assembly "BLToolkit.4" could not be resolved because it has a dependency on
"System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
The referenced assembly "BLToolkit.4" could not be resolved because it has a dependency on
"System.Design, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
```

После того, как сменили версию фреймворка, добавляем BLToolkit с помощью NuGet. Открываем консоль Package Manager Console (*Tools > Package Manager Console* ) и вводим там:

```
PM> install-package bltoolkit
```




В результате должен появиться текст:

```
PM> install-package bltoolkit
```

```
Successfully installed 'BLToolkit 4.1.6'.
Successfully added 'BLToolkit 4.1.6' to BLToolkitExample.
```

После добавления ссылки на BLToolkit собираем приложение и убеждаемся, что все компилируется.

Следующим шагом будет создание «бизнес-объекта». Сделаем его для начала максимально похожим на то, что содержится в базе данных.

```
public class Person {
    public Guid PersonId;
    public string Name;
    public DateTime Birth;
    public bool Resident;
    public string Gender;
    public int Weight;
    public decimal Height;
}
```

Это было не сложно, далее начнется основная магия.

## DataAccessor

Основным классом для работы с BLToolkit у нас будет DataAccessor. Это абстрактный класс и от него будем наследоваться для написания своего фасада доступа к данным. Данный класс управляет практически всеми аспектами и нюансами работы с базой данных, в нем определенно множество методов, которые можно переопределить для тонкой настройки работы с базой.

Начнем с создания класса, который будет наследован от DataAccessor и так же будет абстрактным.

```
using BLToolkit.DataAccess;

public abstract class PersonAccessor : DataAccessor {

}
```

Абстрактный класс используется из-за того, что генерация когда для доступа к базе и маппингу будет происходить в момент исполнения приложения.

Следующим шагом будет создание метода для получения данных из базы. Он так же будет абстрактным.

```
public abstract List<Person> GetAll();
```

Придумываем более-менее адекватное название методу, указываем какой тип данных будет возвращать метод. Естественно, что BLToolkit волшебным образом в данный момент не узнает откуда брать данные. Самый простой способ указать это – воспользоваться атрибутом SqlQuery и написать запрос.

```
[SqlQuery("SELECT * FROM Person")]
public abstract List<Person> GetAll();
```

Но и это еще не все, осталось сделать главное, сказать как соединятся с сервером, откуда будем брать данные. Это можно сделать как в файле конфигурации, так и в коде.

Конфигурация соединения с базой в конфигурации приложения (app.config):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add
        name             = "DemoConnection"
        connectionString = "Server=lethiathan\lcf11ctp3;Database=Test;Integrated
Security=true"
        providerName     = "System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Честно сказать я не люблю указывать в конфигурации настройки, поэтому быстро нашел способ как указать это в программе. Как я уже сказал, класс DataAccessor, позволяет тонко настроить многие параметры работы с базой. Можно задать и соединение с базой.

```
protected override DbManager CreateDbManager() {
    var cs = new SqlConnectionStringBuilder {
                                                InitialCatalog = "Test",
                                                IntegratedSecurity = true,
                                                DataSource = @"LETHIATHAN\LCF11CTP3"
                                            };

