## Урок 3. Работа с БД

Цель урока: Изучить основные принципы работы с базой данных. Краткое описание реляционной модели баз данных. Работа с базой данных (создание таблиц, связей в VS 2012 [не забыть показать где стоит галочка]). Команды INSERT, UPDATE, DELETE, SELECT. Использование LinqToSql и Linq. Создание репозитария, IRepository, SqlRepository. EntityFramework и CodeFirst, создание тестовых данных, инициализация БД.

### Что такое БД

Реляционная база данных — [база данных](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), основанная на [реляционной модели данных](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). Реляционность – это отношения (связи) от англ. relation.

#### Таблицы

Это таблица:

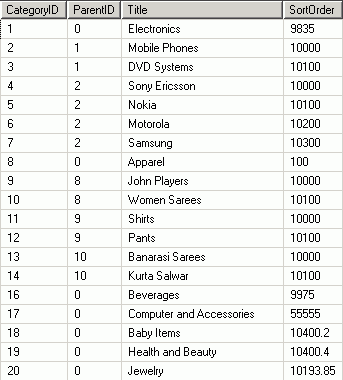


Таблица состоит из столбцов и строк. Столбцы имеют свойства – имя, тип данных.

Таблицы обладать следующими свойствами:

* у таблицы есть имя
* нет двух одинаковых строк
* столбцы имеют разные наименования (нет двух одинаковых столбцов)
* порядок строк в таблице произвольный (т.е. не надо учитывать порядок строк, если не задана сортировка)

Структуру таблицы можно записать в таком виде:

* Имя столбца
* Тип данных для этого столбца

#### Связи

Между таблицами существуют связи (relation). Для установки связи необходимо иметь следующее:

* Первичный ключ – это набор столбцов (атрибутов) таблицы, которые однозначно определяют уникальность строки. Обычно это одно поле, называется ID. Оно является автоикрементным, т.е. при попытке добавления записи там автоматически вставляется 1, 2, 3, 4… n+1, где n – это значение последнего добавленного ID.
* Внешний ключ – это набор столбцов (атрибутов) таблицы, которые однозначно определяют уникальность строки в другой таблице. Опять же это обычно одно поле, названное [Имя таблицы]ID. Но не является автоинкрементным.
* Прописана связь между первичным ключом и внешним ключом.

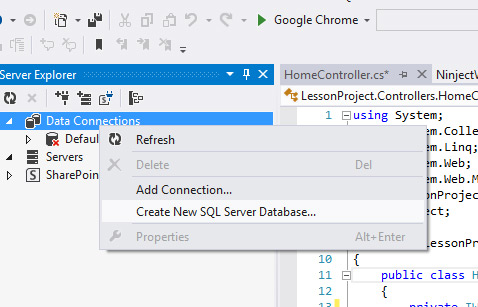
Связи бывают 3 типов:

* Один-к-одному. Т.е. одной строки в таблице соответствует одна строка в другой таблице. Это редко используется, но используется. Например, в одной таблице данные о пользователе, а в другой таблице дополнительные данные о том же пользователе. Такой вариант необходим, чтобы манипулировать по необходимости меньшим количеством данных.
* Один-ко-многим. Одной строке в таблице A соответствует одна или несколько строк в таблице B. Но одной строке в таблице B соответствует только одна строка в таблице A. В этом случае в таблице B существует внешний ключ, который однозначно определяет запись в таблице A.
* Многие-ко-многим. Одной строке в таблице А соответствует одна или несколько строк в таблице В, что истинно и в обратном случае. В данном случае создается дополнительная таблица со своим первичным ключом, и двумя внешними ключами к таблице A и B.

Сейчас как раз разберемся как это делать.

