МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П.,
БАХРОМОВ С.А., ЭЛОВ Б.Б.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ ENTITY FRAMEWORK 6

УЧЕБНОЕ ПОСОБИЕ

МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П.,
БАХРОМОВ С.А., ЭЛОВ Б.Б.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ ENTITY FRAMEWORK 6

УЧЕБНОЕ ПОСОБИЕ

МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕГО СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ УЗБЕКИСТАНА ИМЕНИ МИРЗО УЛУГБЕКА

Мухамедиева Д.Т., Варламова Л.П.,

Бахромов С.А., Элов Б.Б.

Учебное пособие

по курсу

«ПРОГРАММНАЯ ИНЖЕНЕРИЯ»

Entity framework 6

Ташкент-2023 г.

Учебное пособие по курсу “Программная инженерия” составлено на основе образцовой и рабочей программ для студентов специальности 5330100 – “Информационные системы, математика и программное обеспечение” с целью обучения использованию языков программирования, разработке программного обеспечения и систем управления базами данных.

“Dasturiy ta’minot injiniringi” kursi uchun o‘quv qo‘llanma 5330100- “Axborot tizimlari, matematika va dasturiy ta’minot” mutaxassisligi talabalari uchun dasturlash tillaridan foydalanish, dasturiy ta’minot ishlab chiqish va ma’lumotlar bazasini boshqarish tizimlarini o‘rgatish maqsadida tuzilgan. Ushbu qo‘llanmaning 2-qismi Entity Framework 6 da dasturlashni o‘z ichiga oladi.

Авторы:

Д.Т.Мухамедиева - профессор Национального исследовательского университета «Ташкентский институт инженеров ирригации и механизации сельского хозяйства», д.т.н.

Л.П.Варламова - профессор кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, д.т.н.

С.А.Бахромов - доцент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, к.т.н.

Б.Б.Элов - доцент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, к.т.н.

Рецензенты:

Матякубов А.С. Зав. кафедрой “Прикладная математика и компьютерный анализ” Национального университета Узбекистана

Якубов М.С. Профессор кафедры “Информационные технологии” Ташкентского университета информационных технологий

Учебное пособие рекомендовано к изданию на основании приказа Министерства высшего и среднего специального образования Республики Узбекистан от 09 сентября 2022 года №302. Регистрационный номер 302-0397.

Содержание

1. Введение в Entity Framework

Что такое Entity Framework

Entity Framework представляет специальную объектно-ориентированную технологию на базе фреймворка **.NET** для работы с данными. Если традиционные средства **ADO.NET** позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то **Entity Framework** представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища.

Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает **Entity Framework**, мы уже работает с объектами.

Первая версия **Entity Framework - 1.0** вышла еще в **2008** году и представляла очень ограниченную функциональность, базовую поддержку **ORM** (**object-relational mapping** - отображения данных на реальные объекты) и один единственный подход к взаимодействию с **БД** - **Database First**.

С выходом версии **4.0** в **2010** году многое изменилось - с этого времени **Entity Framework** стал рекомендуемой технологией для доступа к данным, а в сам фреймворк были введены новые возможности взаимодействия с **БД** - подходы **Model First** и **Code First**.

Дополнительные улучшения функционала последовали с выходом версии **5.0** в **2012** году. И наконец, в **2013** году был выпущен **Entity Framework 6.0**, обладающий возможностью асинхронного доступа к данным.

Центральной концепцией **Entity Framework** является понятие **сущности** или **entity**. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами.

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает человека, то мы можем выделить такие свойства, как **имя, фамилия, рост, возраст, вес**.

Свойства необязательно представляют простые данные типа **int**, но и могут представлять более комплексные структуры данных. И у каждой

сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют **ключами**.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

Отличительной чертой **Entity Framework** является использование запросов **LINQ** для выборки данных из **БД**. С помощью **LINQ** мы можем не только извлекать определенные строки, хранящие объекты, из **БД**, но и получать объекты, связанные различными ассоциативными связями.

Другим ключевым понятием является **Entity Data Model**. Эта модель сопоставляет классы сущностей с реальными таблицами в **БД**.

Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставления (маппинга) служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц.

Таким образом, мы можем через классы, определенные в приложении, взаимодействовать с таблицами из базы данных.

Способы взаимодействия с БД

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

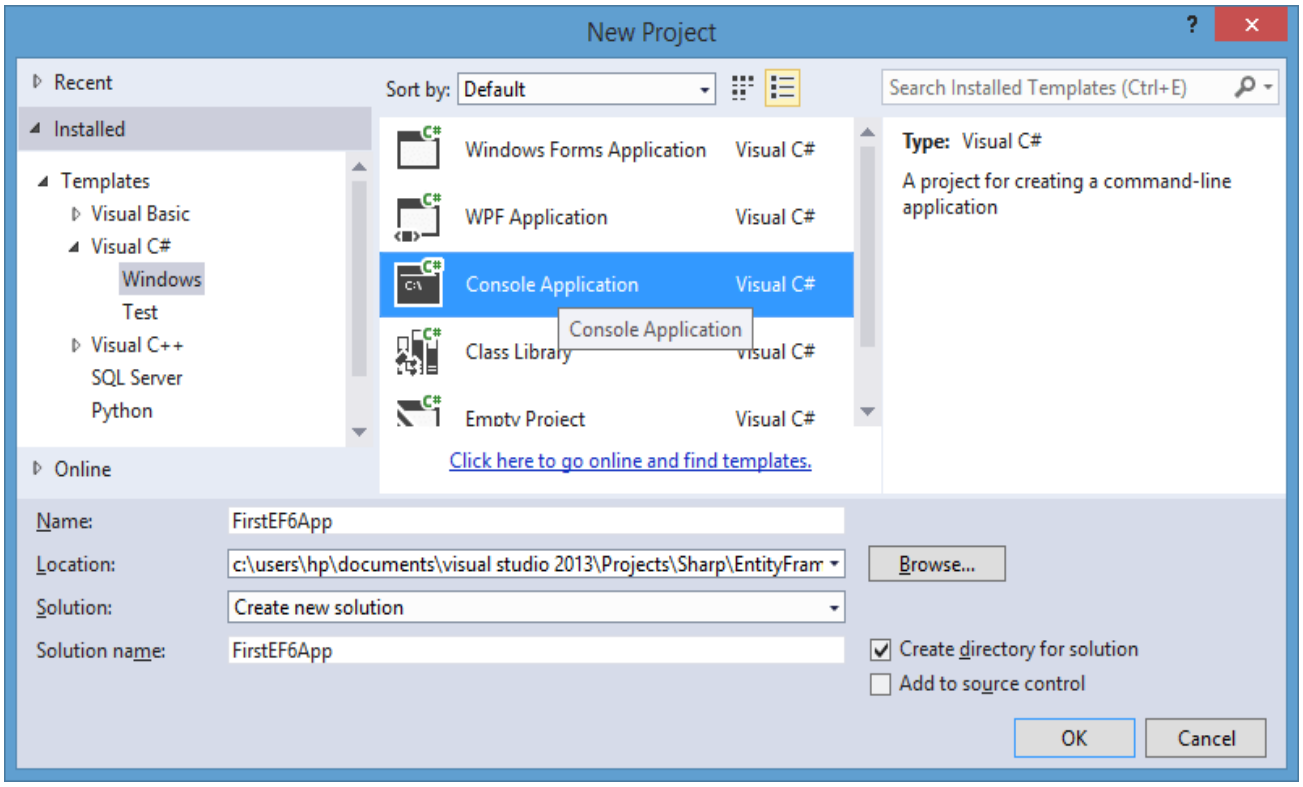
- **Database first**: **Entity Framework** создает набор классов, которые отражают модель конкретной базы данных
- **Model first**: сначала разработчик создает модель базы данных, по которой затем **Entity Framework** создает реальную базу данных на сервере.

- **Code first**: разработчик создает класс модели данных, которые будут храниться в **БД**, а затем **Entity Framework** по этому модели генерирует базу данных и ее таблицы.

Первое приложение с Entity Framework. Подход Code First

Чтобы непосредственно начать работать с **Entity Framework**, создадим первое приложение. Для этого нам нужна будет, во-первых, среда разработки. В качестве среды разработки выберем **Visual Studio 2013**. Можно также выбрать бесплатный выпуск **Visual Studio 2013 Express for Desktop**.

Создадим новый проект по типу **Console Application**.



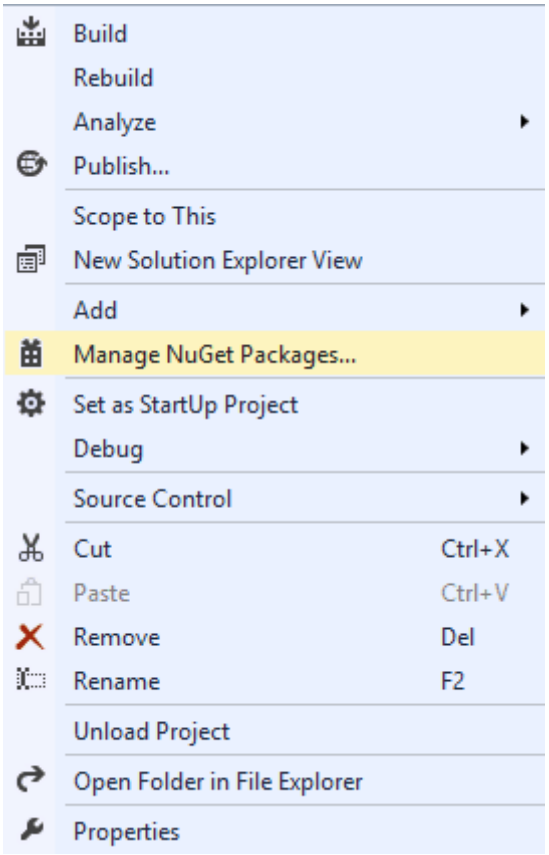
Теперь первым делом добавим новый класс, который будет описывать данные. Пусть наше приложение будет посвящено работе с пользователями. Поэтому добавим в проект новый класс **User**:

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

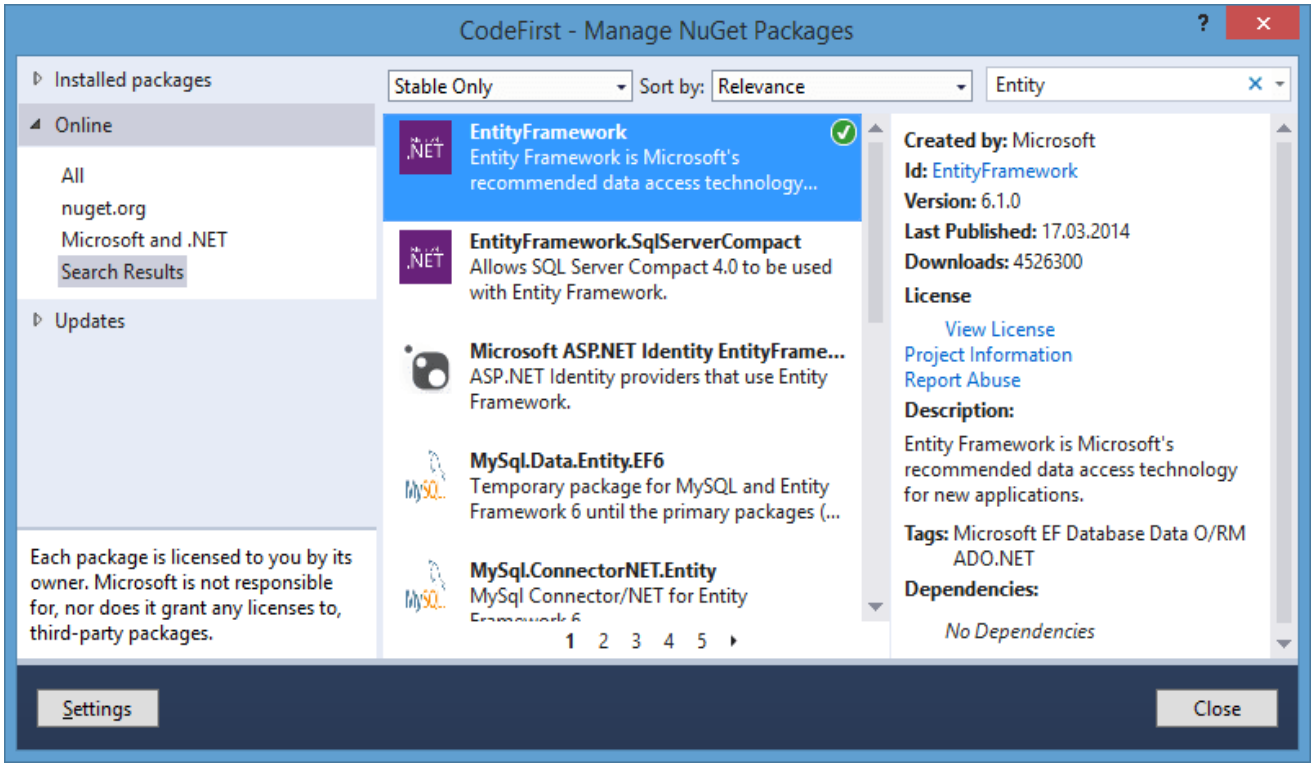
Это обычный класс, который содержит некоторое количество автосвойств. Каждое свойство будет сопоставляться с отдельным столбцом в таблице из **БД**.

Надо отметить, что **Entity Framework** при работе с **Code First** требует определения ключа элемента для создания первичного ключа в таблице в **БД**. По умолчанию при генерации **БД Entity Framework** в качестве первичных ключей будет рассматривать свойства с именами **Id** или **[Имя_класса]Id** (то есть **UserId**). Если же мы хотим назвать ключевое свойство иначе, то нам нужно будет внести дополнительную логику на **C#**.

Теперь для взаимодействия с **БД** нам нужен контекст данных. Это своего рода посредник между **БД** и классами, описывающими данные. Но, у нас по умолчанию еще не добавлена библиотека для **Entity Framework**. Чтобы ее добавить, нажмем на проект правой кнопкой мыши и выберем в контекстном меню **Manage NuGet Packages...**:



Затем в появившемся окне управления **NuGet**-пакетами в окне поиска введем слово "**Entity**" и выберем пакет собственно **Entity Framework** и установим его:



После установки пакета добавим в проект новый класс **UserContext**:

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

namespace FirstEF6App
{
    class UserContext : DbContext
    {
        public UserContext()
            : base("DbConnection")
        { }

        public DbSet<User> Users { get; set; }
    }
}
```

Основу функциональности **Entity Framework** составляют классы, находящиеся в пространстве имен **System.Data.Entity**. Среди всего набора классов этого пространства имен следует выделить следующие:

- **DbContext**: определяет контекст данных, используемый для взаимодействия с базой данных.

- **DbModelBuilder**: сопоставляет классы на языке **C#** с сущностями в базе данных.
- **DbSet/DbSet<TEntity>**: представляет набор сущностей, хранящихся в базе данных

В любом приложении, работающим с **БД** через **Entity Framework**, нам нужен будет контекст (класс производный от **DbContext**) и набор данных **DbSet**, через который мы сможем взаимодействовать с таблицами из **БД**. В данном случае таким контекстом является класс **UserContext**.

В конструкторе этого класса вызывается конструктор базового класса, в который передается строка "**DbConnection**" - это имя будущей строки подключения к базе данных. В принципе мы можем не использовать конструктор, тогда в этом случае строка подключения носила бы имя самого класса контекста данных.

И также в классе определено одно свойство **Users**, которое будет хранить набор объектов **User**. В классе контекста данных набор объектов представляет класс **DbSet<T>**. Через это свойство будет осуществляться связь с таблицей объектов **User** в **БД**.

И теперь нам надо установить подключение к базе данных. Для установки подключения обычно используется файл конфигурации приложения. В проектах для десктопных приложений файл конфигурации называется **App.config** (как в нашем случае), в проектах веб-приложений - **web.config**. В нашем случае это файл **App.config**. После добавления **Entity Framework** он выглядит примерно следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <entityFramework>
```

```
    <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

После закрывающего тега `</configSections>` добавим следующий элемент:

```
<connectionStrings>
    <add name="DBConnection" connectionString="data
source=(localdb)\v11.0;Initial Catalog=userstore.mdf;Integrated
Security=True;"
 providerName="System.Data.SqlClient"/>
  </connectionStrings>
```

Все подключения к источникам данных устанавливаются в секции **connectionStrings**, а каждое отдельное подключение представляет элемент **add**. В конструкторе класса контекста **UserContext** мы передаем в качестве названия подключения строку "**DbConnection**", поэтому данное название указывается в атрибуте name="DBConnection".

Настройку строки подключения задает атрибут **connectionString**. В данном случае мы устанавливаем название базы данных, с которой будем взаимодействовать - **userstore.mdf**.

Теперь перейдем к файлу **Program.cs** и изменим его содержание следующим образом:

```
using System;
namespace FirstEF6App
{
    class Program
    {
        static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                // создаем два объекта User
                User user1 = new User { Name = "Tom", Age = 33 };
```

```
                User user2 = new User { Name = "Sam", Age = 26 };

                // добавляем их в бд
                db.Users.Add(user1);
                db.Users.Add(user2);
                db.SaveChanges();
                Console.WriteLine("Объекты успешно сохранены");