    return new DbManager(new SqlDataProvider(), cs.ConnectionString);
}
```

Остается теперь только получить экземпляр класса PersonAccessor в рабочем коде. Это делается несколько необычным способом, в состав BLToolkit входит класс для создания экземпляров классов типа нашего PersonAccessor:

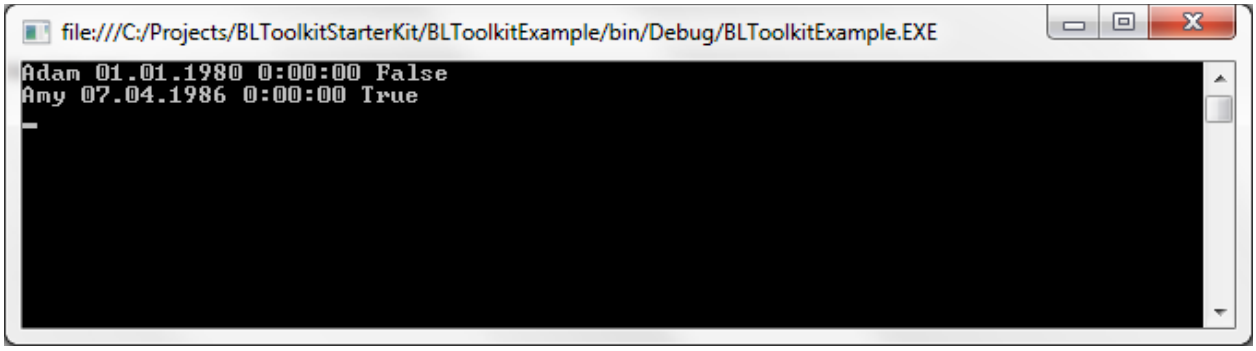
```
var pa = TypeAccessor<PersonAccessor>.CreateInstance();
```

С помощью данного способа вы получаете экземпляр класса PersonAccessor, но он уже не абстрактный. Я так думаю, что во время компиляций исходный абстрактный класс переименовывается и с помощью TypeAccessor на его основе строится новый класс в run-time режиме.

Все, после этого можно обращаться к методам нашего класса и работать с результатами. Итого:

```
private static void Main(string[] args) {
    var pa = TypeAccessor<PersonAccessor>.CreateInstance();

    var list = pa.GetAll();

    foreach (var p in list)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    Console.ReadLine();
}
```

Результат:




Далее можно экспериментировать с классом Person, добавлять новые поля, удалять, делать их свойствами и все равно класс будет заполняться данными из базы. Порядок полей в классе, естественно, значения не имеет. Советую попробовать поэкспериментировать в этом ключе. И да, заполняются только публичные поля класса. Если Name будет приватным полем или же иметь приватный set, то значение не будет восстановлено из базы.

## Сопоставление данных

В целом BLT неважно как получен набор данных, главная цель сопоставить его указанному классу. Т.е. в запросе можно указать представление, или же, как далее будет показано, использовать хранимые процедуры, которые возвращают значения.

В нашем случае можно написать представление на таблицу Person:

```
create view vPerson
as
SELECT PersonId
      ,Name
      ,Birth
      ,Resident
      ,Gender
      ,Weight
      ,Height
  FROM dbo.Person
```

Модифицировать запрос:

```
[SqlQuery("SELECT * FROM vPerson")]
public abstract List<Person> GetAll();
```

Всё будет работать точно так же как прежде.

## Параметры

При использовании атрибута SqlQuery можно передавать в запрос параметры. Что может быть весьма полезно в некоторых случаях.

Для построения запроса с параметрами можно использовать стандартный синтаксис MS SQL, причем имя параметра в запросе должно совпадать с именем параметра в методе.

```
[SqlQuery("SELECT * FROM vPerson where resident = @isResident")]
public abstract List<Person> GetAll(bool isResident);
```

## Хранимые процедуры

Для многих разработчиков является нормой сделать всю работу с базой на хранимых процедурах, лет десять назад это было очень популярно. Впрочем, еще много ситуации когда без хранимых процедур будет тяжело обходиться, так что посмотрим, как же работать с ними с помощью BLToolkit.

Начнем с написания хранимой процедуры. Пусть нам требуется выборка людей старше определенного возраста.

```
create procedure Person_GetOlderThan(
             @Age int
)
as
set nocount on;

select PersonId
      ,Name
      ,Birth
      ,Resident
      ,Gender
      ,Weight
      ,Height
  from dbo.Person
 where datediff(yy,Birth, getdate()) > @Age
```

Прогоняем процедуру, убеждаемся, что все работает как надо и переходим к написанию C# кода. Для этого в нашем абстрактном классе PersonAccessor объявляем новый метод, который будет принимать целочисленное значение возраста в качестве параметра и возвращать список людей старше указанного значения.