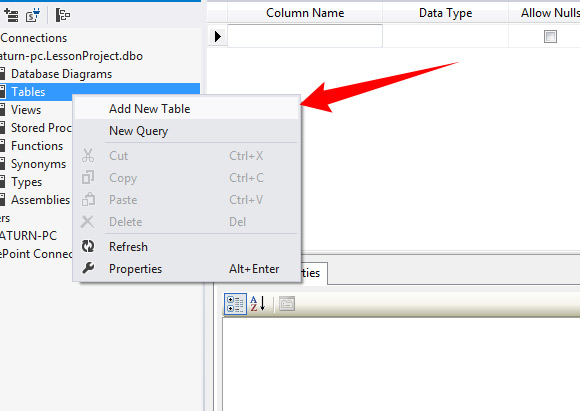
### Создание простой схемы в БД

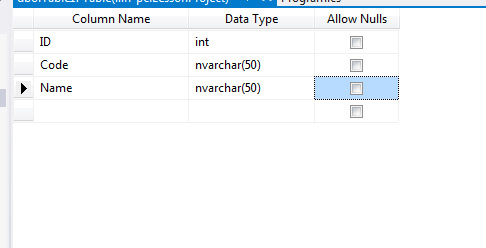
Создадим БД в VS 2012:



Назовем LessonProject, и добавим 3 таблицы Role User и UserRole.

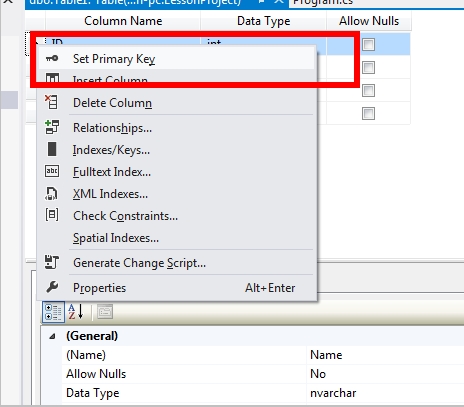
Создадим таблицу Role:



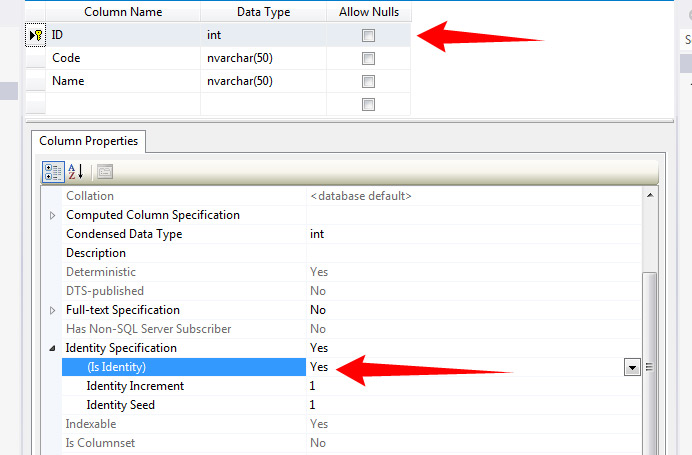


Для строковых значений используем тип nvarchar(n), где n – максимальная длина строки, обычно используется от 50 до 500. Для больших текстовых строк используется nvarchar(MAX).

Устанавливаем первичный ключ:



Задаем для ID автоинкремент:



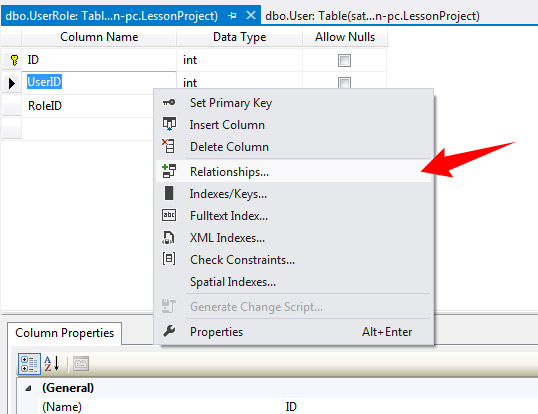
Подобным образом создаем таблицу User:

|  |  |
| --- | --- |
| **Поле** | **Тип поля** |
| ID | int |
| Email | nvarchar(150) |
| Password | nvarchar(50) |
| AddedDate | datetime |
| ActivatedDate | Datetime (null) |
| ActivatedLink | nvarchar(50) |
| LastVisitDate | Datetime (null) |
| AvatarPath | nvarchar(150) (null) |

Создаем таблицу UserRole:

|  |  |
| --- | --- |
| **Поле** | **Тип поля** |
| ID | int |
| UserID | int |
| RoleID | int |