                // получаем объекты из бд и выводим на консоль
                var users = db.Users;
                Console.WriteLine("Список объектов:");
                foreach (User u in users)
                {
                    Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name,
u.Age);
                }
            }
            Console.Read();
        }
    }
}
```

Так как класс **UserContext** через родительский класс **DbContext** реализует интерфейс **IDisposable**, то для работы с **UserContext** с автоматическим закрытием данного объекта мы можем использовать конструкцию **using**.

В конструкции **using** создаются два объекта **User** и добавляются в базу данных. Чтобы получить список данных из **БД**, достаточно воспользоваться свойством **Users** контекста данных: `var users = db.Users;`

В результате после запуска программа выведет на консоль:

Объекты успешно сохранены

Список объектов:

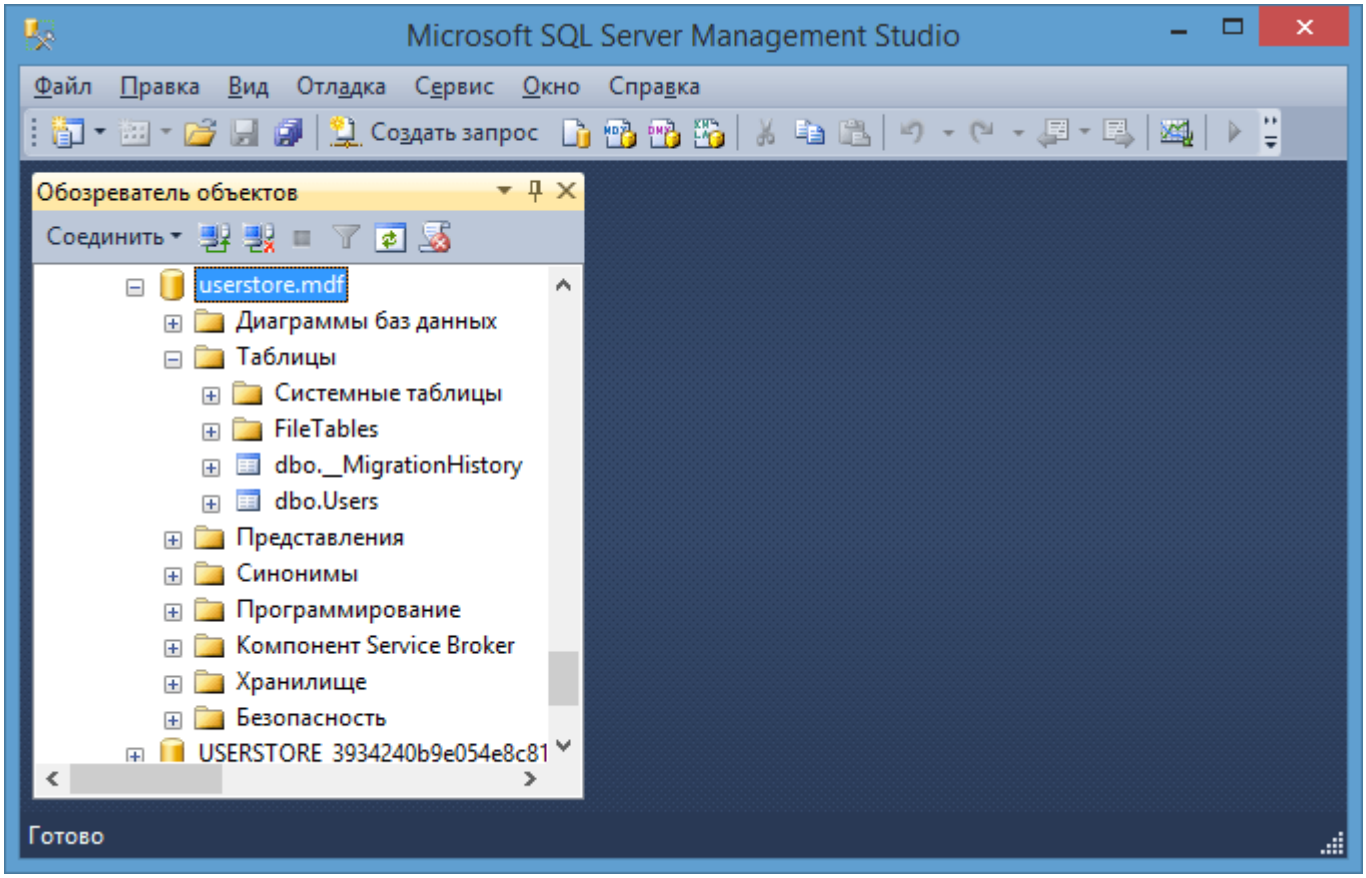
1.Tom - 33

2.Sam - 26

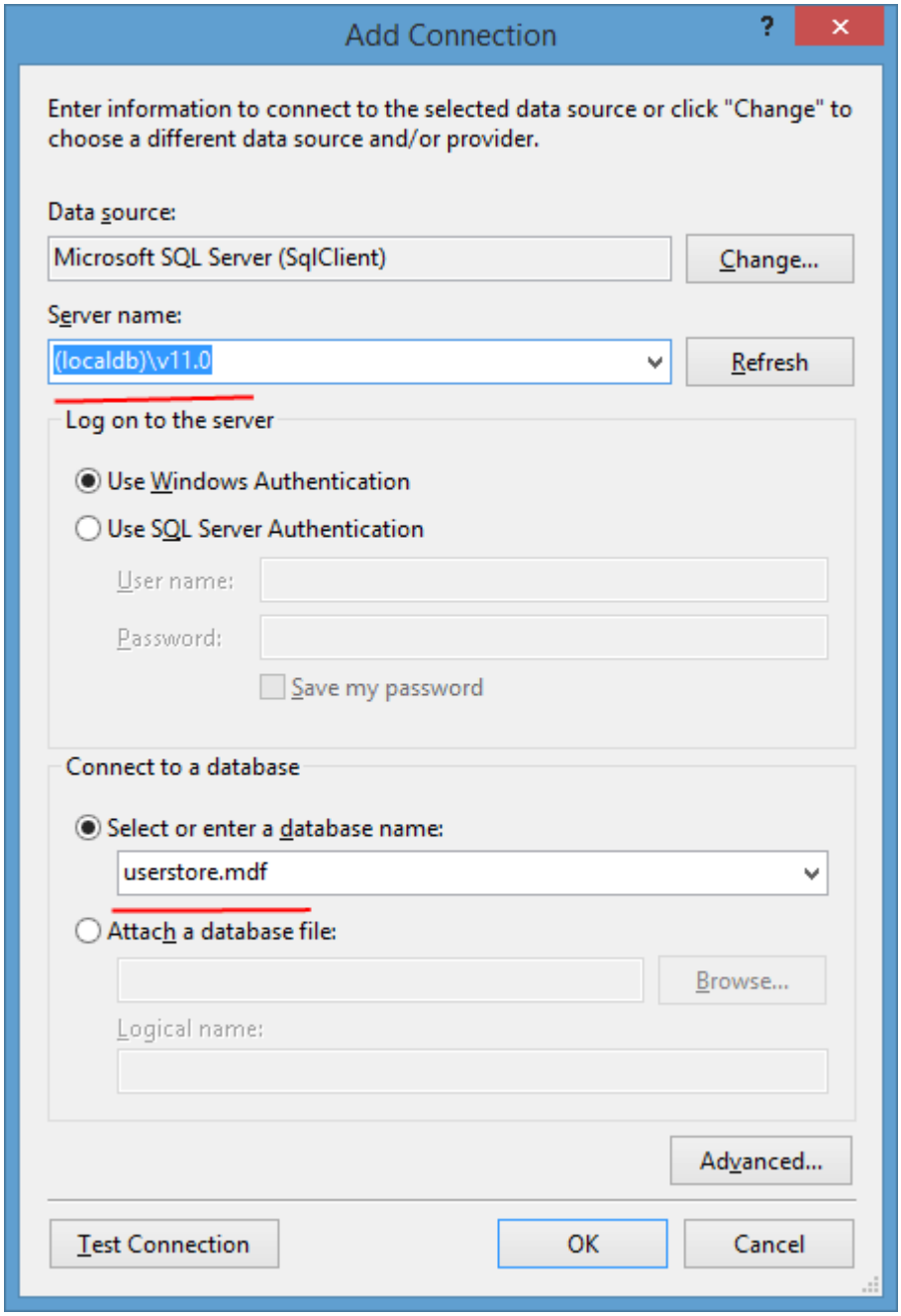
Таким образом, **Entity Framework** обеспечивает простое и удобное управление объектами из базы данных. При том в данном случае нам не надо даже создавать базу данных и определять в ней таблицы. **Entity Framework** все сделает за нас на основе определения класса контекста данных и классов моделей. И если база данных уже имеется, то **Entity Framework** не будет повторно создавать ее.

Наша задача - только определить модель, которая будет храниться в базе данных, и класс контекста. Поэтому данный подход называется **Code First** - сначала пишется код, а потом по нему создается база данных и ее таблицы.

Возникает вопрос, а где же находится **БД**? Чтобы физически увидеть базу данных, мы можем подключиться к ней из **Visual Studio** через окно **Database Explorer** или через специальный инструмент управления **SQL Server Management Studio**:



Для просмотра базы данных через **Visul Studio** выберем в меню пункт **View->Other Windows->Database Explorer**. В нем окне **Database Explorer** подключимся к новой базе данных, выбрав **Connect to Database**.



В окне добавления подключения укажем сервер **(localdb)\v11.0** и название нашей базы данных.

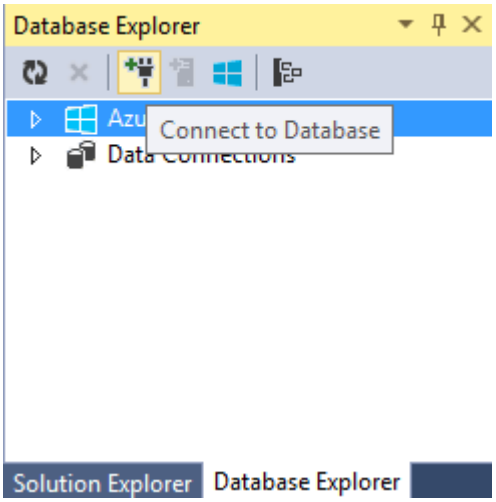
Физически база данных будет располагаться в каталоге **SQL Server**а, в частности, у меня она размещена в каталоге **C:\Program Files\Microsoft SQL Server\MSSQL11.SQLEXPRESS\MSSQL\DATA**, только назваться она будет по названию подключения - **DbConnection.mdf**.

2.Взаимодействие с данными. Подходы

Code First к существующей базе данных

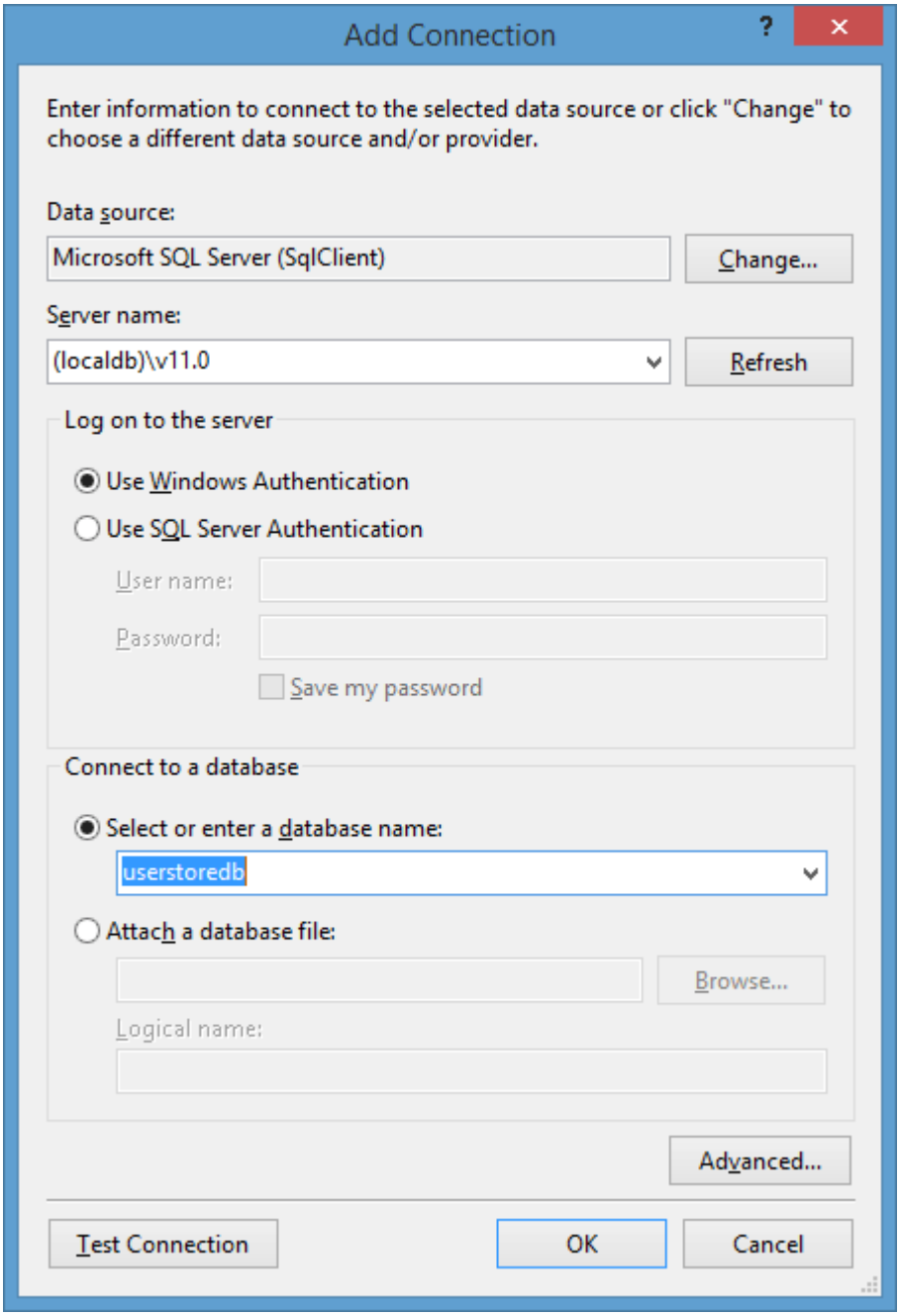
В первой главе при создании первого приложения с **Entity Framework** мы использовали подход **Code First**. Этот подход очень прост и удобен. Но он также и очень гибкий. Так, вполне часто распространена ситуация, когда база данных уже имеется. И здесь опять же поможет **Code First**. Иногда программисты называют данный подход **Code Second**. Посмотрим на примере.

Вначале создадим новый проект. Затем создадим тестовую базу данных. В **Visual Studio** выберем в меню пункт **View->Other Windows->Database Explorer**. В нем окне **Database Explorer** подключимся к новой базе данных, выбрав **Connect to Database**.

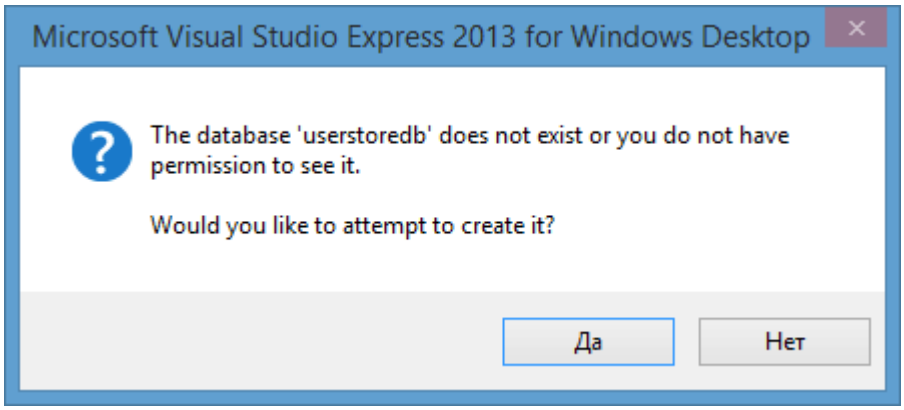


В окне создания подключения в качестве сервера выберем **(localdb)\v11.0**. В данном случае мы выбираем движок **localdb** для работы с **MS SQL Server**, который предназначен специально для целей разработки. Хотя мы также могли выбрать, как в предыдущей теме полноценный **SQL Server Express**.

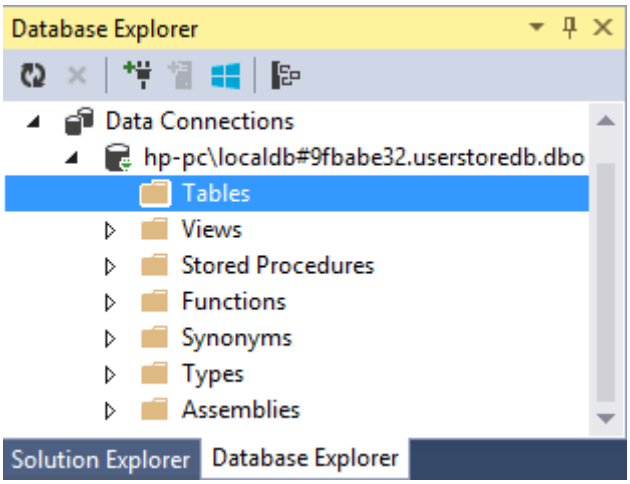
А в качестве имени базы данных введем **userstoredb**:



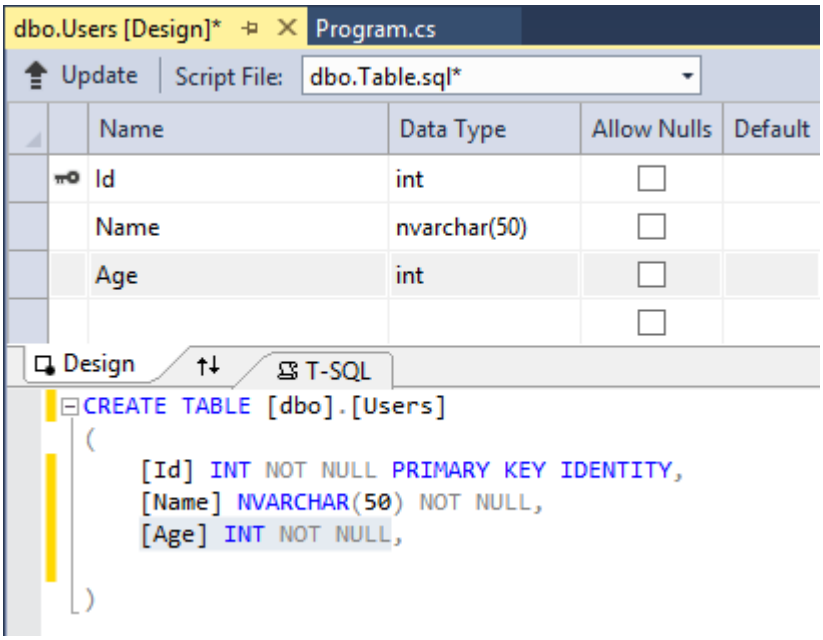
И если база данных не существует, нам отобразится окно с подтверждением ее создания:



Нажмем '**Да**'. И после этого в окне **Database Explorer** отобразится созданная база данных:

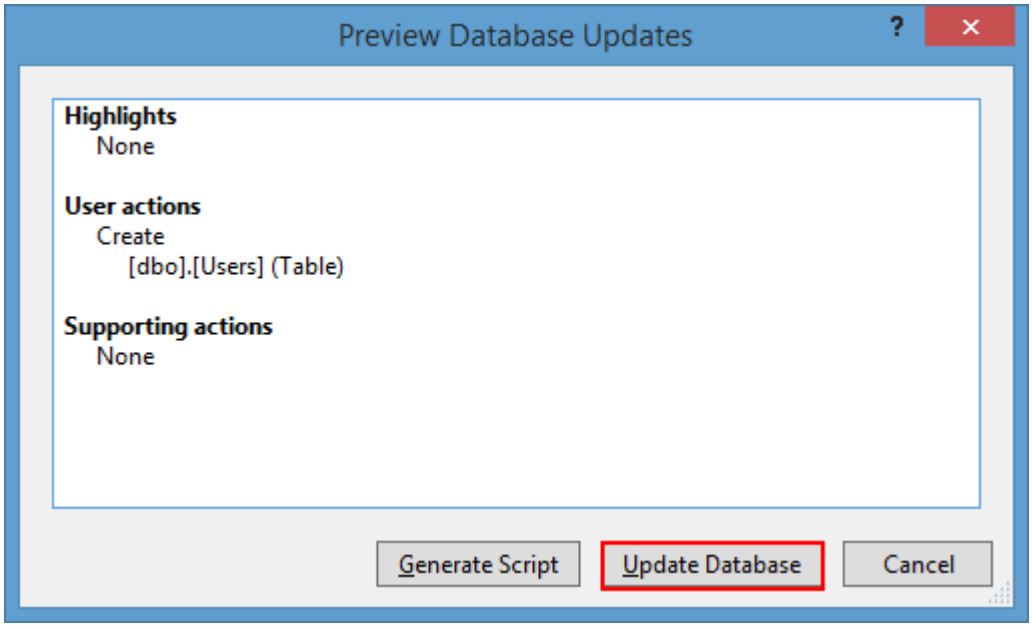


Она еще пуста, поэтому добавим в нее таблицу. Нажмем правой кнопкой мыши на узел **Tables** и в появившемся контекстном меню выберем **Add New Table**. Затем в центральном поле в режиме дизайнера создадим следующее определение таблицы:



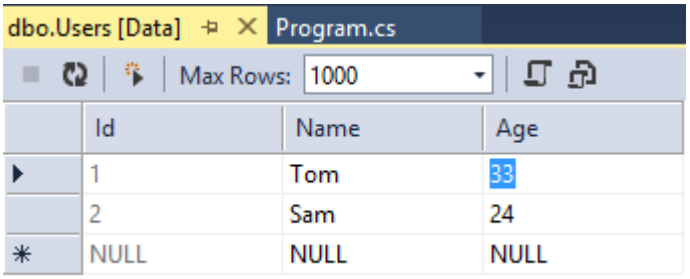
В поле **T-SQL** (или графически) определим структуру и имя таблицы, типы столбцов и после этого нажмем в верхнем левом углу на кнопку **Update**.

В новом окне нам будет выдана некоторая информация об изменениях, производимых в **БД**:



Нажмем на кнопку **Update Database**. И после этого будет создана таблица **Users**. Обновив окно **Database Explorer** и открыв узел **Tables**, вы сможете увидеть новую таблицу **Users**.

Мы можем добавить некоторые данные в таблицу. Для этого нажмем на таблицу в окне **Database Explorer** правой кнопкой мыши и выберем пункт **Show Table Data** (Показать данные таблицы). У нас откроется форма для работы с данными, в которую введем пару строк:



База данных готова. Теперь нам надо добавить подключение в файл конфигурации приложения. В **Solution Explorer** найдем файл **App.config** и откроем него. Перед закрывающим тегом **</configuration>** добавим новую секцию:

```
<connectionStrings>
    <add name="UserDB" connectionString="data
source=(localdb)\v11.0;Initial Catalog=userstoredb;Integrated
Security=True;"
 providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Теперь определим классы модели данных и контекста. Добавим класс модели **User**:

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

И также добавим класс контекста данных:

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

namespace FirstEF6App
{
    class UserContext : DbContext
    {
        public UserContext()
            : base("DbConnection")
        { }

        public DbSet<User> Users { get; set; }
    }
}
```

В конструкторе контекста данных мы передаем в конструктор базового класса имя строки подключения из файла конфигурации **App.config**. Так как мы определили там строку подключения **UserDB (<add name="UserDB")**, то именно это значение и используется в конструкторе.

Однако, как вариант, мы могли не использовать конструктор в классе контекста данных, а определить в качестве имени строки подключения название этого класса, например: **<add name="UserContext" connectionString="....**

И для получения данных определим следующий код в консольном приложении:

```
using (UserContext db = new UserContext())
            {
                var users = db.Users;
```