```
public abstract List<Person> GetOlderThan(int age);
```

Всё!

После этого в основной программе можно вызвать этот метод и получить данные.

```
private static void Main(string[] args) {
    var pa = TypeAccessor<PersonAccessor>.CreateInstance();

    var list = pa.GetOlderThan(30);

    foreach (var p in list)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    Console.ReadLine();
}
```

Все дело в принятых соглашениях по названию процедур. Наверно вас сразу насторожило название процедуры в базе и название метода в классе.

По умолчанию, BLToolkit формирует имена процедур из имени шаблонного класса и названия метода: **%имя_шаблона%_%имя метода%**. Однако такое поведение можно переопределить в классе доступа к базе.

Если бы у нас было корпоративное правило, по которому все процедуры должны называться, используя префикс “**sp_**”, это можно было бы легко реализовать. Для этого в классе PersonAccessor надо переопределить метод **GetDefaultSpName**.

```
protected override string GetDefaultSpName(string typeName, string actionName) {
    return typeName == null
               ? actionName
               : string.Format("sp_{0}_{1}", typeName, actionName);
}
```

Для чистоты эксперимента проверяем, что в базе не осталось исходной процедуры, а только sp_Person_GetOlderThan. Запускаем приложение и видим что все работает отлично.

В более хардкорных случаях, когда в вашей организации процедуры принято кодировать и настоящее имя процедуры выглядит примерно так: **sp_prsn_get_oldr** – будет очень не сладко использовать такое имя в коде.

В этом случае можно использовать атрибут **SprocName**.

```
[SprocName("sp_prsn_get_oldr")]
public abstract List<Person> GetOlderThan(int age);
```

Несмотря на перегруженный метод GetDefaultSpName, код отработает как ожидалось, так как применение атрибутов имеет более высокий приоритет, что логично.

## Параметры

BLToolkit сопоставляет параметры метода и процедуры по именам, поэтому не имеет значения в каком порядке вы объявили переменные. Допустим, процедура принимает 3 параметра

```
create procedure Person_GetOlderThan(
             @Age int,
             @SomeOtherParam nvarchar(100),
             @Weight int
)
as
...
```

Вызов процедуры пройдет нормально, если сигнатура метода будет выглядеть как:

```
public abstract List<Person> GetOlderThan(string someOtherParam, int age, int weight);
```

Если же и имена параметров никак вас не устраивают, то можно написать код вручную, который по идее генерируется на лету во время исполнения.

```
public List<Person> GetOlderThan(string text, int age, int weight) {
    using (var db = GetDbManager()) {
        return db
            .SetSpCommand("Person_GetOlderThan",
                          db.Parameter("@someOtherParam", text),
                          db.Parameter("@age", age),
                          db.Parameter("@weight", weight))
            .ExecuteList<Person>();
    }
}
```

В данном случае уже не имеет значения как назван метод, имя процедуры, параметров и прочего указывается явно. Думаю, что код в данном методе опысывает себя сам и в дополнительных пояснениях не нуждается.

BLToolkit позволяет проделывать еще множество фокусов с процедурами, но описанного должно хватить для большинства ситуаций в реальной жизни.

## Linq

Согласитесь, что писать запросы перед методом не очень хорошо, тем более, когда есть еще дополнительные условия для фильтрации. Лучше всего обратиться к Linq, раз уж существует такая возможность.

Для начала работы с Linq и указанию BLT на то, какие таблицы/представления будут участвовать в запросах, надо создать новый класс и наследовать его от **DbManager**.

```
public class BltLinq : DbManager {}
```

Далее надо создать конструктор, который либо будет указывать на строку соединения в конфигурации, либо задавать ее самостоятельно (всего конструкторов 5+), например, так:

```
public class BltLinq : DbManager {
    public BltLinq() : base(new SqlDataProvider(), @"Data
Source=LETHIATHAN\LCF11CTP3;Initial Catalog=Test;Integrated Security=True") {}
}
```

После этого можно создать свойство, которое будет возвращать тип **Table**, который реализует интерфейс IQueryable.