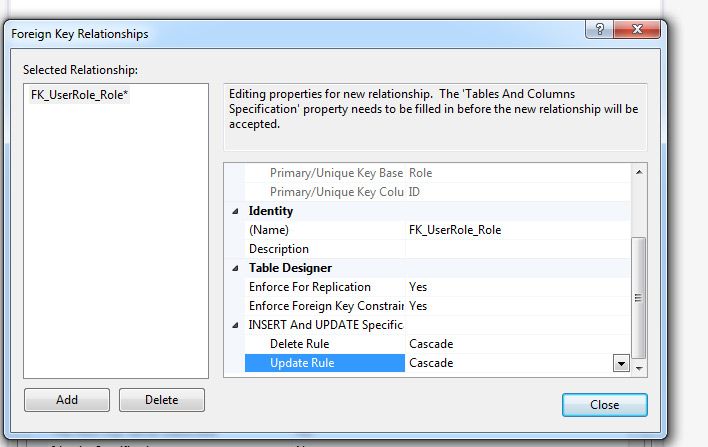
Добавим связи:



Добавляем новую связь, нажав Add, добавление связей происходит в таблице где находятся внешние ключи. Раскрываем Tables and Columns вкладку и выставляем таблицу с первичным ключом, и выбираем внешний ключ в текущей таблице UserRole.



В свойствах INSERT And UPDATE Specification выставляем On Update/On Delete свойства Cascade:



Это необходимо для того, что при изменении/удалении столбца из таблицы Role все связанные с этой строкой строки таблицы UserRole должны быть изменены или удалены. Тут касается больше удаления, так как первичный ключ в таблице обычно никогда не изменяют.

Аналогичную связь мы устанавливаем с таблицей User.

Таким образом, таблицы Role и User имеют отношения многие ко многим через таблиц UserRole. Т.е. у одного пользователя может быть больше одной роли, и одна и та же роль может быть у нескольких пользователей.

### Использование команд SELECT , INSERT, UPDATE, DELETE.

В реляционных базах данных используется язык запросов SQL.

Есть 4 основные команды для манипулирования данными, SELECT, INSERT, UPDATE, DELETE

SELECT – выбрать данные и таблицы(таблиц).

Пример:

SELECT \* FROM User

INSERT - Добавление строк в таблицу

Пример:

INSERT INTO Role (Code, Name)  
VALUES (“admin”, “Администратор”)

UPDATE – изменение значений в таблице

Пример:

UPDATE User  
SET Password=”password1”   
WHERE ID=1

DELETE – удаление строк из таблице

Пример:

DELETE FROM User  
WHERE ID =1

*Примечание: Подробнее можно изучить SQL по ссылкам:*

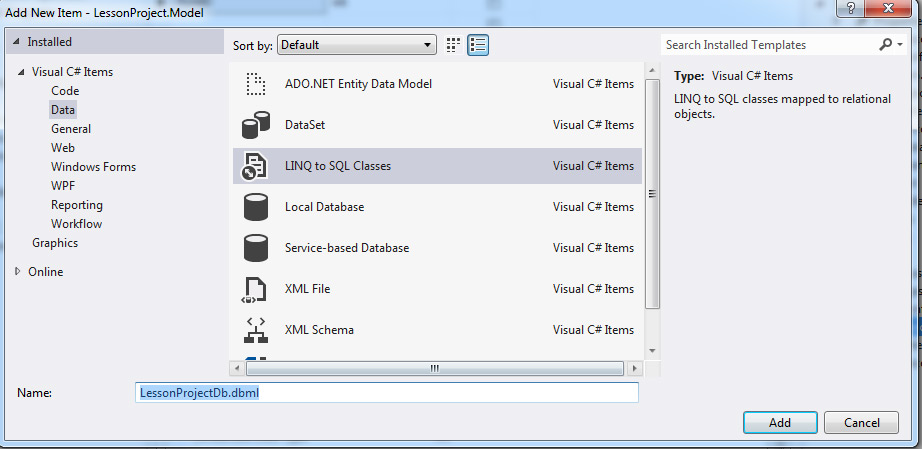
<http://www.w3schools.com/sql/>

<http://codingcraft.ru/sql_queries.php>

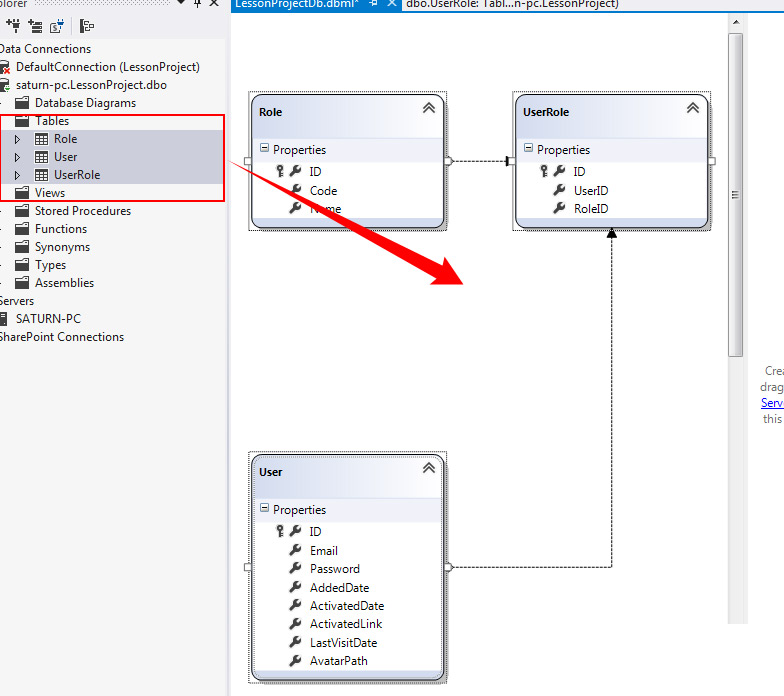
### LinqToSQL и Linq.

Создаем проект LessonProject.Model для работы с БД типа ClassLibrary.

Добавляем LINQ to SQL Classes тип, называем LessonProejctDb.dbml



Открываем объект, выделяем все таблицы и мышкой переносим на холст:



Собственно, с помощью таких простых действий мы получаем:

* классы готовые к использованию в работе с БД
* визуальное отображение таблиц и связей

Добавим несколько данных в таблицу Role и User:

|  |  |  |
| --- | --- | --- |
| 1 | admin | Админ |
| 2 | customer | Пользователь |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | chernikov@gmail.com | 123456 | 1/1/2012 12:00:00 AM | NULL | 123456 | NULL | NULL |
| 3 | chernikov2@gmail.com | 123456 | 1/1/2012 12:00:00 AM | NULL | 123456 | NULL | NULL |

И UserRole

|  |  |  |
| --- | --- | --- |
| **ID** | **UserID** | **RoleID** |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 2 |

Создадим консольный проект Lesson3 и поключим LessonProject.Model. Добавим сборку System.Configuration и System.Data.Linq. Проинициализируем context и выведем данные о ролях:

class Program

{

static void Main(string[] args)

{

var context = new LessonProjectDbDataContext(ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString);

var roles = context.Roles.ToList();

foreach (var role in roles)

{

Console.WriteLine("{0} {1} {2}", role.ID, role.Code, role.Name);

}

Console.ReadLine();

}

}

Для добавления строки в Role делаем так:

var newRole = new Role

{

Code = "manager",

Name = "Менеджер"

};

context.Roles.InsertOnSubmit(newRole);

context.Roles.Context.SubmitChanges();

Для удаления строки в Role делаем так:

var role = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault();

if (role != null)

{

context.Roles.DeleteOnSubmit(role);

context.Roles.Context.SubmitChanges();

}

Для изменения данных делаем так:

var role = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault();

if (role != null)

{

role.Name = "Манагер";

context.Roles.Context.SubmitChanges();

}

Для манипуляции данных используется язык запросов Linq. Мы рассмотрим только некоторые основные функции Linq. Linq применяется для типов реализующий интерфейс IQueryable<>

**.Where()** – основная функция фильтрации. Возвращает тип IQueryable. Условие внутри должно возвращать булево значение (bool).

var roles = context.Roles.Where(p => p.Name == "Менеджер")

**.FirstOrDefault()** - .First(), .Single(), .SingleOrDefault() – получают первую или единственную запись. Если записи нет, то FirstOrDefault() или SingleOrDefault() возвращают null (на самом деле значение по умолчанию этого типа).

var roles = context.Roles.Where(p => p.Name == "Менеджер").FirstOrDefault() – получаем первую (или не получаем) роль названную «Менеджер».