```
            foreach (User u in users)
            {
                Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name,
u.Age);
            }
        }
```

Соглашения по наименованию в Code First

При создании таблиц и их столбцов в базе данных в **Entity Framework** по умолчанию действуют некоторые соглашения по именованию, которые указывают, какие имена должны быть у таблиц, столбцов, какие типы и т.д. Рассмотрим некоторые из этих соглашений.

Сопоставление типов

Типы **SQL Server** а и **C#** сопоставляются следующим образом:

- **int : int**
- **bit : bool**
- **char : string**
- **date : DateTime**
- **datetime : DateTime**
- **datetime2 : DateTime**
- **decimal : decimal**
- **float : double**
- **money : decimal**
- **nchar : string**
- **ntext : string**
- **numeric : decimal**
- **nvarchar : string**
- **real : float**
- **smallint : short**
- **text : string**
- **tinyint : byte**
- **varchar : string**

NULL и NOT NULL

Все первичные ключи по умолчанию имеют определение **NOT NULL**.

Столбцы, сопоставляемые со свойствами ссылочных типов (**string, array**), в базе данных имеют определение **NULL**, а все значимые типы (**DateTime, bool, char, decimal, int, double, float**) - **NOT NULL**.

Если свойство имеет тип **Nullable<T>**, то оно сопоставляется со столбцом с определением **NULL**.

Ключи

Entity Framework требует наличия первичного ключа, так как это позволяет ему отслеживать объекты. По умолчанию в качестве ключей **EF** рассматривает свойства с именем **Id** или **[Название_типа]Id** (например, **PostId** в классе **Post**).

Как правило, ключи имеют тип **int** или **GUID**, но также могут представлять и любой другой примитивный тип.

Названия таблиц и столбцов

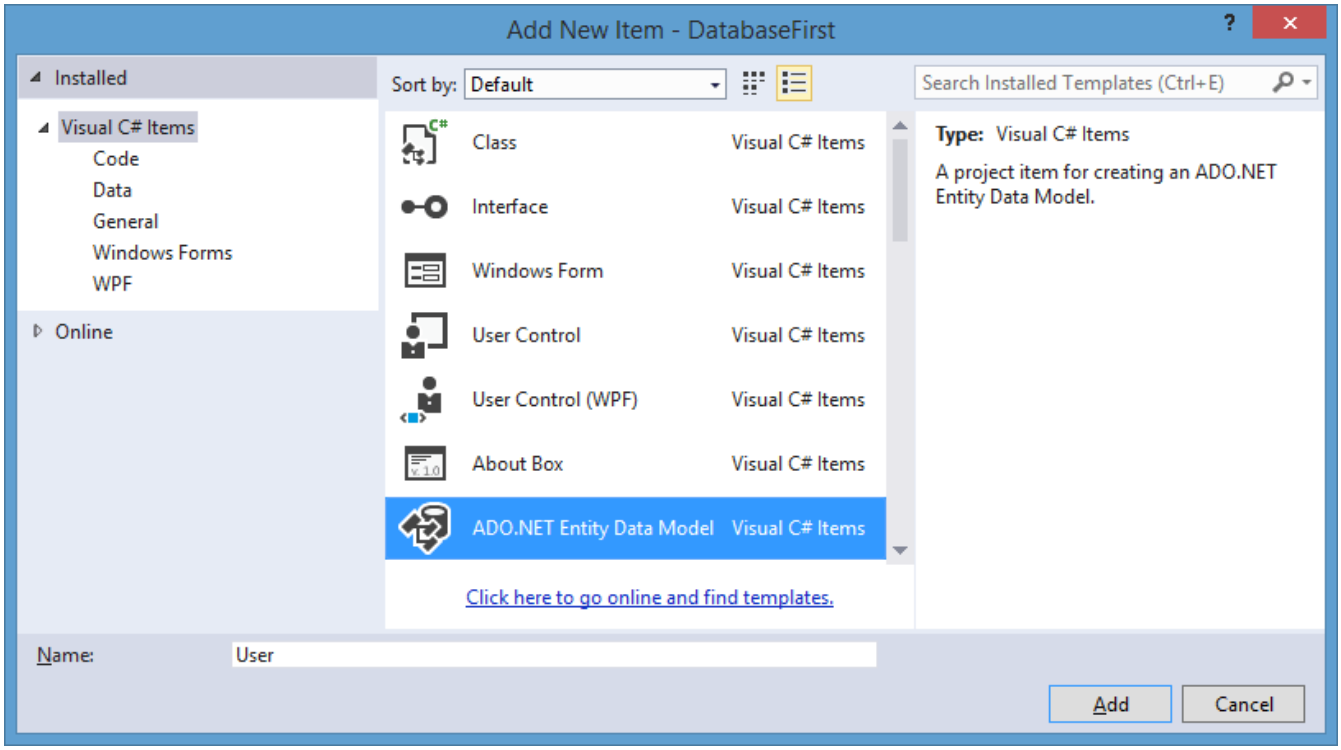
С помощью специального класса **PluralizationService Entity Framework** проводит сопоставление между именами классов моделей и именами таблиц. При этом таблицы получают по умолчанию в качестве названия множественное число в соответствии с правилами английского языка, например, класс **User** - таблица **Users**, класс **Person** - таблица **People** (но не **Persons!**).

Названия столбцов получают названия свойств модели.

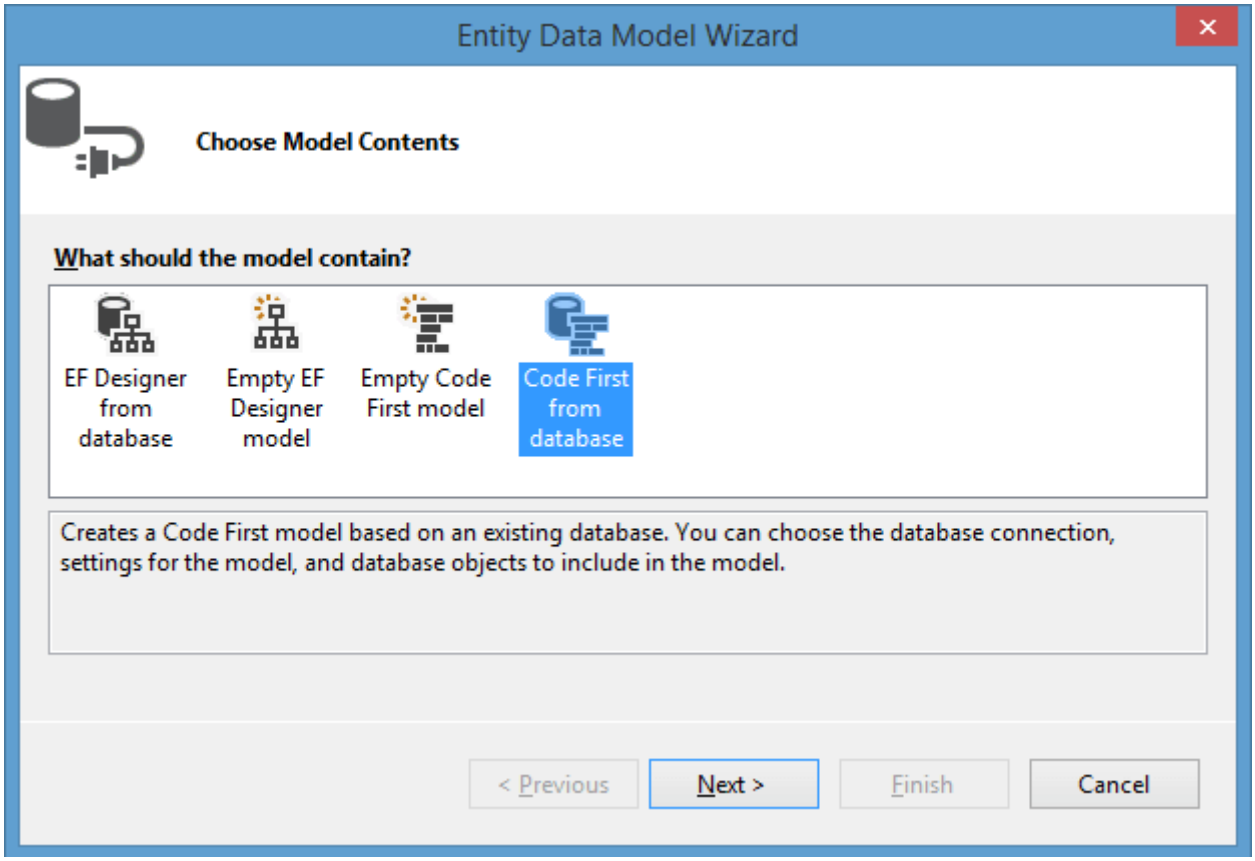
Если нас не устраивают названия таблиц и столбцов по умолчанию, то мы можем переопределить данный механизм с помощью **Fluent API** или аннотаций.

Автоматизация Code First

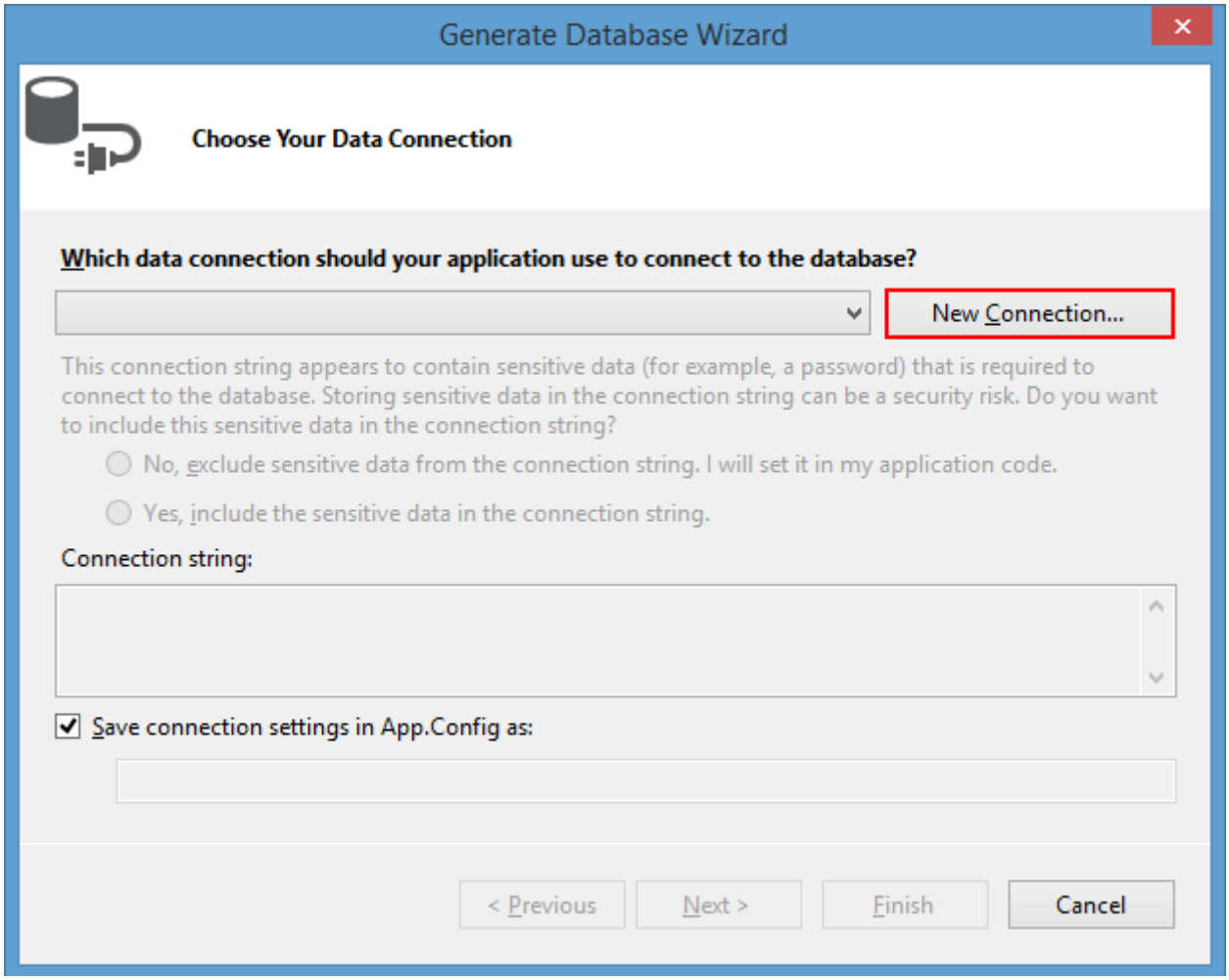
Вручную создавать классы по уже готовой **БД** со всеми полями и связями между собой довольно утомительно, особенно если таблиц в **БД** очень много. В обновленных версиях **Visual Studio 2013** с пакетами обновления **SP3** мы можем автоматизировать этот процесс. Для этого добавим в проект новый элемент **ADO.NET Entity Data Model**:



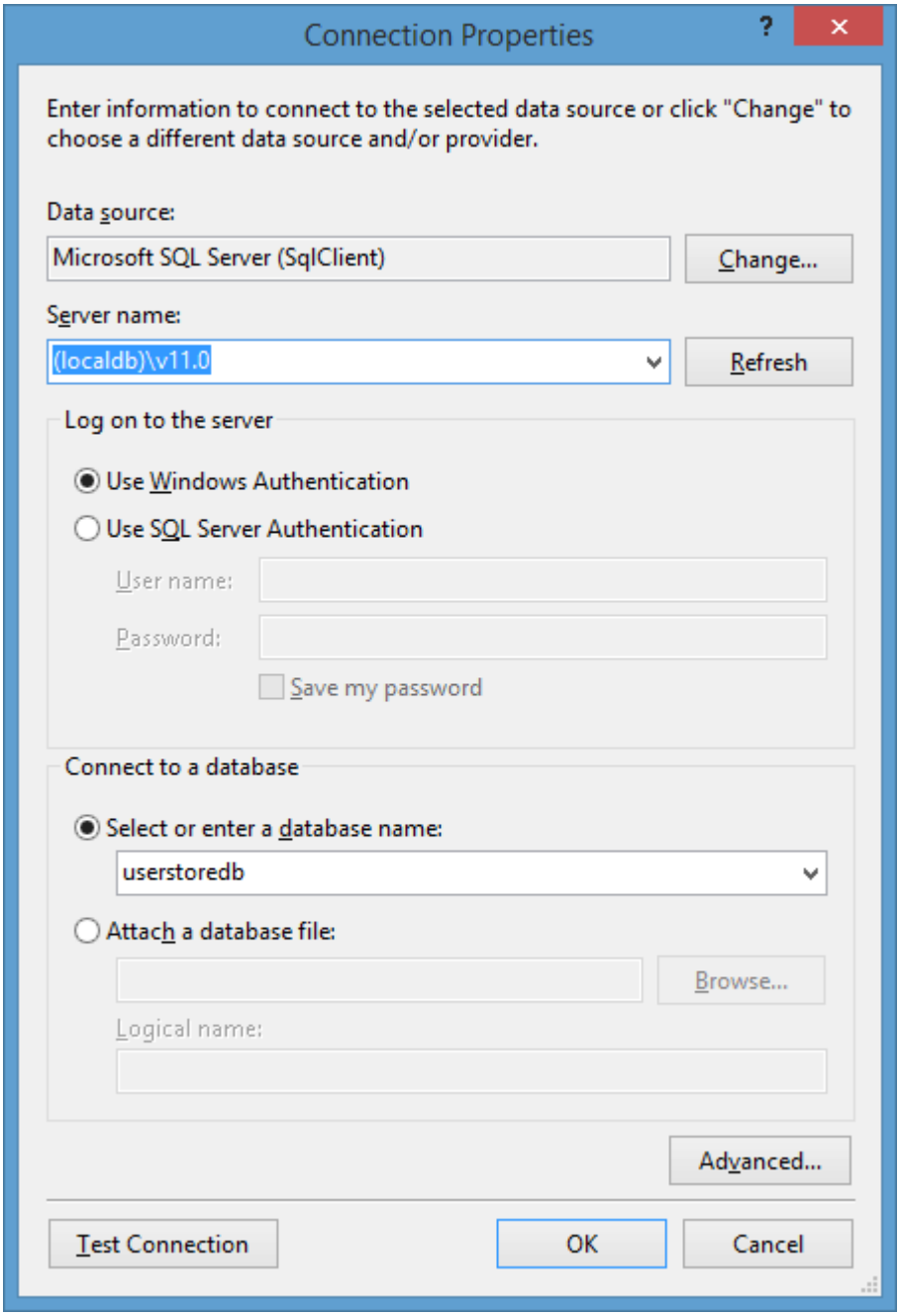
Нажмем **OK** и нам откроется мастер создания модели. Здесь нам надо выбрать пункт **Code First from database**:



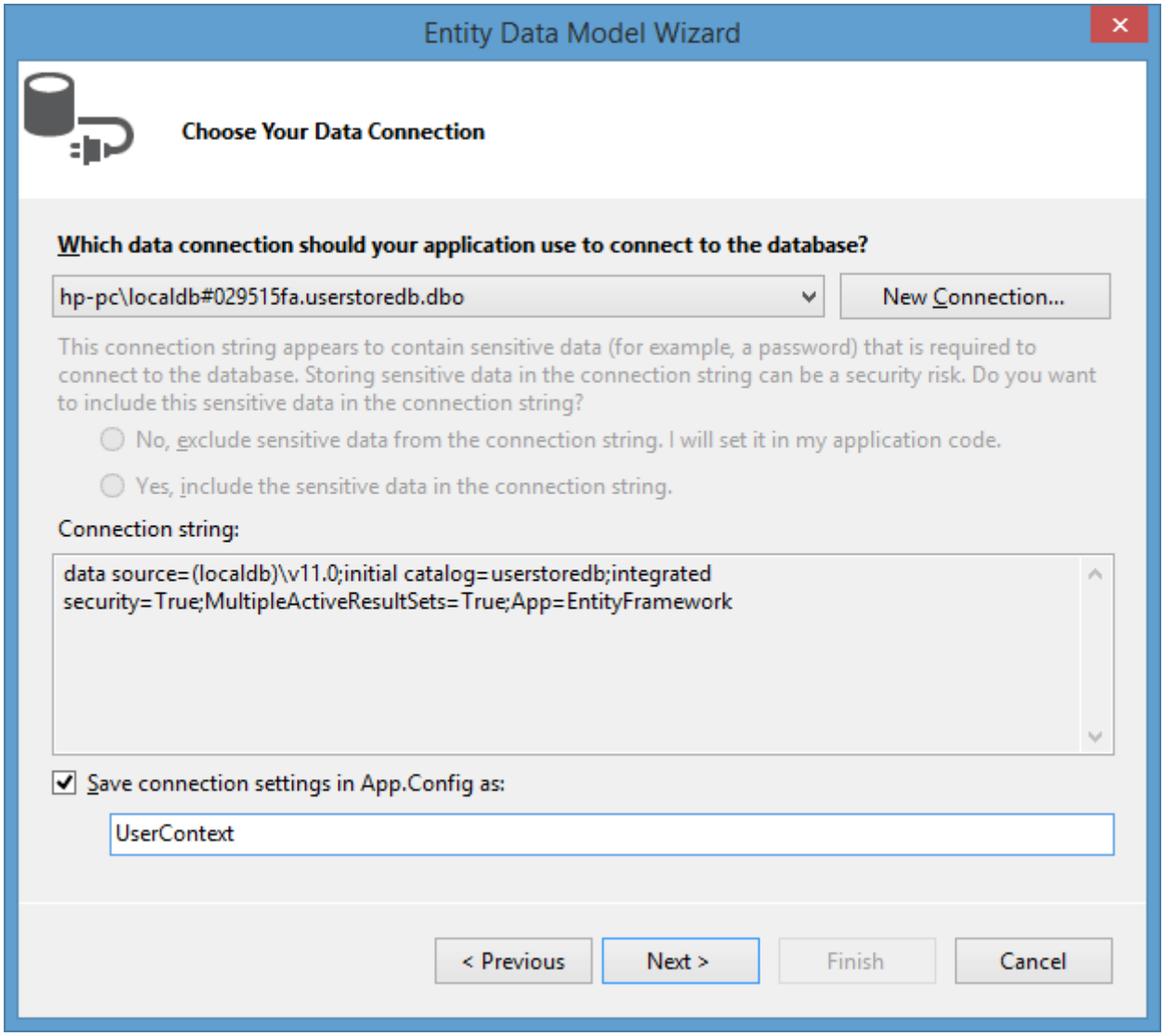
Далее на следующем шаге настройки модели надо будет установить подключение к имеющейся базе данных.



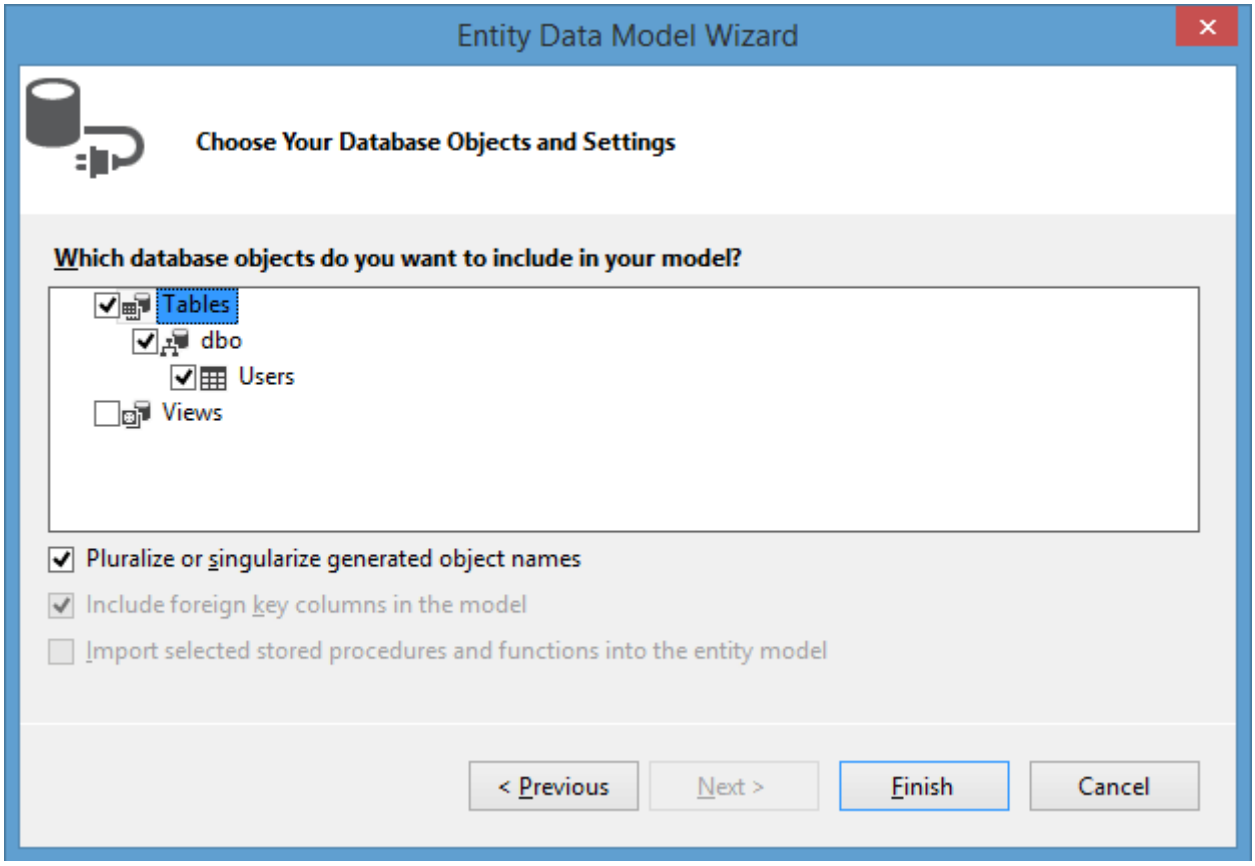
Нажмем на кнопку **New Connection** и в следующем окне настроек подключения выберем сервер и базу данных, с которой мы хотим работать:



После этого в окне мастера настройки модели появится выбранное подключение. И также здесь мы можем установить название подключения, которое будет использоваться в файле конфигурации **App.config**. Изменим его, например, на **UserContext**:



Нажмем **Next**, и на следующем шаге нам будет предложено выбрать те таблицы из **БД**, по которым нам надо создать модели:



И затем нажмем **Finish**. После этого будут сгенерированы классы моделей. Например, в моем случае по единственной таблице в **БД** будет сгенерирован следующий класс:

```
namespace NewAutoCodeSecond
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;
    using System.ComponentModel.DataAnnotations.Schema;
    using System.Data.Entity.Spatial;

    public partial class User
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public int Age { get; set; }
    }
}
```

И также надо отметить, что в файле **App.config** появилось определение подключения:

```
<connectionStrings>
    <add name="UserContext" connectionString="data
source=(localdb)\v11.0;initial catalog=userstore.mdf;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Для полноценной работы нам осталось добавить класс контекста данных:

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

namespace NewAutoCodeSecond
{
    class UserContext : DbContext
    {
        public UserContext()
            : base("UserContext")
        { }
        public DbSet<User> Users { get; set; }
    }
}
```

И теперь мы можем взаимодействовать с базой данных:

```
using System;

namespace NewAutoCodeSecond
{
    class Program
    {
        static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                foreach (User u in db.Users)
                    Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name,
u.Age);
            }
            Console.ReadKey();
        }
    }
}
```

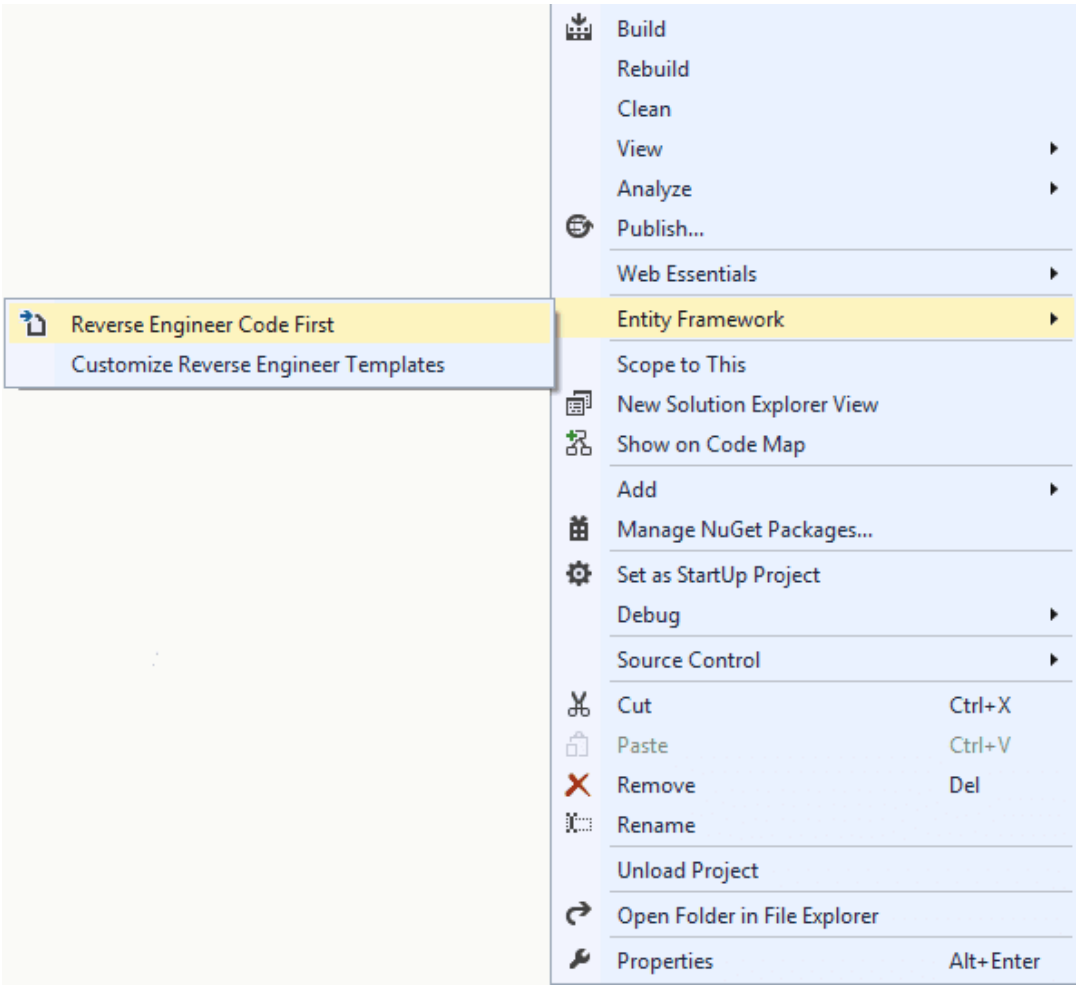
Автоматизация Code First и EF Power Tools

Кроме вышеописанной функциональность **Microsoft** предлагает нам полезный инструмент, также призванный автоматизировать данный процесс. Этот инструмент называется **EF Power Tools**.

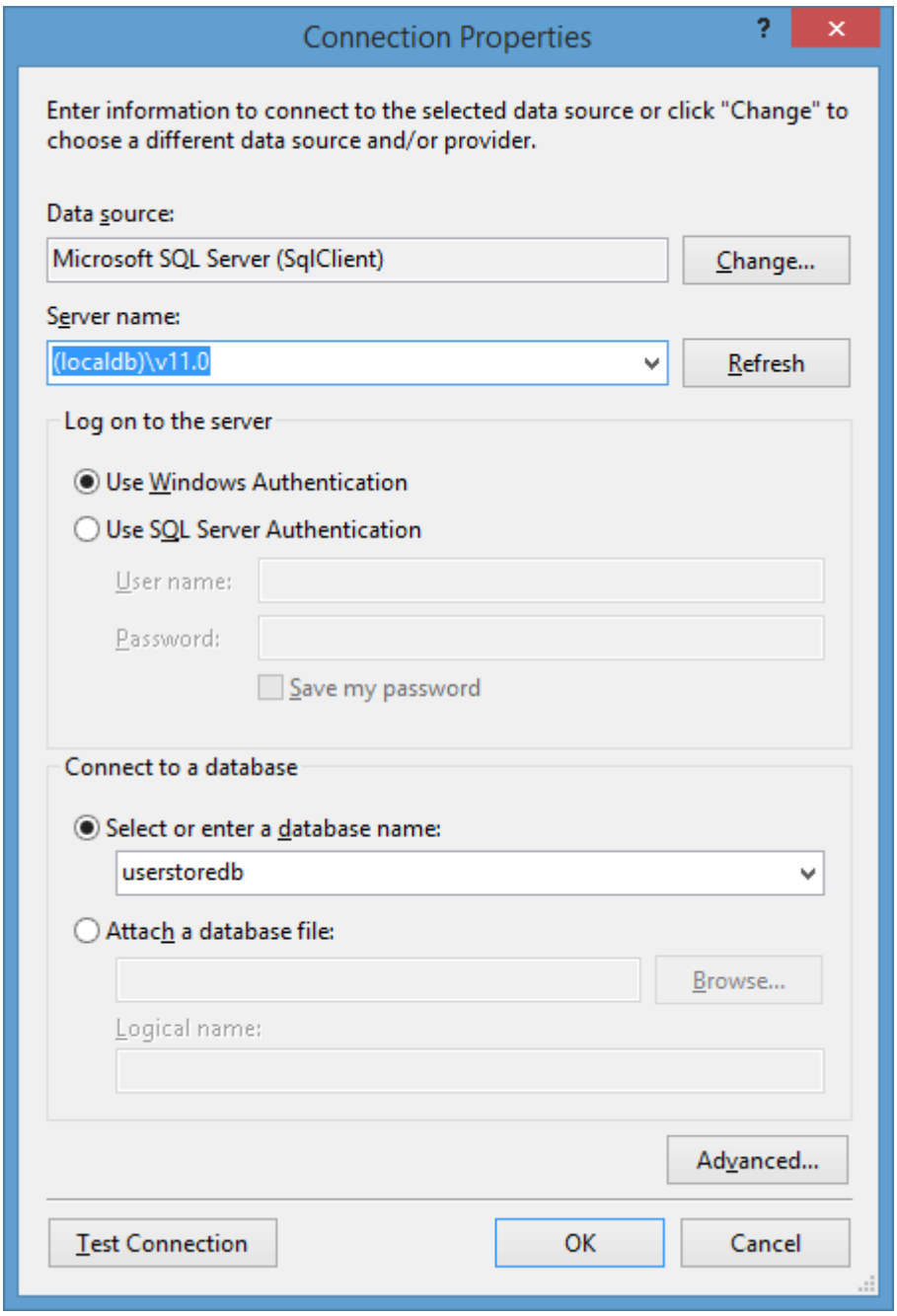
EF Power Tools представляет собой надстройку к **Visual Studio**. Эту надстройку, а также краткое описание можно найти на странице **Entity Framework Power Tools**. Для установки достаточно нажать на этой веб-странице на кнопку "**Загрузка**" и дальше следовать инструкциям.

В то же время есть некоторые ограничения: надстройка **Entity Framework Power Tools** работает только в полных версиях **Visual Studio**. В экспресс же версиях не работает.

Итак, если у вас полнофункциональная версия **Visual Studio 2012** или **2013**, то вы можете нажать в окне **Solution Explorer** (Обозреватель решений) на проект правой кнопкой мыши и увидеть в контекстном меню пункт **Entity Framework**:

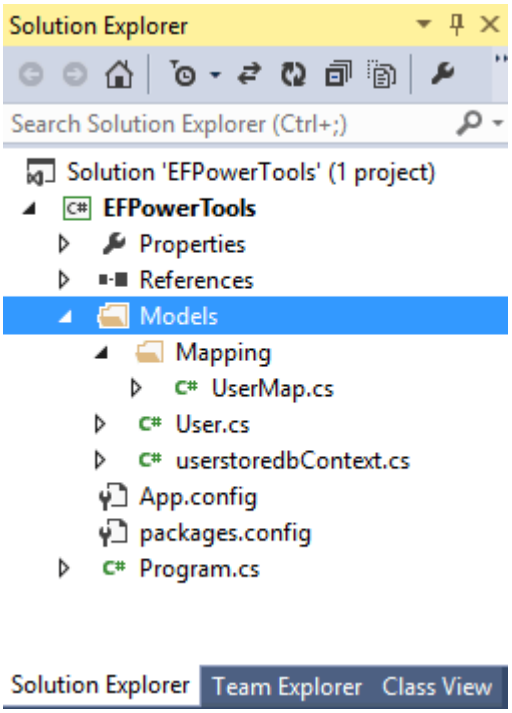


Выберем пункт **Entity Framework –> Reverse Engineer Code First**. Затем откроется окно настройки подключения, где нам надо задать сервер и базу данных, с которой нам надо взаимодействовать:



Я выбрал базу данных, которая была создана в прошлой теме.

Затем нажмем **OK**. После этого в проект будет добавлена папка **Models**, в которой будут находиться все созданные классы. По умолчанию для каждой таблицы создается свой класс, и также генерируется класс контекста данных:



Поскольку в моей базе данных была одна таблица **Users**, то автоматически был создан класс **User**, который отражает структуру таблицы:

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

namespace EFPowerTools.Models
{
    public partial class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

Контекст данных **userstoredbContext**:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using EFPowerTools.Models.Mapping;

namespace EFPowerTools.Models
{
    public partial class userstoredbContext : DbContext
    {
        static userstoredbContext()
        {
            Database.SetInitializer<userstoredbContext>(null);
```

```
        }

        public userstoredbContext()
            : base("Name=userstoredbContext")
        {
        }

        public DbSet<User> Users { get; set; }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Configurations.Add(new UserMap());
        }
    }
}
```

Он имеет два конструктора. Статический конструктор призван выполнить начальную инициализацию данных. В данном случае он ничего не выполняет.

Стандартный конструктор обращается к конструктору базового класса (то есть класса **DbContext**) и передает ему название строки подключения **(base("Name=userstoredbContext")).**

Для взаимодействия с таблицей **Users** класс контекста имеет одноименное свойство **public DbSet<User> Users { get; set; }**

И в методе **OnModelCreating** выполняются действия при создании моделей. В данном случае с помощью класса **UserMap** настраивается конфигурация связей между классами и базой данных.

Этот класс **UserMap** выполняет сопоставления между столбцами и таблицами и классами и их свойствами:

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;

namespace EFPowerTools.Models.Mapping
{
    public class UserMap : EntityTypeConfiguration<User>
    {
        public UserMap()
        {
            // Primary Key
            this.HasKey(t => t.Id);

            // Properties
            this.Property(t => t.Name)
```

```
                .IsRequired()
                .HasMaxLength(50);

            // Table & Column Mappings
            this.ToTable("Users");
            this.Property(t => t.Id).HasColumnName("Id");
            this.Property(t => t.Name).HasColumnName("Name");
            this.Property(t => t.Age).HasColumnName("Age");
        }
    }
}
```

Остальная работа с базой данных будет происходить также, как и при стандартном подходе **Code First**. То есть, если нам надо добавить в таблицу новый объект **User**, то мы пишем:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using EFPowerTools.Models.Mapping;

namespace EFPowerTools.Models
{
    public partial class userstoredbContext : DbContext
    {
        static userstoredbContext()
        {
            Database.SetInitializer<userstoredbContext>(null);
        }

        public userstoredbContext()
            : base("Name=userstoredbContext")
        {
        }

        public DbSet<User> Users { get; set; }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Configurations.Add(new UserMap());
        }
    }
}
```

Конечно, в реальности базы данных, как правило, обладают более сложной структурой, и поэтому структура генерируемых классов также будет

сложнее класса **User**. Но на данном примере уже понятно, что мы можем значительно автоматизировать часть работы по созданию классов моделей.

Database First

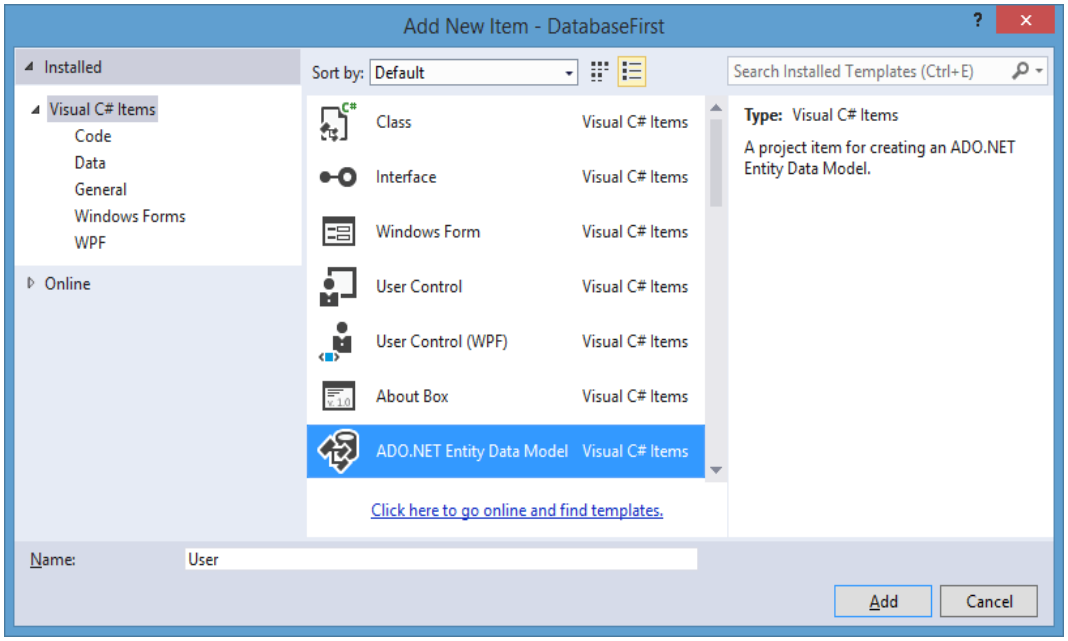
Database First был первым подходом, который появился в **Entity Framework**. Данный подход во многом похож на **Model First** и подходит для тех случаев, когда разработчик уже имеет готовую базу данных.

Чтобы **Entity Framework** мог получить доступ к базе данных, в системе должен быть установлен соответствующий провайдер. Так, **Visual Studio** уже поддерживает соответствующую инфраструктуру для **СУБД MS SQL Server**. Для остальных **СУБД**, например, **MySQL, Oracle** и других надо устанавливать соответствующие провайдеры. Список провайдеров для наиболее распространенных **СУБД** можно найти на странице **ADO.NET Data Providers**.

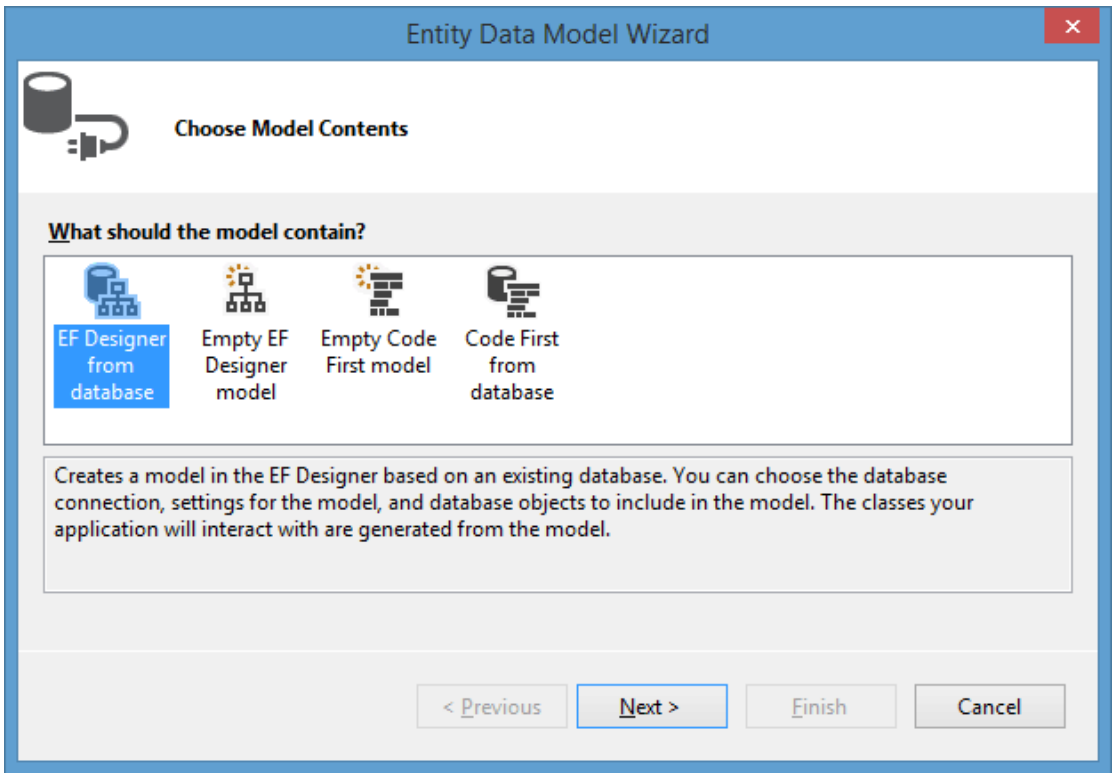
Итак, создадим новый проект по типу **Console Application**. Его функциональность будет той же самой, что и у предыдущих проектов, только подход к использованию **Entity Framework** будет отличаться.