```
public class BltLinq : DbManager {
    public BltLinq() : base(new SqlDataProvider(), @"Data
Source=LETHIATHAN\LCF11CTP3;Initial Catalog=Test;Integrated Security=True") {}

    public Table<Person> Persons {
        get { return GetTable<Person>(); }
    }
}
```

После этих несложных манипуляций можно писать запросы к таблице Person. Например:

```
using (var db = new BltLinq()) {
    var query = db.Persons;

    foreach (var employee in query) {
        Console.WriteLine("{0} {1}", employee.Name, employee.Birth);
    }
}
```

Или

```
using (var db = new BltLinq()) {
    var query = from person in db.Persons
                where person.Weight < 55
                select person;
```

```
    foreach (var employee in query) {
        Console.WriteLine("{0} {1}", employee.Name, employee.Birth);
    }
}
```

Согласитесь, что это уже лучше, можно добавлять условия и изменять переменную типа IQueryable в процессе выполнения программы добавляя новые условия фильтрации, если нужно.

Сгенерированный запрос в процессе выполнения второго примера:

```
--  Sql MsSql2005
SELECT
       [person].[PersonId],
       [person].[Name],
       [person].[Birth],
       [person].[Resident],
       [person].[Gender],
       [person].[Weight],
       [person].[Height]
FROM
       [Person] [person]
WHERE
       [person].[Weight] < 55
```

Внимательный читатель наверняка уже задался вопросом, а как же обращаться к представлению или таблице, если ее имя не совпадает с именем класса, на который та проецируется? В этом случае используется атрибут **TableName**.

```
[TableName("vPerson")]
public class Person {
  ...
}
```

## Связанные таблицы

По умолчанию BLT не строит ассоциативных связей между таблицами по вторичным ключам, это можно сделать декларативно, указав связи между таблицами. Относительно CQRS данный функционал не должен применяться по идее, однако это может оказаться полезным. Выглядит это не очень красиво в коде, ну да ничего не поделаешь с этим.

Итак, дополним базу данных адресами наших знакомцев.

```
create table Address(
        AddressId uniqueidentifier primary key default newid()
       ,Country   nvarchar(100)
       ,Region    nvarchar(100)
       ,City      nvarchar(100)
       ,Street    nvarchar(100)
       ,Residence nvarchar(100)
)

alter table Person add AddressId uniqueidentifier
alter table Person add constraint fk_address2person foreign key (AddressId) references
Address(AddressId)
go


insert into Address values(newid(), 'Russia', 'N.Novgorod', 'N.Novgorod','Minina', '1A')
```

```
insert into Address values(newid(), 'Russia', 'Msk', 'Msk', 'Lenina', '40 - 123')

update Person set Person.AddressId = Address.AddressId
 from Address where city = 'Msk' and Name = 'Adam'

update Person set Person.AddressId = Address.AddressId
 from Address where city <> 'Msk' and Name = 'Amy'
```

После этого можно продолжить работу с приложением. Так как у нас появилась новая сущность в базе данных, то нам потребуется новый класс, в который будем записывать данные из базы. В целом класс будет очень похож на то, что мы делали для таблицы Person.

```
public class Address {
    public Guid AddressId;
    public string Country;
    public string Region;
    public string City;
    public string Street;
    public string Residence;
}
```

Далее мы хотим получить связь между классом Person и классом Address, для этого в класс Person добавляем поле типа Address, которое будет в дальнейшем выступать связью для построения linq запросов.