**.Take()** – выбирает N первых записей

var roles = context.Roles.Where(p => p.Name == "Менеджер").Take(4) – выберет 4 первые записи

**.Skip()** – пропускает выбор N первых записей

var roles = context.Roles.Where(p => p.Name == "Менеджер"). Skip(2).Take(3) – пропустит первые 2 и выберет 3 следующие записи

**.OrderBy()** – сортирует по возрастанию. А также OrderByDescending(), ThenBy(), ThenByDescending(). Лябда выражение должно возвращать int по которому и будет происходить сортировка.

var roles = context.Roles.Where(p => p.Name == "Менеджер").OrderBy(p => p.ID) – сортирует по порядку

**.Count()** – получает количество записей

var rolesCount = context.Roles.Where(p => p.Name == "Менеджер").Count() – количество записей

**.Any()** – существует одна или больше записей по данному условию

var rolesExist = context.Roles.Where(p => p.Name == "Менеджер").Any() – есть ли запись такая

**.Select()** – возвращает IQueryable заданного типа, может быть даже динамический:

var otherRole = context.Roles.Where(p => p.Name == "Менеджер").Select(p => new {  
ID = p.ID, Kod = p.Code}) – получаем динамический тип сформированный на основе Role.

**.SelectMany()** – возвращает объединение всех IQueryable типов внутри выборки:

var otherRole = context.Roles.Where(p => p.Name == "Менеджер").SelectMany(p => p.UserRoles) – получаем все UserRole из роли названной «Менеджер»

**.Distinct()** – удаляет дубликаты

var managers = context.Roles.Where(p => p.Name == "Менеджер").SelectMany(p => p.UserRoles).Select(p => p.User).Distinct() – все пользователи с ролью названной «Менеджер»

*Примечание: First(), FirstOrDefault(), Single(), SingleOrDefault(), Any(), Count() – могут применять параметр соответствующий Where() тем самым можно сокращать запись:*

*var roles = context.Roles.FirstOrDefault(p => p.Name == "Менеджер")*

*Примечание, больше примеров и вариантов использования linq можете найти:*<http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>

### Создание репозитория IRepository , SqlRepository.

Собственно с БД мы уже можем работать, только теперь надо отделить модель данных от конкретной реализации, т.е. наши контроллеры про context и System.Data.Linq вообще не должны ничего знать.

Для этого создадим интерфейс IRepository где будет дан доступ ко данным, а также выведены методы для создания, изменения, удаления этих данных.

public interface IRepository

{

IQueryable<Role> Roles { get; }

bool CreateRole(Role instance);

bool UpdateRole(Role instance);

bool RemoveRole(int idRole);

…

}

И реализацию назовем SqlRepository. Так как мы с данным контекстом SqlRepository не хотим особо связывать, то добавим Ninject модуль в проект LessonProject.Model :

Install-Package Ninject

Получим класс SqlRepository:

public class SqlRepository : IRepository

{

[Inject]

public LessonProjectDbDataContext Db { get; set; }

public IQueryable<Role> Roles

{

get { throw new NotImplementedException(); }

}

public bool CreateRole(Role instance)

{

throw new NotImplementedException();

}

public bool UpdateRole(Role instance)

{

throw new NotImplementedException();

}

public bool RemoveRole(int idRole)

{

throw new NotImplementedException();

}

}

Прежде, чем реализовать доступ ко всем таблицам, создание, удаление и изменение подумаем, что файл этот будет выглядеть весьма монструозно, и тяжело будет физически управлять таким кодом. Так что сделаем отдельную папку SqlRepository и SqlRepository класс сделаем partial, а в папке создадим реализации разбитые по каждой таблице. Назовем файл Role и реализуем:

public partial class SqlRepository

{

public IQueryable<Role> Roles

{

get

{

return Db.Roles;

}

}

public bool CreateRole(Role instance)

{

if (instance.ID == 0)

{

Db.Roles.InsertOnSubmit(instance);

Db.Roles.Context.SubmitChanges();

return true;

}

return false;

}

public bool RemoveRole(int idRole)

{

Role instance = Db.Roles.FirstOrDefault(p => p.ID == idRole);

if (instance != null)

{

Db.Roles.DeleteOnSubmit(instance);

Db.Roles.Context.SubmitChanges();

return true;

}

return false;

}

}

Ок. Небольшой проект содержит от 10 до 40 таблиц, большой проект от 40, и собственно всё это хотелось бы хоть как-то автоматизировать. Создадим несколько сниппетов, для IRepository, для SqlRepository. Сниппеты – это уже готовые шаблоны кода, которые вызываются с помощью intelliSence, и позволяют быстро создавать код.