После создания нового проекта, чтобы задействовать базу данных, нам надо ее иметь. Создадим новую **БД** или возьмем уже имеющуюся.

В **Visual Studio** в окне **Solution Explorer** нажмем на проект правой кнопкой мыши и выберем в **Add - > New Item**. Далее в появившемся окне добавления нового элемента выберем **ADO.NET Entity Data Model**. Дадим новому компоненту какое-либо название, например, **User**:

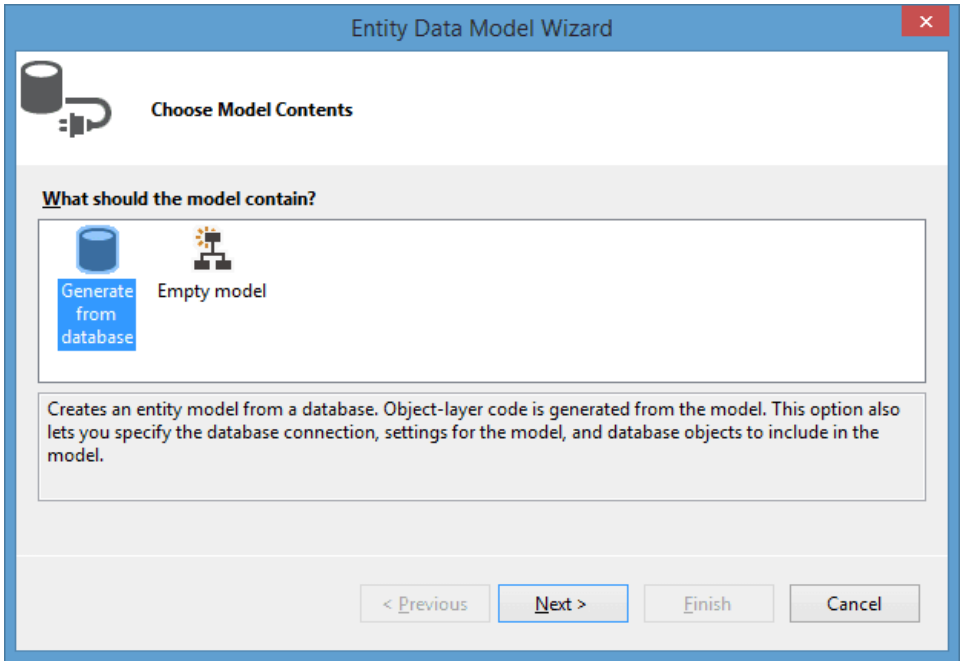


После этого нам откроется окно мастера создания модели. Если вы работаете с **Visual Studio 2013** с пакетом обновления **SP2, SP3**, то откроется следующее окно мастера модели:



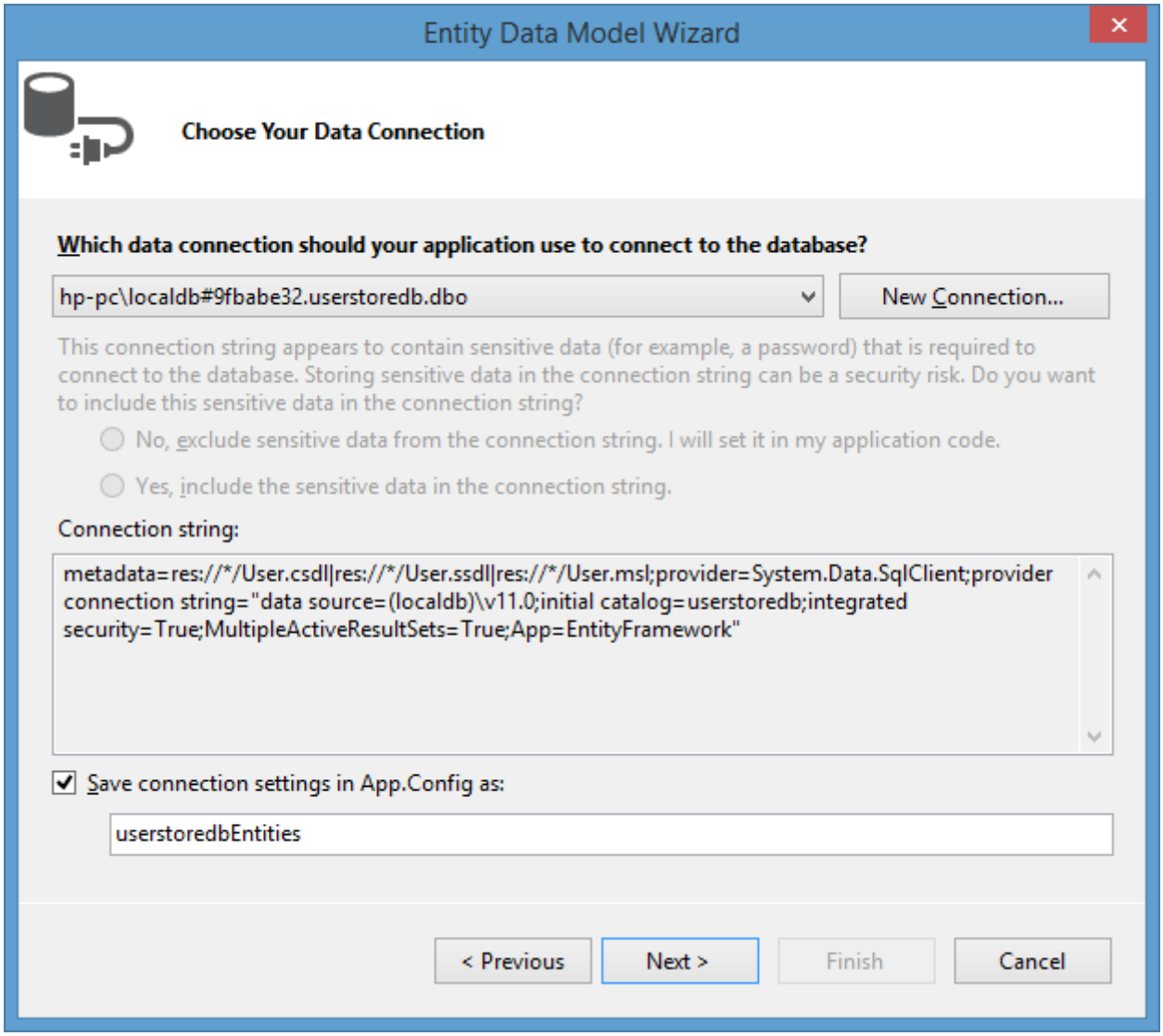
В нем нам надо выбрать опцию **EF Designer from database**.

Если нашей целевой средой является **Visual Studio 2013** без пакетов обновлений, то окно мастера будет выглядеть следующим образом:



В этом случае надо выбрать пункт **Generate from database** (Создание модели по имеющейся базе данных)

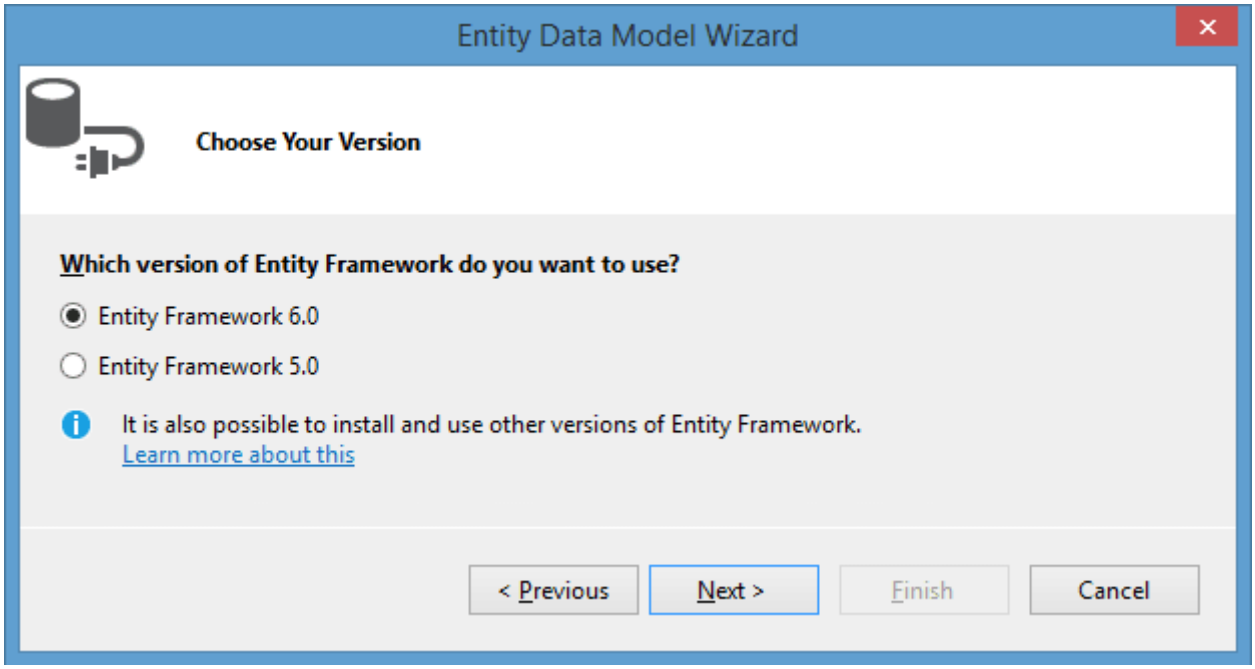
Затем откроется окно следующего шага по созданию модели, на котором надо будет установить подключение к базе данных:



В выпадающем списке выберем одно из доступных подключений. Если в списке нет предпочтительных подключений, то можно нажать на кнопку **New Connection** и установить новое подключение.

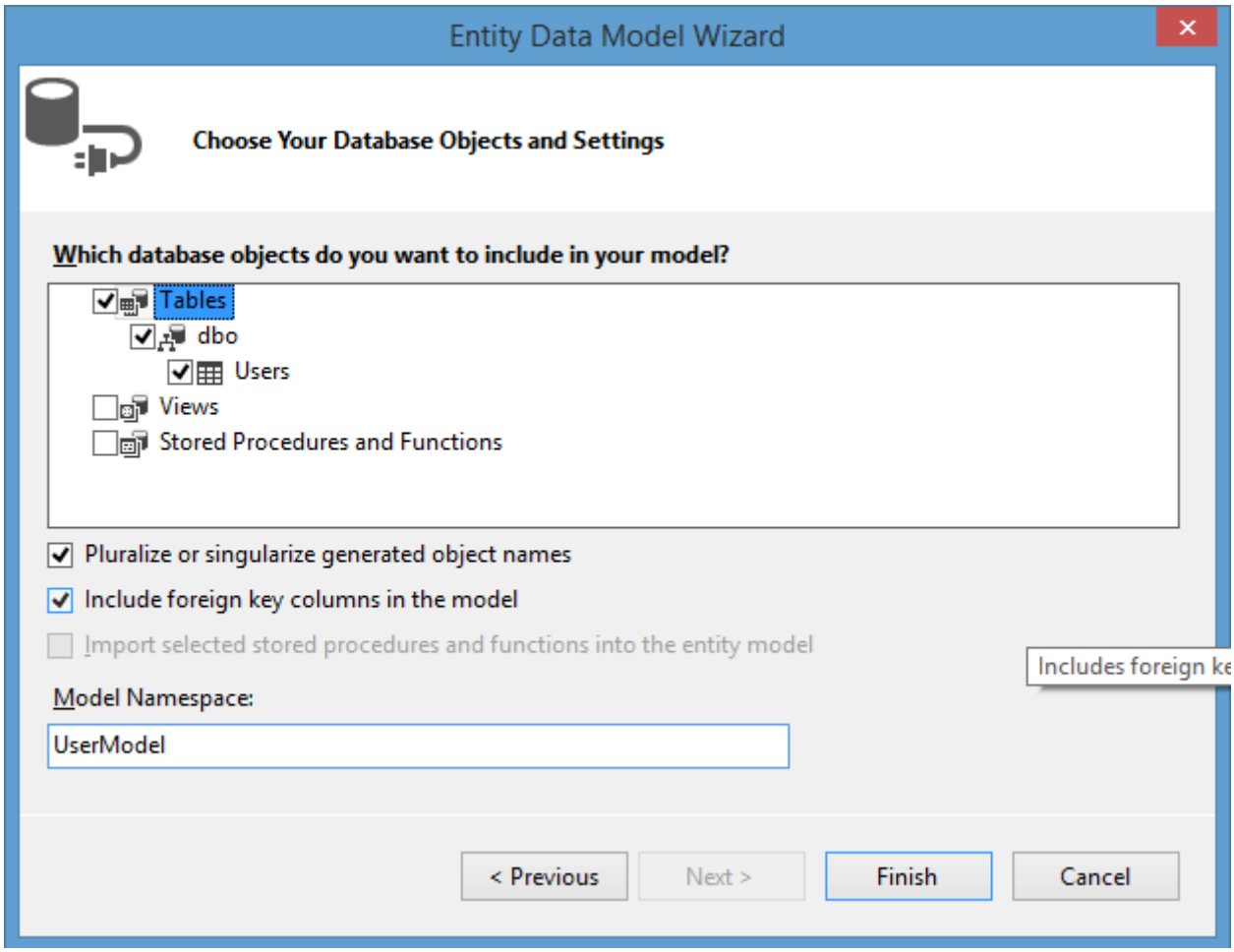
Также внизу указывается название **контекста данных**, который будет использоваться для доступа к данным. По умолчанию у меня контекст имеет название **userstoredbEntities**. Можно изменить, а можно и оставить.

Выбрав подключение, переходим к следующему шагу. Если у нас **Visual Studio 2013** без пакетов обновления, то будет предложено также выбрать версию **Entity Framework**. Выберем шестую версию:



В версиях **Visual Studio 2013 SP2, SP3** по умолчанию используется **EF 6**, поэтому этот шаг пропускается.

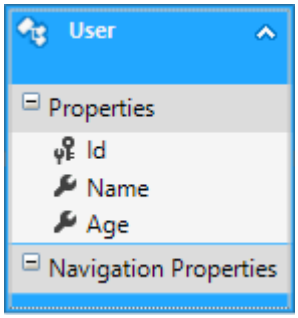
Далее **Visual Studio** извлекает всю информацию о базе данных:



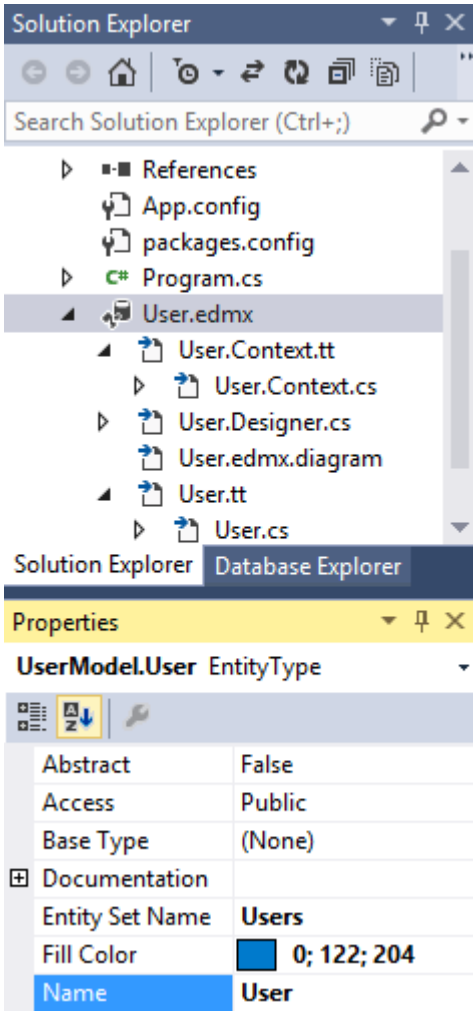
Раскроем узел **Tables**. Он отображает все таблицы, имеющиеся в базе данных. В моем случае имеется только одна таблица **Users**. Отметим все подузлы в ветке **Tables**.

В поле **Model Namespace** установим предпочтительное имя модели и нажмем **Finish**. После этого **Entity Framework** сгенерирует модель по базе данных и добавит ее в проект.

Visiual Studio отобразит нам схему модели. В моем случае в **БД** есть только одна таблица **Users**, поэтому на схеме отображается только одна сущность **User**:



После выделения сущности в правом нижнем углу **Visual Studio** мы увидим свойства для этой сущности:



Свойство **Name** в окне свойств указывает на класс, которым будет представлена данная сущность (то есть классом **User**). А свойство **Entity Set Name** указывает на имя набора объектов (то есть свойство **DbSet** контекста данных) - в данном случае **Users**.

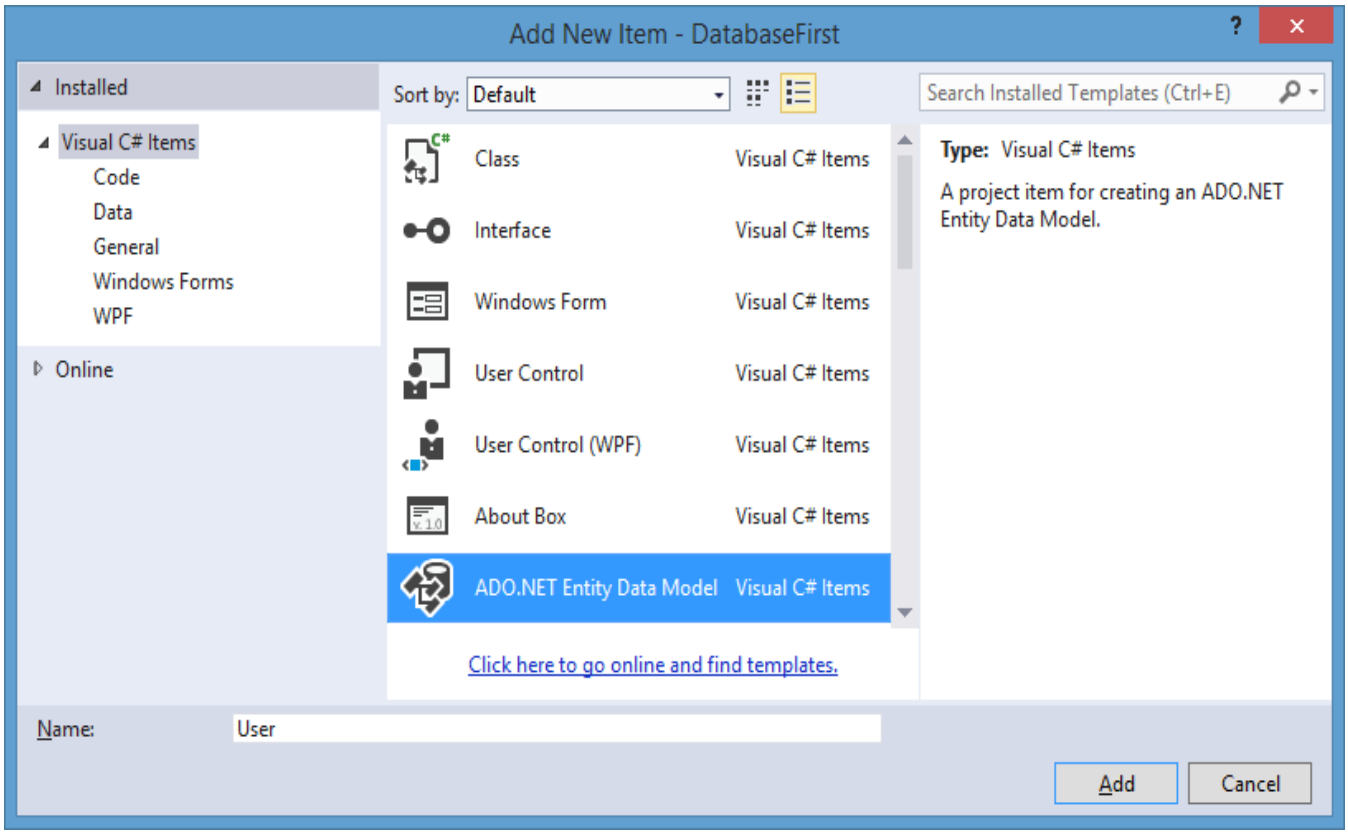
И теперь определим минимальный код для получения данных в коде приложения:

```
using (userstoredbEntities db = new userstoredbEntities())
            {
                var users = db.Users;
                foreach (User u in users)
                    Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name,
u.Age);
            }
```