```
public class Person {
    public Guid PersonId;
    public string Name;
    public DateTime Birth;
    public bool Resident;
    public string Gender;
    public int Weight;
    public decimal Height;

    public Address Address;
}
```

Для установления связи, необходимо воспользоваться атрибутом **Association**. В параметрах указываются ключи связи и индикатор, возможно ли значение null у связываемой таблицы.

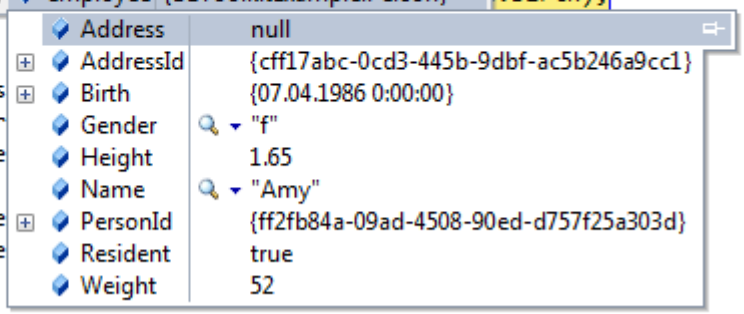
```
public class Person {
    public Guid PersonId;
    public string Name;
    public DateTime Birth;
    public bool Resident;
    public string Gender;
    public int Weight;
    public decimal Height;

    public Guid AddressId;

    [Association(ThisKey="AddressId", OtherKey="AddressId")]
    public Address Address;
}
```

Обязательно при этом **добавить поле ключа связи – AddressId**, иначе ничего работать не будет.

Далее в класс **BltLinq** добавляем свойство для доступа к таблице Address, по образу и подобию того как мы сделали ранее для Person.

После чего можно использовать свойство в построении запросов. Но и тут есть нюансы. Например, возьмем старый код, который уже использовали для получения списка людей с весом меньше 55, и посмотрим в дебаге, заполнилось ли поле Address.

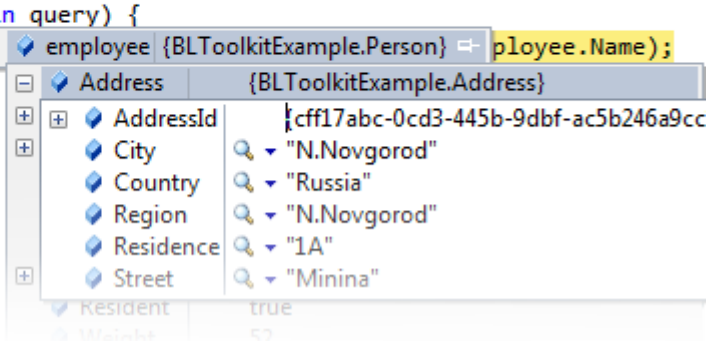
```
using (var db = new BltLinq()) {
    var query = from person in db.Persons
                where person.Weight < 55
                select person;

    foreach (var employee in query) {
        Console.WriteLi  employee {BLToolkitExample.Person}  .Birth);
    }

    var query1 =from pers
                where per
                select pe

    foreach (var employee
        Console.WriteLine
    }
}
```




Члены класса, к которым применена ассоциация - не заполняются автоматически при чтении объекта из базы данных целиком. Однако эти свойства можно заполнить вручную. Для этого лучше всего сделать конструктор копирования, и не забыть оставить пустой конструктор, который будет использован BLT по умолчанию. После этого запрос отработает как ожидалось.

```
using (var db = new BltLinq()) {
    var query = from person in db.Persons
                where person.Weight < 55
                select new Person(person) {
                                              Address = person.Address
                                          };

    foreach (var employee in query) {
        Console.WriteLine("{0} {1}", employee.Address.City, employee.Name);
    }
}
```

```
using (var db = new BltLinq()) {
    var query = from person in db.Persons
                where person.Weight < 55
                select new Person(person) {
                                                Address = person.Address
                                            };

    foreach (var employee in query) {
        Console.WriteLine
    }
}

Console.ReadLine();
```




## Заключение

Это можно считать вводным курсом в BLToolkit. Фреймворк предоставляет еще множество настроек и возможностей по оптимизации запросов и получения именно того, что вам требуется. Так же поддерживается запись и обновление строк в таблицах. Но об этом как-нибудь в другой раз, когда придет необходимость в этом.

По BLToolkit много материала в сети, так как это уже не новый фреймворк с большой армией поклонников.

В целом мне понравилось работать с BLToolkit, приятное впечатление оставляет фреймворк. Буду использовать.

Hard'n'heavy!

Violet Tape