### Сниппеты

Для IRepository таблиц, назовем table.snippet:

<CodeSnippets

xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<CodeSnippet Format="1.0.0" >

<Header>

<Title>

Table

</Title>

<Shortcut>Table</Shortcut>

</Header>

<Snippet>

<Declarations>

<Literal>

<ID>Table</ID>

<ToolTip>Table name for create.</ToolTip>

<Default>Table</Default>

</Literal>

</Declarations>

<Code Language="CSharp">

<![CDATA[

#region $Table$

IQueryable<$Table$> $Table$s { get; }

bool Create$Table$($Table$ instance);

bool Update$Table$($Table$ instance);

bool Remove$Table$(int id$Table$);

#endregion

]]>

</Code>

</Snippet>

</CodeSnippet>

</CodeSnippets>

Для SqlRepository создадим сниппет sqlTable.snippet:

<CodeSnippets

xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<CodeSnippet Format="1.0.0" >

<Header>

<Title>

Sql repository

</Title>

<Shortcut>sqltable</Shortcut>

</Header>

<Snippet>

<Declarations>

<Literal>

<ID>Table</ID>

<ToolTip>Table name for create.</ToolTip>

<Default>Table</Default>

</Literal>

</Declarations>

<Code Language="CSharp">

<![CDATA[

public IQueryable<$Table$> $Table$s

{

get

{

return Db.$Table$s;

}

}

public bool Create$Table$($Table$ instance)

{

if (instance.ID == 0)

{

Db.$Table$s.InsertOnSubmit(instance);

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

public bool Update$Table$($Table$ instance)

{

$Table$ cache = Db.$Table$s.Where(p => p.ID == instance.ID).FirstOrDefault();

if (cache != null)

{

//TODO : Update fields for $Table$

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

public bool Remove$Table$(int id$Table$)

{

$Table$ instance = Db.$Table$s.Where(p => p.ID == id$Table$).FirstOrDefault();

if (instance != null)

{

Db.$Table$s.DeleteOnSubmit(instance);

Db.$Table$s.Context.SubmitChanges();

return true;

}

return false;