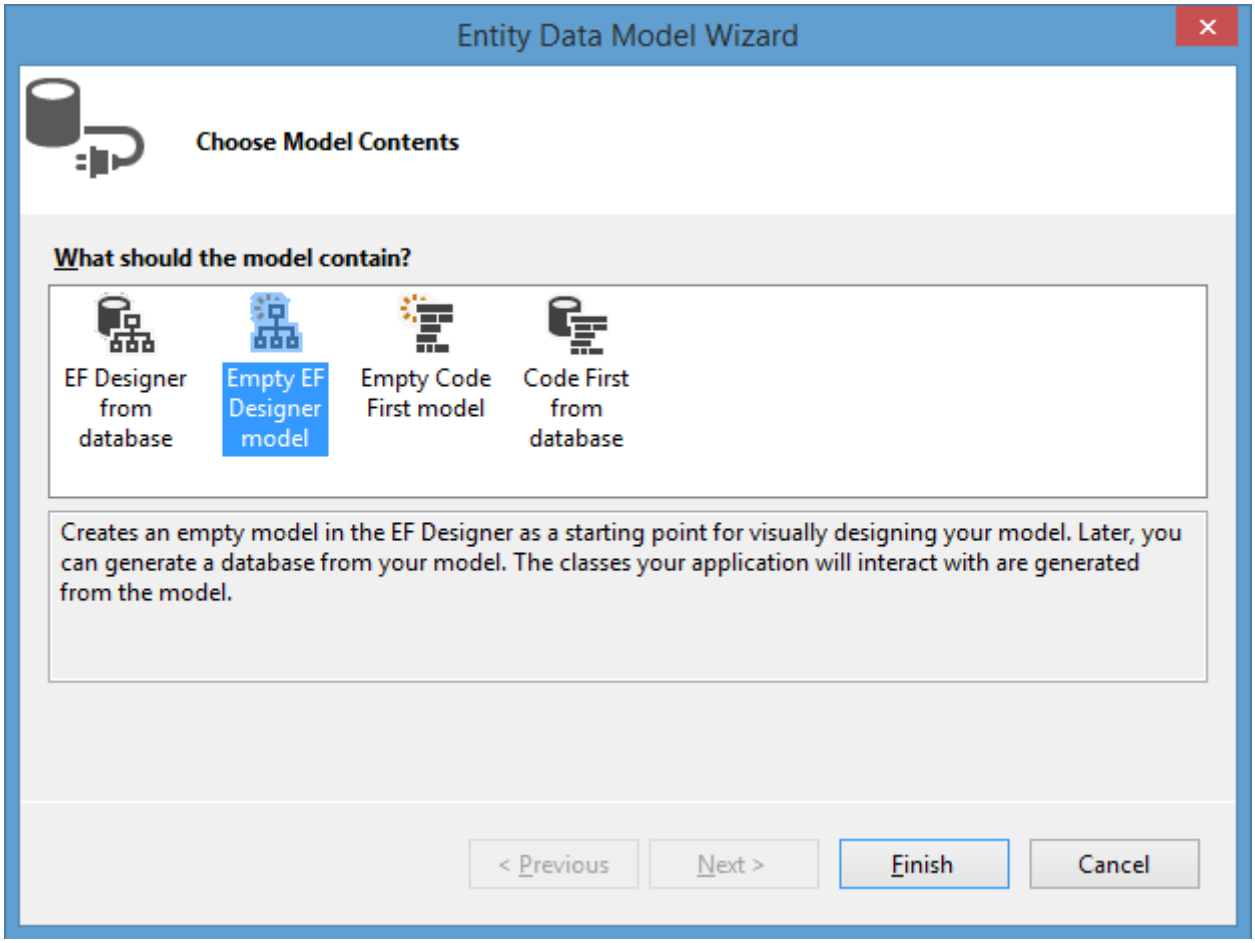
Model First

Model First представляет еще один подход к работе с **Entity Framework**. Суть данного подхода состоит в том, что сначала делается модель, а потом по ней создается база данных.

Итак, создадим новый проект по типу **Console Application**. И затем добавим в проект новый элемент. Нажмем правой кнопкой мыши на проект в окне **Solution Explorer** и в появившемся списке выберем **Add -> New Item**. И затем в окне добавления нового элемента выберем **ADO.NET Entity Data Model**:

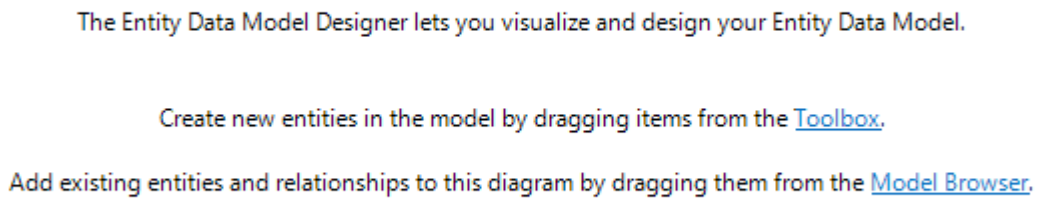


Поскольку модель будет описывать человека, то назовем ее **User**. Нажмем **OK** и нам откроется мастер создания модели. Если у нас **Visual Studio** с пакетами обновления **SP2, SP3**, то мастер создания модели выглядит следующим образом:

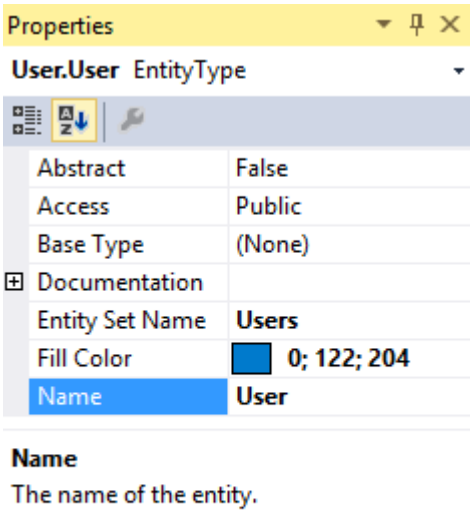


Окно нам предлагает четыре варианта создания модели, из которых нам надо выбрать **Empty EF Designer Model.**

Нажмем кнопку **Finish**, и перед нами откроется пустое окно создания модели.



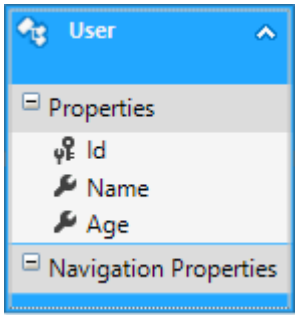
Перетащим на это поле с панели **Toolbox** (**Панель Инструментов**) в левой части элемент **Entity**. Теперь у нас на поле создания модели имеется небольшая схема будущей модели, в которой сейчас по умолчанию указано лишь одно поле - **Id**. Во-первых, переименуем сущность. По умолчанию она называется **Entity1**. Выделим схему и перейдем к окну свойств в правом нижнем углу:



Здесь изменим значение свойства **Name** на **User**. Это у нас будет имя сущности. И также изменим значение свойства **Entity Set Name** на **Users**. Это у нас будет название набора объектов **User**.

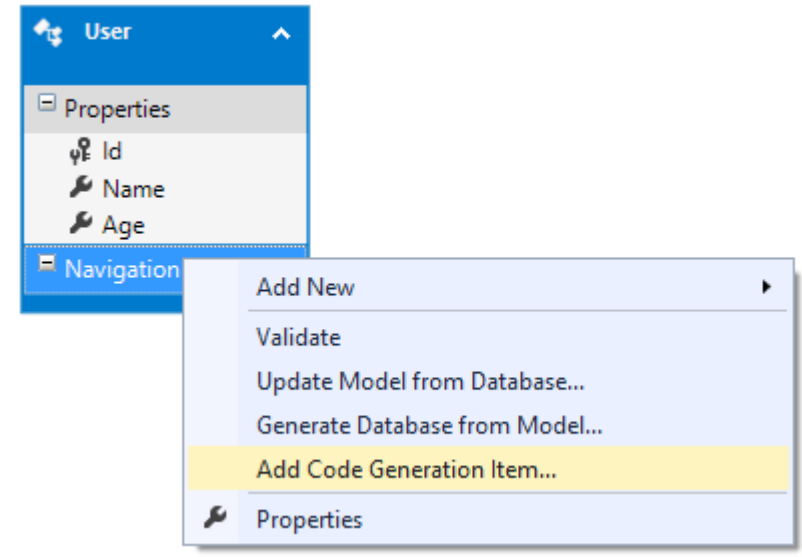
Далее создадим несколько свойств. Сущность у нас будет простая и будет содержать всего два свойства для имени и возраста. Итак, выделим схему сущности и нажмем на правую кнопку мыши. В выпадающем списке выберем **Add New -> Scalar Property**. После этого будет добавлено новое свойство. **Scalar Property** подразумевают свойства на основе простейших типов **int, float, string** и т.д. Добавим два свойства - **Name** и **Age**. По умолчанию все добавляемые свойства имеют тип **string**. Однако мы можем изменить тип в окне свойств.

Таким образом, у нас должна получиться следующая схема сущности:

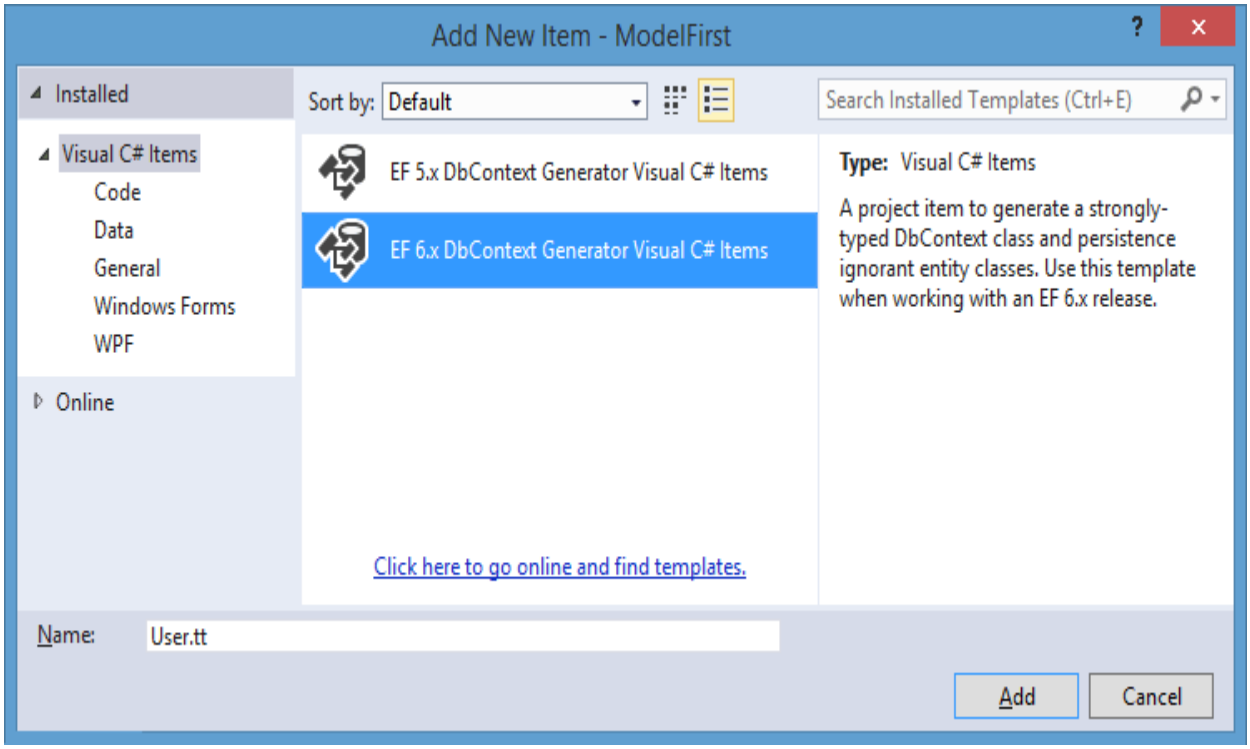


После создания диаграммы модели перестроим проект с помощью опции **Rebuild**.

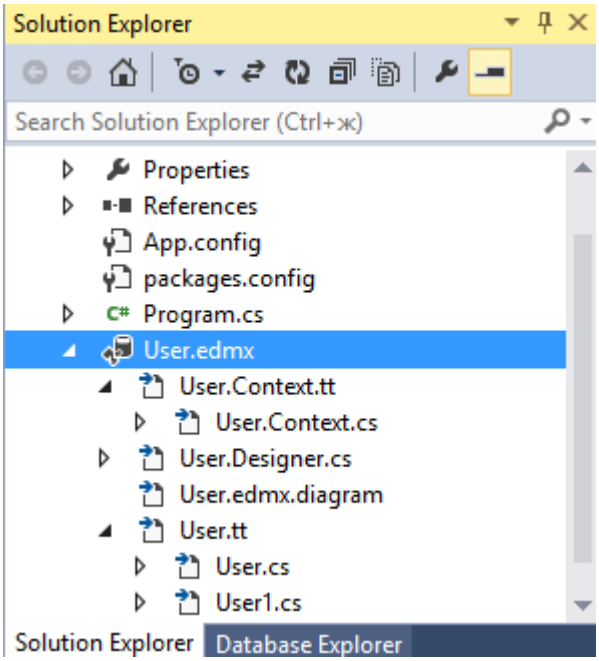
Теперь по модели мы можем сгенерировать код и базу данных. Вначале сгенерируем код модели. Для этого нажмем на диаграмму модели правой кнопкой мыши и выберем пункт **Add Code Generation Item**:



Далее нам будет предложено выбрать версию **EF**. Выберем шестую версию:



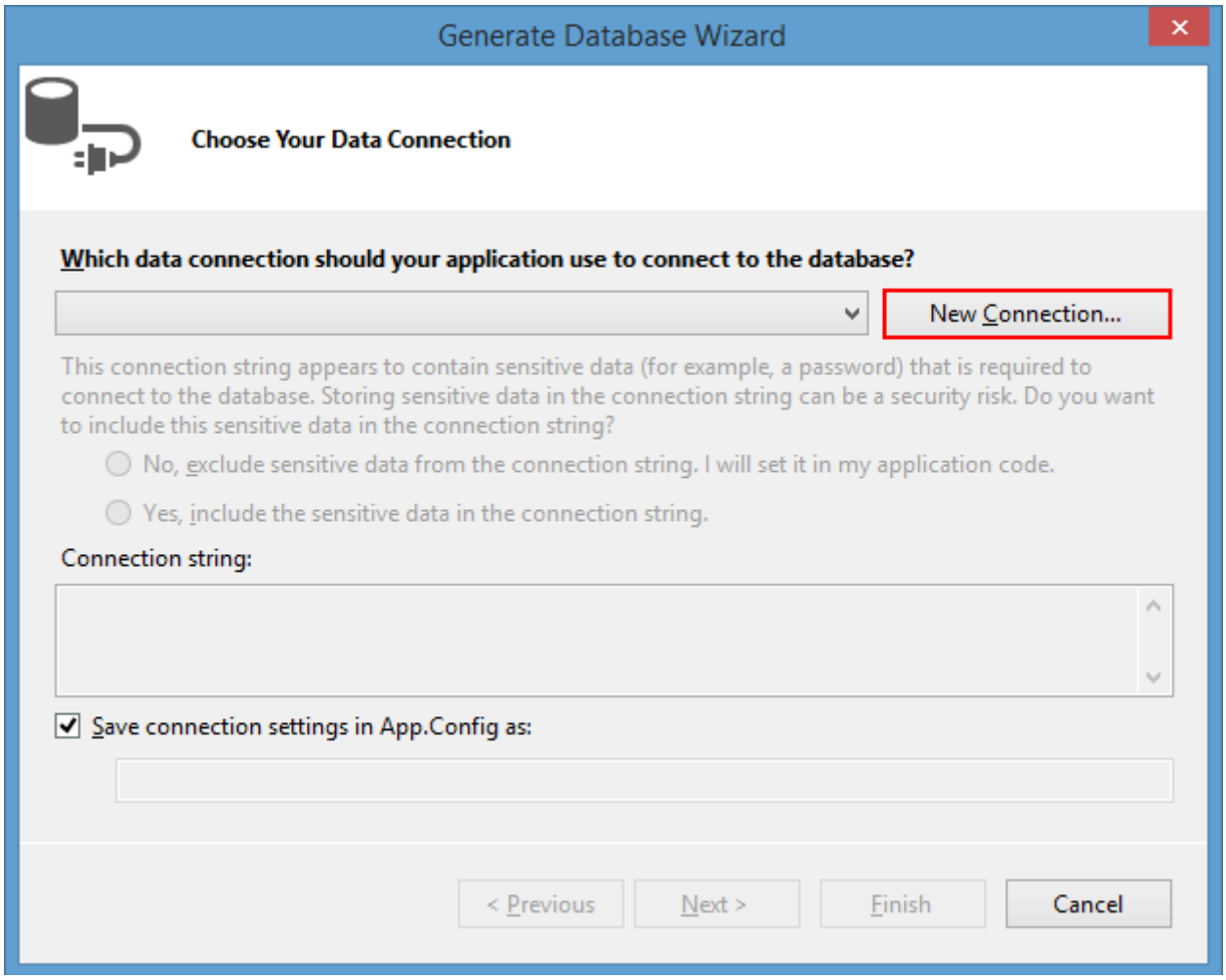
После этого в структуре проекта мы можем увидеть узел **User.tt**, который в качестве подузла будет содержать класс модели в файле **User.cs**:



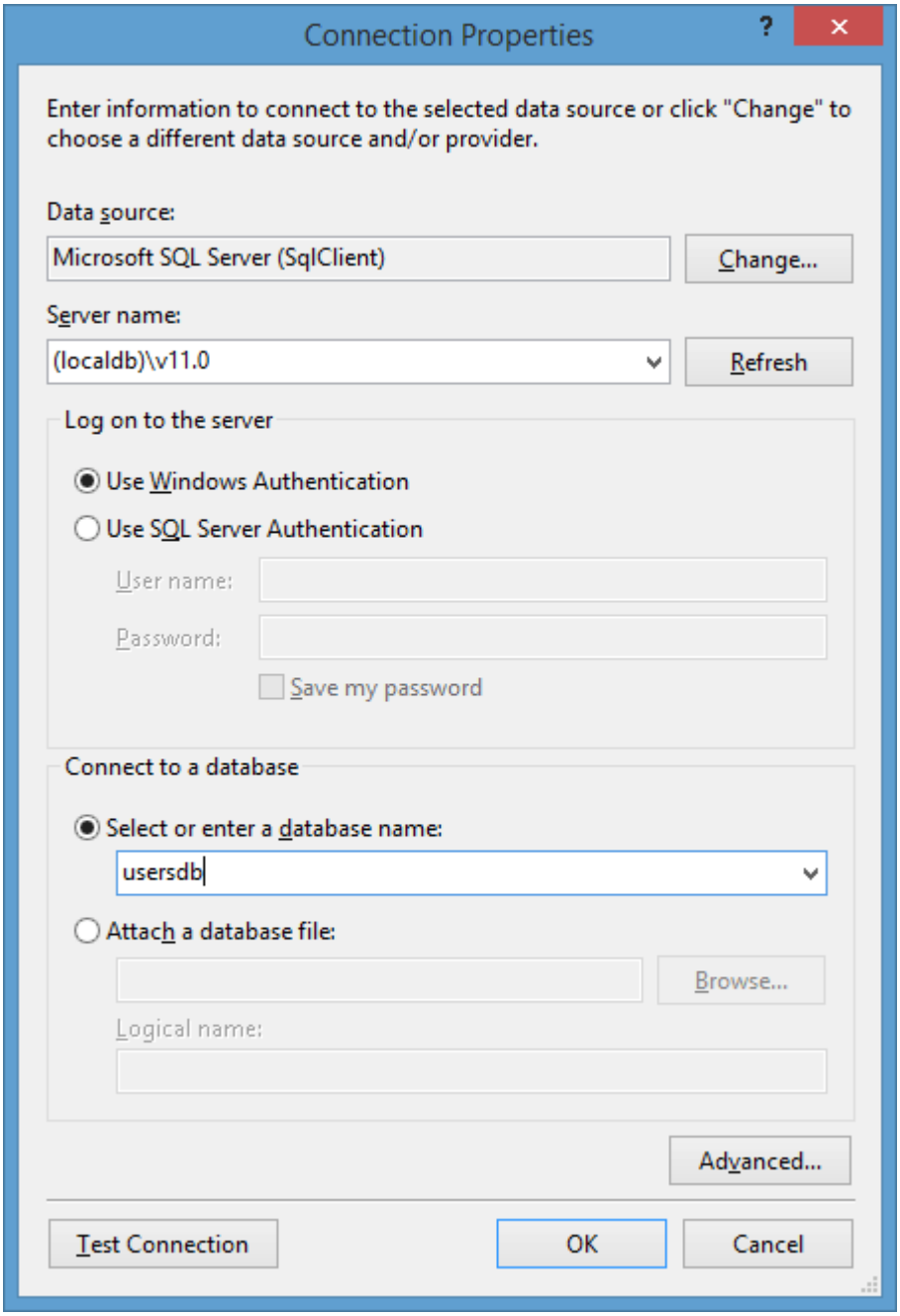
Также здесь мы можем найти файл контекста данных **User.Context.cs**, который выглядит в моем случае следующим образом:

```
namespace ModelFirstApp
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class UserContainer : DbContext
    {
        public UserContainer()
            : base("name=UserContainer")
        {
        }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<User> UserSet { get; set; }
    }
}
```