}

]]>

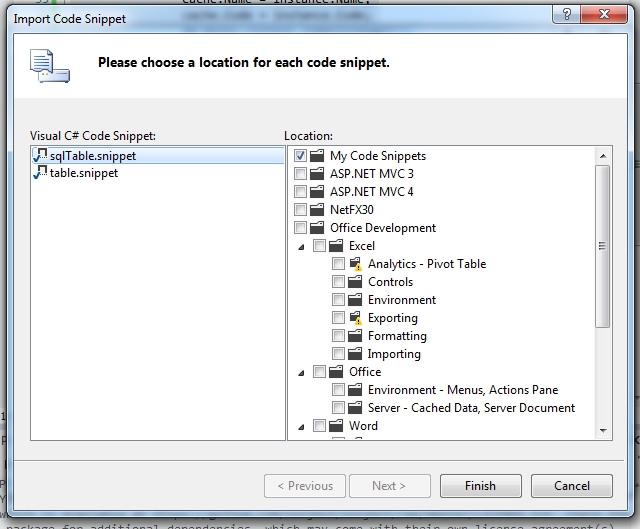
</Code>

</Snippet>

</CodeSnippet>

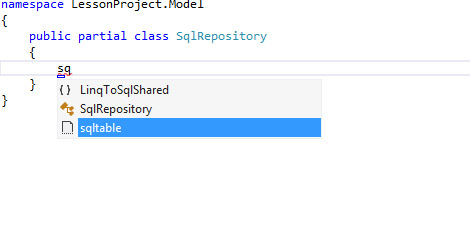
</CodeSnippets>

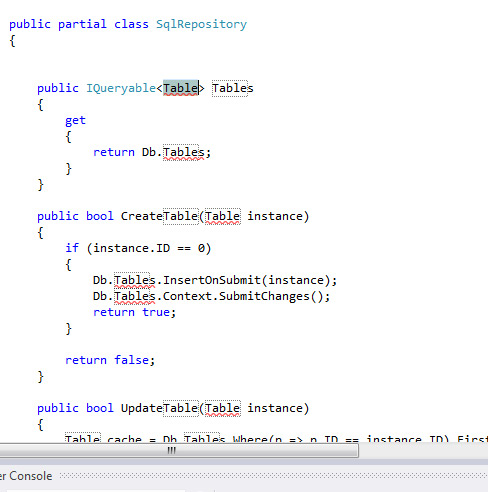
Для того чтобы добавить code-snippet откроем TOOLS -> Code Snippet Manager… (Ctrl-K, B). В открывшемся окне нажимаем Import и импортируем оба сниппета в My Code snippet:



Finish, OK.

Используем для таблиц User и UserRole.





Осталось прописать только поля для Update[имя таблицы], но это уже меньше работы.

### Proxy

Как видно классы, которые мы используем являются partial, поэтому их можно дополнить, создадим подобно SqlRepository папку Proxy, где будем размещать partial классы. Например для класса User создадим метод, который автоматически генерирует строку требуемую для активации пользователя:

public partial class User

{

public static string GetActivateUrl()

{

return Guid.NewGuid().ToString("N");

}

}

Используем это:

public bool CreateUser(User instance)

{

if (instance.ID == 0)

{

**instance.AddedDate = DateTime.Now;**

**instance.ActivatedLink = User.GetActivateUrl();**

Db.Users.InsertOnSubmit(instance);

Db.Users.Context.SubmitChanges();

return true;

}

return false;

}

### Использование в asp.net mvc

Добавим строку доступа к БД в web.Config:

<connectionStrings>

<add name="ConnectionString" connectionString="Data Source=SATURN-PC;Initial Catalog=LessonProject;Integrated Security=True;Pooling=False" providerName="System.Data.SqlClient" />

</connectionStrings>

Проинициализируем работу с БД в Ninject:

private static void RegisterServices(IKernel kernel)

{

kernel.Bind<LessonProjectDbDataContext>().ToMethod(c => new LessonProjectDbDataContext(ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString));

kernel.Bind<IRepository>().To<SqlRepository>().InRequestScope();

}

Как видно тут мы используем область InRequestScope(), т.е. каждый запрос будет использовать отдельный объект SqlRepository. Это позволит избежать коллизий при исполнении, так как одновременно к БД может происходить много запросов.

Объявляем IRepository в контроллере:

public class HomeController : Controller

{

[Inject]

public IRepository Repository { get; set; }

public ActionResult Index()

{

var roles = Repository.Roles.ToList();

return View(roles);

}

}

И обновляем View:

@model IList<LessonProject.Model.Role>

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>LessonProject</h2>

<p>

@foreach (var role in Model)

{

<div class="item">

<span class="id">

@role.ID

</span>

<span class="name">

@role.Name

</span>

<span class="Code">

@role.Code

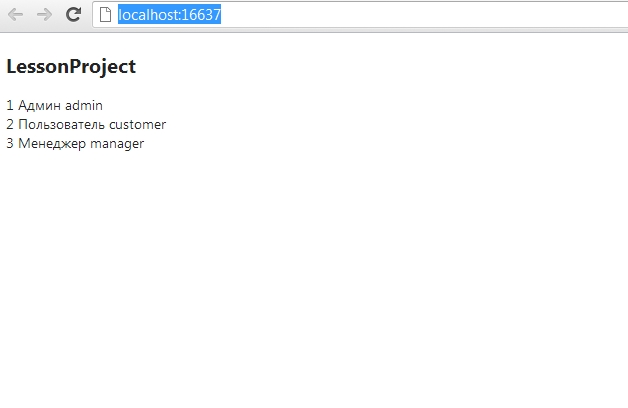
</span>

</div>

}

</p>

Получаем хороший результат:



### CodeFirst

*Создание объектов, инициализация в структуру БД. Создание тестовых данных.*

### Структура модели и использование контеста.

*Команды DbContext, Использование IRepository<TEntity>, EntityRepository.*