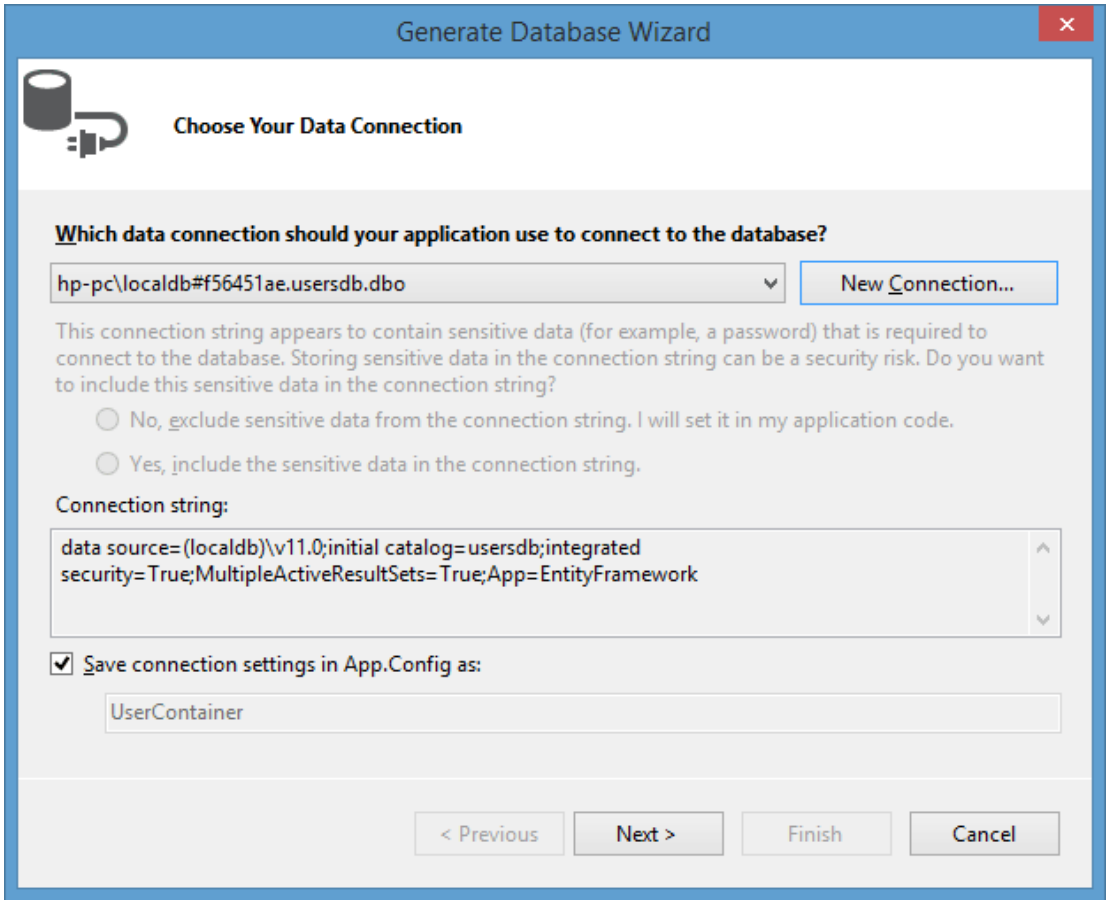
Теперь сгенерируем базу данных по нашей модели. Итак, нажмем на диаграмму модели правой кнопкой мыши и в выпадающем списке выберем **Generate Database from Model** (**Сгенерировать базу данных по модели**). Перед нами откроется мастер создания подключения.



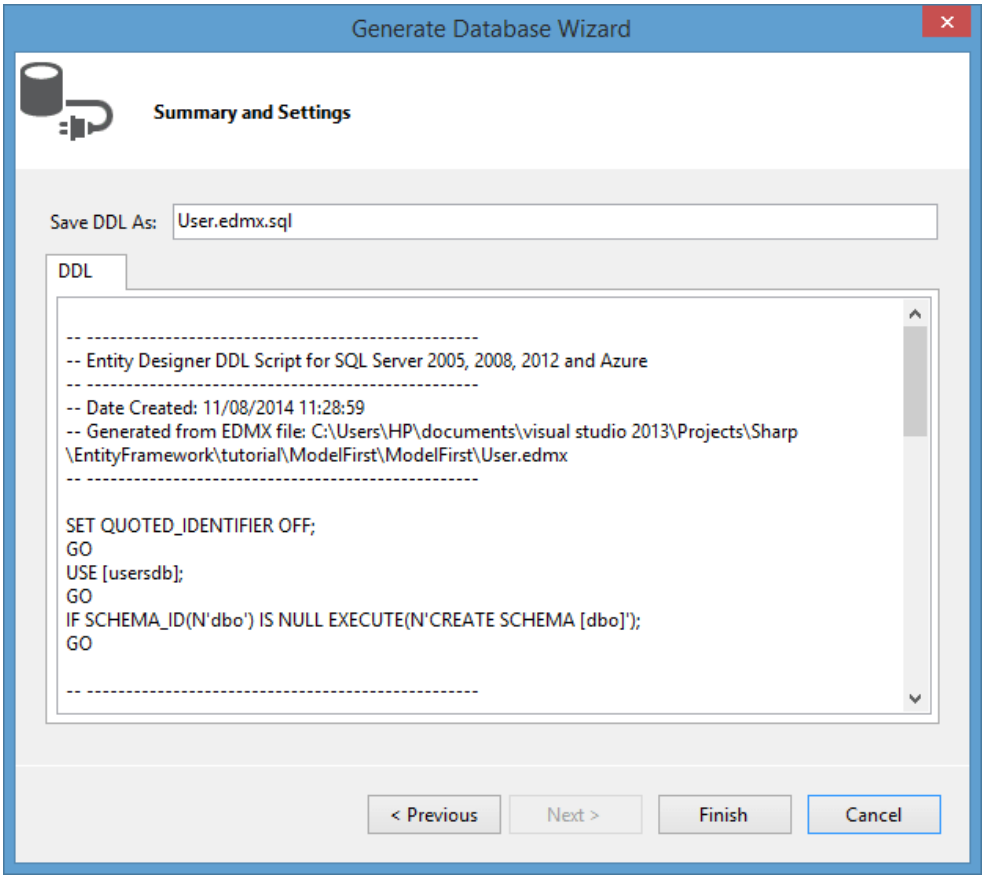
Нажмем на кнопку **New Connection** (**Новое подключение**). Далее открывается новое окно, где будет предложено настроить подключение и создать базу данных:



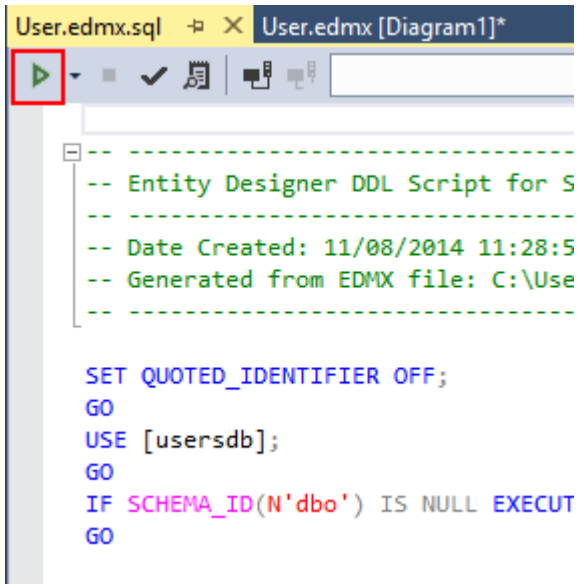
Здесь нам надо ввести имя сервера. А также название **БД**. В качестве имени базы данных введем **usersdb**, а в качестве сервера: **(localdb)\v11.0**. Нажмем **OK** и затем **Visual Studio** установит в качестве подключения модели только что созданную базу данных:



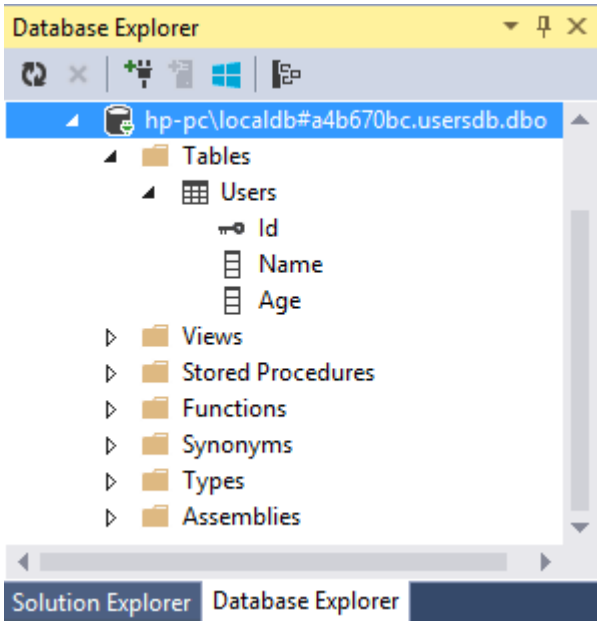
После этого будет сгенерирован скрипт базы данных:



Нажмем **Finish** (**Готово**). У нас автоматически откроется в **Visual Studio** файл скрипта **User.edmx.sql**. И в завершении нам надо будет запустить этот скрипт. Для этого нажмем в верхнем левом углу на зеленую кнопку **Execute** (**Выполнить**):



После этого в нижнем окне **Visual Studio** нам сообщит об успешном (или неуспешном) создании базы данных. Если мы откроем окно **Database Explorer** (его можно открыть, выбрав в меню **View->Other Windows**), то мы увидим нашу базу данных. А раскрыв узел, также увидим, что она содержит всю ту схему, которую мы определили в модели:



Это все была работа по созданию модели и базы данных. И в конце определим минимальный код для работы с базой данных:

```
static void Main(string[] args)
        {
            using (UserContainer db = new UserContainer())
            {
                // добавление элементов
                db.Users.Add(new User { Name = "Tom", Age = 45 });
                db.Users.Add(new User { Name = "John", Age = 22 });
                db.SaveChanges();
                // получение элементов
                var users = db.Users;
                foreach (User u in users)
                    Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name,
u.Age);
            }
            Console.Read();
        }
```

3.Основы Entity Framework

Основные операции с данными

Большинство операций с данными представляют собой **CRUD**-операции (**Create, Read, Update, Delete**), то есть получение данных, создание, обновление и удаление. **Entity Framework** позволяет легко производить данные операции.

Создадим полноценное приложение, которое будет выполнять все эти операции. Итак, создадим новый проект по типу **Windows Forms**. Новое приложение будет работать с базой данных футболистов. В качестве подхода взаимодействия с **БД** выберем **Code First**.

Вначале добавим в проект новый класс, который описывает модель футболистов:

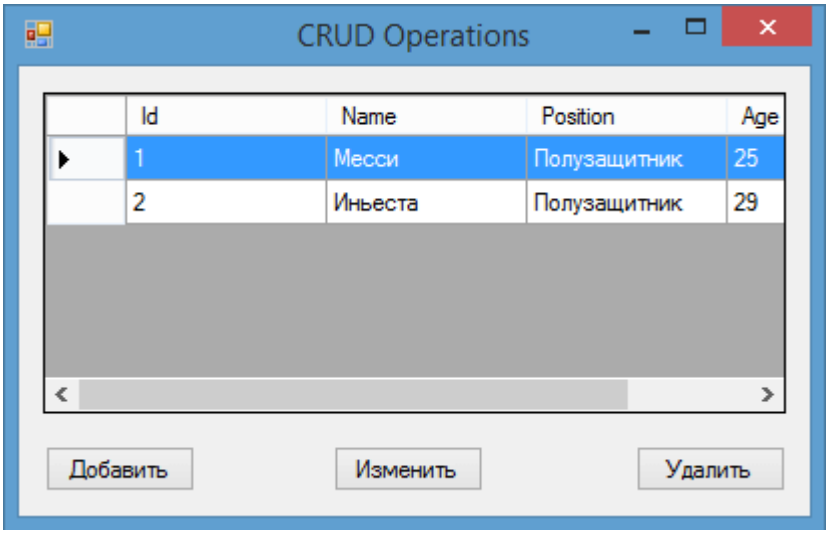
```
class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Position { get; set; }
        public int Age { get; set; }
    }
```

Тут всего четыре свойства: **Id**, имя, позиция на поле и возраст. Также добавим в проект через **NuGet** пакет **Entity Framework** и новый класс контекста данных:

```
using System.Data.Entity;
namespace NewModelFirstApp
{
    class SoccerContext : DbContext
    {
        public SoccerContext()
            : base("DefaultConnection")
        { }

        public DbSet<Player> Players { get; set; }
    }
}
```

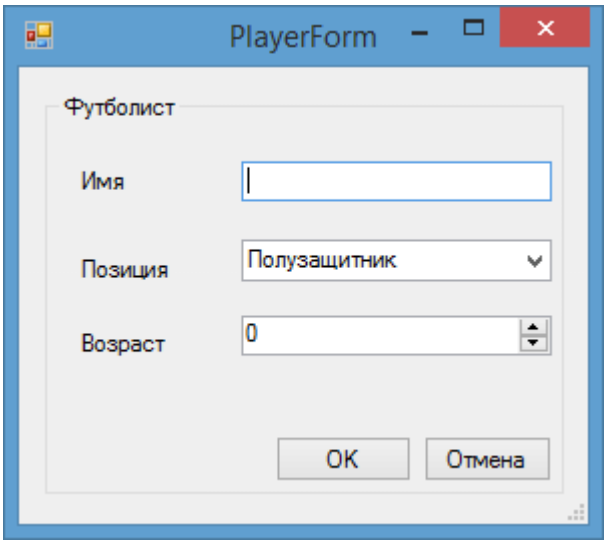
Теперь визуальная часть. По умолчанию в проекте уже есть форма **Form1**. Добавим на нее элемент **DataGridView**, который будет отображать все данные

из **БД**, а также три кнопки на каждое действие - добавление, редактирование, удаление, чтобы в итоге форма выглядела так:



У элемента **DataGridView** установим в окне свойств для свойства **AllowUserToAddRows** значение **False**, а для свойства **SelectionMode** значение **FullRowSelect**, чтобы можно было выделять всю строку.

Это основная форма, но добавление и редактирование объектов у нас будет происходить на вспомогательной форме. Итак, добавим в проект новую форму, которую назовем **PlayerForm**. Она будет иметь следующий вид:



Здесь у нас текстовое поле для ввода имени, далее выпадающий список **ComboBox**, в который мы через свойство **Items** добавляем четыре позиции. И последнее поле - **NumericUpDown** для ввода чисел для указания возраста.

Также есть две кнопки. Для кнопки "**ОК**" в окне свойств для свойства **DialogResult** укажем значение **OK**, а для кнопки "**Отмена**" для того же свойства установим значение **Cancel**.

Никакого кода данная форма не будет содержать. Теперь перейдем к основной форме **Form1**, которая и будет содержать всю логику. Весь ее код:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Data.Entity;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace NewModelFirstApp
{
    public partial class Form1 : Form
    {
        SoccerContext db;
        public Form1()
        {
            InitializeComponent();

            db = new SoccerContext();
            db.Players.Load();

            dataGridView1.DataSource = db.Players.Local.ToBindingList();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        // добавление
        private void button1_Click(object sender, EventArgs e)
        {
            PlayerForm plForm = new PlayerForm();
            DialogResult result = plForm.ShowDialog(this);

            if (result == DialogResult.Cancel)
                return;

            Player player = new Player();
```

```
            player.Age = (int)plForm.numericUpDown1.Value;
            player.Name = plForm.textBox1.Text;
            player.Position = plForm.comboBox1.SelectedItem.ToString();

            db.Players.Add(player);
            db.SaveChanges();

            MessageBox.Show("Yangi yozuv qo'shildi!");
        }

        // удаление
        private void button3_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Player player = db.Players.Find(id);
                db.Players.Remove(player);
                db.SaveChanges();

                MessageBox.Show("Yozuv o'chirildi");
            }
        }

        // редактирование
        private void button2_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Player player = db.Players.Find(id);

                PlayerForm plForm = new PlayerForm();

                plForm.numericUpDown1.Value = player.Age;
                plForm.comboBox1.SelectedItem = player.Position;
                plForm.textBox1.Text = player.Name;
```

```
                DialogResult result = plForm.ShowDialog(this);

                if (result == DialogResult.Cancel)
                    return;

                player.Age = (int)plForm.numericUpDown1.Value;
                player.Name = plForm.textBox1.Text;
                player.Position =
plForm.comboBox1.SelectedItem.ToString();

                db.Entry(player).State = EntityState.Modified;
                db.SaveChanges();

                MessageBox.Show("Yozuv o'zgartirildi");
            }
        }
    }
}
```

Чтобы получить данные из **БД**, используется выражение **db.Players**. Однако нам надо кроме того выполнить привязку к элементу **DataGridView** и динамически отображать все изменения в случае добавления, редактирования или удаления. Поэтому вначале используется метод **db.Players.Load()**, который загружает данные в объект **DbContext**, а затем выполняется привязка **(dataGridView1.DataSource = db.Players.Local.ToBindingList())**

Добавление

При добавлении объекта использует вторая форма:

```
Player player = new Player();
player.Age = (int)plForm.numericUpDown1.Value;
player.Name = plForm.textBox1.Text;
player.Position = plForm.comboBox1.SelectedItem.ToString();

db.Players.Add(player);
db.SaveChanges();
```

Для добавления объекта используется метод **Add**, определенный у класса **DbSet**. В этот метод передается новый объект, свойства которого формируются из полей второй формы. Метод **Add** устанавливает значение **Added** в качестве состояния нового объекта. Поэтому метод **db.SaveChanges()** сгенерирует выражение **INSERT** для вставки модели в таблицу.

Редактирование

Редактирование имеет похожую логику. Только вначале мы передаем значения свойств объекта во вторую форму, а после получаем с нее же измененные значения для свойств объекта.
Чтобы указать, что объект был изменен, нам надо использовать выражение:
db.Entry(player).State = EntityState.Modified;

Это выражение служит указанием методу **db.SaveChanges()**, что надо сформировать **SQL**-выражение **UPDATE** для данного объекта.

Удаление

С удалением проще всего: получаем по **Id** нужный объект в **БД** и передаем его в метод **db.Players.Remove(player)**. Данный метод установит статус объекта в **Deleted**, благодаря чему **Entity Framework** при выполнении метода **db.SaveChanges()** сгенерирует **SQL**-выражение **DELETE**.

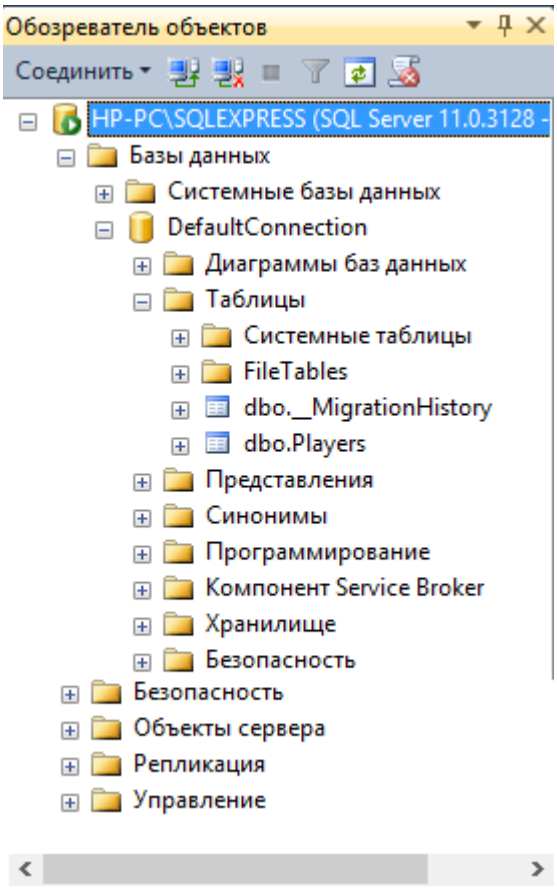
Строка подключения

В предыдущей теме использовался следующий контекст данных:

```
class SoccerContext : DbContext
    {
        public SoccerContext()
            : base("DefaultConnection")
        { }

        public DbSet<Player> Players { get; set; }
    }
```

В конструктор базового класса передается наименование подключения - **DefaultConnection**. И так как в данном случае испльзовался подход **Code First**, то при обращении к базе данных, если ее нет, создавалась новая база данных с названием **DefaultConnection** на **MS SQL Server** е. А после создания все запросы также шли к этой базе данных. Мы можем подключиться к **MS SQL Server** через

Visual Studio или воспользоваться специальной средой **SQL Server Management Studio**, чтобы ее увидеть:



Но в реальности может потребоваться динамически изменять местоположение или название базы данных, к которой идет подключения. В этом случае нам надо указать строку подключения. По умолчанию **Visual Studio** добавляет в проекты файл конфигурации. В проектах для десктопных приложений (**Windows Forms**, консольные приложения и т.д.) этот файл называется **App.config**, в проектах веб-приложений (**ASP.NET**) это файл **Web.config**. Но вне зависимости от типа проекта файл конфигурации имеет определенную структуру и элементы, среди которых также есть также элемент **connectionStrings**, определяющий строки подключения.

Итак, что делать, если мы хотим определить базу данных со случайным именем, которая будет находиться на **MS SQL Server** е? Добавим в файл **App.config** перед закрывающим тегом **configuration** следующий код:

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=.\SQLEXPRESS;Initial Catalog=Players;Integrated Security=True"
         providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Элемент **add** добавляет в секцию **connectionStrings** определение строки подключения к **БД**. Таких элементов **add** можно определить несколько, если мы планируем использовать различные подключения.

Атрибут **name="DefaultConnection"** указывает на название подключения. Имя должно согласоваться с контекстом данных, который использует данное подключение. Так, в конструктор базового класса **(base("DefaultConnection"))** мы передаем название **DefaultConnection**, поэтому и подключение носит такое же название. Но если бы определили бы контекст без вызова конструктора базового класса:

```
class SoccerContext : DbContext
    {
        public SoccerContext()
             { }

        public DbSet<Player> Players { get; set; }
    }
```

В этом случае название подключения должно было бы отражать название контекста: **name="SoccerContext"**

Следующий элемент подключения определяет параметры строки подключения. Этот элемент разбивается на несколько частей:

- **Data Source**: название сервера. Для **MS SQL Express** этот параметр имеет значение **.\SQLEXPRESS**
- **Initial Catalog**: название каталога базы данных. В данном случае **Players**, поэтому при использовании подхода **Code First** на сервере будет созадваться база данных **Players.mdf**
- **Integrated Security**: устанавливает проверку подлинности

И последний элемент устанавливает провайдер:
providerName="System.Data.SqlClient"

И таким образом, в процессе работы приложения будет создана (если еще не существует) и использоваться база данных **Players**.

Строка подключения в Model First и Database First

При использовании подходов **Model First** и **Database First** строка подключения выглядит иначе. Например:

```
<connectionStrings>
```

```
    <add name="persondbEntities"  providerName="System.Data.EntityClient"
     connectionString="metadata=res://*/Person.csdl|res://*/Person.ssdl|
res://*/Person.msl;provider=System.Data.SqlClient;
          provider connection" string="data source=HP-PC\SQLEXPRESS;
          initial catalog=persondb;integrated security=True;
          MultipleActiveResultSets=True;App=EntityFramework"" />
</connectionStrings>
```

Здесь для нас важны следующие параметры строки подключения:

- параметр **metadata**, содержащий метаданные модели (так как в данном случае модель называется **Person**, то метаданные включают ресурсы **Person.csdl, Person.ssdl, Person.msl**)
- параметр **data source** также устанавливает сервер **MS SQL**
- а параметр **initial catalog** также устанавливает каталог базы данных

И в случае изменения базы данных, ее положения, наименования, или использования другой модели, нам надо соответственно изменить эти параметры.

Навигационные свойства и lazy loading

В предыдущей теме в качестве модели был использован класс **Player**, представляющий футболиста и содержащий четыре свойства. Однако эта была очень простая модель. В реальности в нашей базе данных может быть не одна, а несколько таблиц, которые связаны между собой различными связями.

Допустим, для каждого футболиста может быть определена футбольная команда, в которой он играет. И, наоборот, в одной футбольной команде могут играть несколько футболистов. То есть в данном случае у нас связь **один-ко-многим (one-to-many)**.

Например, у нас определен следующий класс футбольной команды **Team**:

```
class Team
    {
        public int Id { get; set; }
        public string Name { get; set; } // название команды
        public string Coach { get; set; } // тренер
    }
```

А класс **Player**, описывающий футболиста, мог бы выглядеть следующим образом:

```
class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Position { get; set; }
        public int Age { get; set; }

        public int? TeamId { get; set; }
        public Team Team { get; set; }
    }
```

Кроме обычных свойств типа **Name, Position** и **Age** здесь также определен внешний ключ. Внешний ключ состоит из обычного свойства и навигационного.

Свойство `public Team Team { get; set; }` в классе **Player** называется **навигационным свойством** - при получении данных об игроке оно будет автоматически получать данные из **БД**.

Вторая часть внешнего ключа - свойство **TeamId**. Чтобы в связке с навигационным свойством образовать внешний ключ оно должно принимать одно из следующих вариантов имени:

- *Имя_навигационного_свойства+Имя ключа из связанной таблицы* - в нашем случае имя навигационного свойства Team, а ключа из модели Team - Id, поэтому в нашем случае нам надо обозвать свойство TeamId, что собственно и было сделано в вышеприведенном коде.
- *Имя_класса_связанной_таблицы+Имя ключа из связанной таблицы* - в нашем случае класс Team, а ключа из модели Team - Id, поэтому опять же в этом случае получается TeamId

Как уже было сказано, внешний ключ позволяет получать связанные данные. Например, после генерации базы данных с помощью **Code First** таблица **Players** будет иметь следующее определение:

```
CREATE TABLE [dbo].[Players] (
    [Id]       INT            IDENTITY (1, 1) NOT NULL,
    [Name]     NVARCHAR (MAX) NULL,
    [Position] NVARCHAR (MAX) NULL,
    [Age]      INT            NOT NULL,
    [TeamId]   INT            NULL,
    CONSTRAINT [PK_dbo.Players] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.Players_dbo.Teams_TeamId] FOREIGN KEY ([TeamId])
REFERENCES [dbo].[Teams] ([Id])
);
```

При определении внешнего ключа нужно иметь в виду следующее. Если тип обычного свойства во внешнем ключе определяется как **int?**, то есть допускает значения **null**, то при создании базы данных соответствующее поле так будет принимать значения **NULL: [TeamId] INT NULL**.

Однако если мы изменим в классе **Player** тип **TeamId** на просто **int: public int TeamId { get; set; }**, то в этом случае соответствующее поле имело бы ограничение **NOT NULL**, а внешний ключ определял бы каскадное удаление:

```
CREATE TABLE [dbo].[Players] (
    [Id]       INT            IDENTITY (1, 1) NOT NULL,
    [Name]     NVARCHAR (MAX) NULL,
    [Position] NVARCHAR (MAX) NULL,
    [Age]      INT            NOT NULL,
    [TeamId]   INT            NOT NULL,
    CONSTRAINT [PK_dbo.Players] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.Players_dbo.Teams_TeamId] FOREIGN KEY ([TeamId])
REFERENCES [dbo].[Teams] ([Id]) ON DELETE CASCADE
);
```

Способы получения связанных данных

Есть различные способы получения связанных данных в **Entity Framework.** Одним из них является так называемая "**жадная загрузка**" или **eager loading**. Ее суть заключается в том, чтобы использовать для подгрузки связанных по внешнему ключу данных метод **Include**. Например, получим всех игроков с их командами:

```
using (SoccerContext db = new SoccerContext())
            {
                IEnumerable<Player> players = db.Players.Include(p =>
p.TeamId);
                foreach (Player p in players)
                {
                    MessageBox.Show(p.Team.Name);
                }
            }
```

Без использования метода **Include** мы бы не могли бы получить связанную команду и ее свойства: **p.Team.Name**

Другой способ представляет так называемая "ленивая загрузка" или **lazy loading**. При таком способе подгрузки при первом обращении к объекту, если связанные данные не нужны, то они не подгружаются. Однако при первом же обращении к навигационному свойству эти данные автоматически подгружаются из **БД**.

При использовании ленивой загрузки надо иметь в виду некоторые моменты при объявлении классов. Так, классы, использующие ленивую загрузку должны быть публичными, а их свойства должны иметь модификаторы **public** и **virtual**. Например, классы **Player** и **Team** могут иметь следующее определение:

```
public class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Position { get; set; }
        public int Age { get; set; }

        public int? TeamId { get; set; }
        public virtual Team Team { get; set; }
    }

public class Team
    {
```

```
        public int Id { get; set; }
        public string Name { get; set; } // название команды
        public string Coach { get; set; } // тренер

        public virtual ICollection<Player> Players { get; set; }

        public Team()
        {
            Players = new List<Player>();
        }
    }
```

Теперь рассмотрим способы связи моделей в приложении с **Entityt Framework**.

Связь один-к-одному

Строго говоря в **Entity Framework** нет как таковой связи один-к-одному, так как ожидается, что обработчик будет использовать связь один-ко-многим. Но все же нередко возникает потребность в наличие подобной связи между объектами в приложении, и в **Entity Framework** мы можем настроить данный тип отношений.

Рассмотрим стандартный пример подобных отношений: есть класс пользователя **User**, который хранит логин и пароль, то есть данные учетных записей. А все данные профиля, такие как имя, возраст и так далее, выделяются в класс профиля **UserProfile**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using System.Data;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace OneToOneApp
{

    public class User
    {
        public int Id { get; set; }
        public string Login { get; set; }
```

```
        public string Password { get; set; }

        public UserProfile Profile { get; set; }
    }

    public class UserProfile
    {
        [Key]
        [ForeignKey("User")]
        public int Id { get; set; }

        public string Name { get; set; }
        public int Age { get; set; }
        public User User { get; set; }
    }
}
```

В этой связи между классами класс **UserProfile** является дочерним или подчиненным по отношению к классу **User**. И чтобы установить связь одни к одному, у подчиненного класса устанавливается свойство идентификатора, которое называется также, как и идентификатор в основном классе. То есть в классе **User** свойство называется **Id**, то и в **UserProfile** также свойство называется **Id**. Если бы в классе **User** свойство называлось бы **UserId**, то такое же название должно было быть и в **UserProfile**.

И в классе **UserProfile** над этим свойством **Id** устанавливаются два атрибута: **[Key]**, который показывает, то это первичный ключ, и **[ForeignKey]**, который показывает, что это также и внешний ключ. Причем внешний ключ к таблице объектов **User**.

Соответственно классы **User** и **UserProfile** имеют ссылки друг на друга.

В классе контекста определяются свойства для взаимодействия с талицами в **БД**:

```
public class UserContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public DbSet<UserProfile> UserProfiles { get; set; }
    }
```

Для этих классов контекст данных будет создавать следующую таблицу **UserProfiles:**

```
CREATE TABLE [dbo].[Users](
```

```
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Login] [nvarchar](max) NULL,
    [Password] [nvarchar](max) NULL,
 CONSTRAINT [PK_dbo.Users] PRIMARY KEY CLUSTERED
)

CREATE TABLE [dbo].[UserProfiles] (
    [Id]   INT            NOT NULL,
    [Name] NVARCHAR (MAX) NULL,
    [Age]  INT            NOT NULL,
    CONSTRAINT [PK_dbo.UserProfiles] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.UserProfiles_dbo.Users_Id] FOREIGN KEY ([Id])
REFERENCES [dbo].[Users] ([Id])
);
```

Посмотрим, как работать с моделями с такой связью.

Добавление и **получение**:

```
static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                User user1 = new User { Login = "login1", Password =
"pass1234" };
                User user2 = new User { Login = "login2", Password =
"5678word2" };
                db.Users.AddRange(new List<User> { user1, user2 });
                db.SaveChanges();
                UserProfile profile1 = new UserProfile { Id = user1.Id,
Age = 22, Name = "Ahmad" };
                UserProfile profile2 = new UserProfile { Id = user2.Id,
Age = 27, Name = "Salim" };
                db.UserProfiles.AddRange(new List<UserProfile> {
profile1, profile2 });
                db.SaveChanges();

                foreach (User user in
db.Users.Include("Profile").ToList())
                    Console.WriteLine("Name: {0}  Age: {1}  Login: {2}
Password: {3}",
                              user.Profile.Name, user.Profile.Age,
user.Login, user.Password);
            }
            Console.ReadLine();
        }
```

Редактирование:

```
static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                User user1 = db.Users.FirstOrDefault();
                if (user1 != null)
                {
                    user1.Password = "dsfvbggg";
                    db.Entry(user1).State = EntityState.Modified;
                    db.SaveChanges();
                }

                UserProfile profile2 = db.UserProfiles.FirstOrDefault(p
=> p.User.Login == "login2");
                if (profile2 != null)
                {
                    profile2.Name = "Alice II";
                    db.Entry(profile2).State = EntityState.Modified;
                    db.SaveChanges();
                }
            }
            Console.ReadLine();
        }
```

При удалении надо учитывать следующее: так как объект **UserProfile** требует наличие объекта **User** и зависит от этого объекта, то при удалении связанного объекта **User** надо будет удалить и связанный с ним объект **UserProfile**. Поскольку по молчанию у нас не предусмотрено каскадное даление при данной связи. Если же будет удален объект **UserProfile**, на объект **User** это никак не повлияет:

```
static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                User user1 =
db.Users.Include("Profile").FirstOrDefault();
                if (user1 != null)
                {
                    db.UserProfiles.Remove(user1.Profile);
                    db.Users.Remove(user1);
                    db.SaveChanges();
                }
```

```
                UserProfile profile2 = db.UserProfiles.FirstOrDefault(p
=> p.User.Login == "login2");
                if (profile2 != null)
                {
                    db.UserProfiles.Remove(profile2);
                    db.SaveChanges();
                }
            }
            Console.ReadLine();
        }
```

Связь один ко многим

Связь один-ко-многим реализуется, если одна модель хранит ссылку на один объект другой модели, а вторая модель может ссылаться на коллекцию объектов первой модели. Например, в одной команде играет несколько игроков:

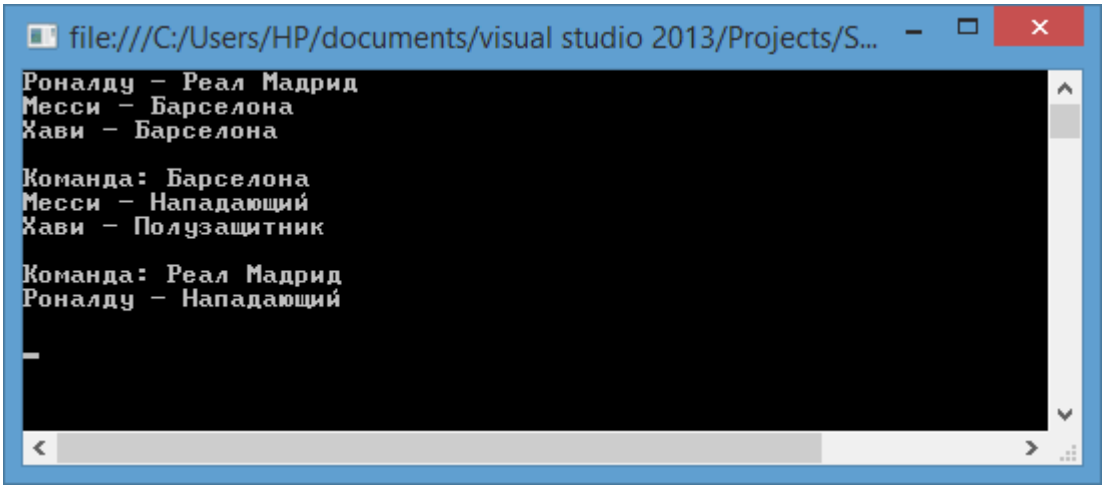
```
class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Position { get; set; }
        public int Age { get; set; }

        public int? TeamId { get; set; }
        public Team Team { get; set; }
    }

class Team
    {
        public int Id { get; set; }
        public string Name { get; set; } // название команды

        public ICollection<Player> Players { get; set; }
        public Team()
        {
            Players = new List<Player>();
        }
    }

class SoccerContext : DbContext
    {
        public SoccerContext()
```

```
            : base("SoccerContext")
        { }

        public DbSet<Player> Players { get; set; }
        public DbSet<Team> Teams { get; set; }
    }
```

Добавление и вывод:

```
using (SoccerContext db = new SoccerContext())
            {
                // создание и добавление моделей
                Team t1 = new Team { Name = "Барселона" };
                Team t2 = new Team { Name = "Реал Мадрид" };
                db.Teams.Add(t1);
                db.Teams.Add(t2);
                db.SaveChanges();
                Player pl1 = new Player { Name = "Роналду", Age = 31,
Position = "Нападающий", Team = t2 };
                Player pl2 = new Player { Name = "Месси", Age = 28,
Position = "Нападающий", Team = t1 };
                Player pl3 = new Player { Name = "Хави", Age = 34,
Position = "Полузащитник", Team = t1 };
                db.Players.AddRange(new List<Player> { pl1, pl2, pl3 });
                db.SaveChanges();

                // вывод
                foreach (Player pl in db.Players.Include(p => p.Team))
                    Console.WriteLine("{0} - {1}", pl.Name, pl.Team !=
null ? pl.Team.Name : "");
                Console.WriteLine();
                foreach (Team t in db.Teams.Include(t => t.Players))
                {
                    Console.WriteLine("Команда: {0}", t.Name);
                    foreach (Player pl in t.Players)
                    {
                        Console.WriteLine("{0} - {1}", pl.Name,
pl.Position);
                    }
                    Console.WriteLine();
                }
            }
```

Результат вывода:



Редактирование:

```
//редактирование
      t2.Name = "Реал М."; // изменим название
      db.Entry(t2).State = EntityState.Modified;
      // переведем игрока из одной команды в другую
      pl3.Team = t2;
      db.Entry(pl3).State = EntityState.Modified;
db.SaveChanges();
```

Удаление:

```
      //удаление игрока
      Player pl_toDelete = db.Players.First(p => p.Name == "Роналду");
      db.Players.Remove(pl_toDelete);
      // удаление команды
      Team t_toDelete = db.Teams.First();
      db.Teams.Remove(t_toDelete);
      db.SaveChanges();
```

При удалении команды свойство **Team** у объектов **Player** получает значение **null**.

Связь один ко многим. Практический пример

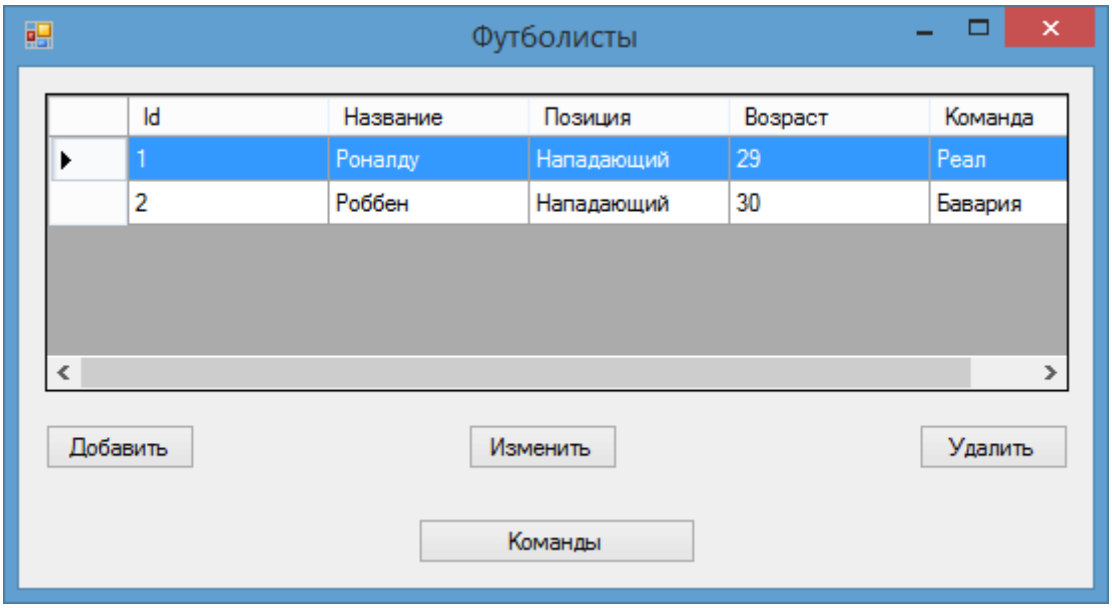
Воспользуемся теоретическим материалом из прошлой темы и создадим новое приложение, в котором будет реализована связь один-ко-многим. Для реализации подобной связи будем использовать **lazy loading**.

Создадим новый проект **Windows Forms**. Первым делом подключим через **NuGet Entity Framework** и добавим все наши модели. Итак, добавим следующие классы:

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

namespace BirKup
{
    public class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Position { get; set; }
        public int Age { get; set; }

        public int? TeamId { get; set; }
        public virtual Team Team { get; set; }
    }

    public class Team
    {
        public int Id { get; set; }
        public string Name { get; set; } // название команды
        public string Coach { get; set; } // тренер

        public virtual ICollection<Player> Players { get; set; }

        public Team()
        {
            Players = new List<Player>();
        }
        public override string ToString()
        {
            return Name;
        }
    }

    class SoccerContext : DbContext
    {
        public SoccerContext()
            : base("SoccerDb")
        { }

        public DbSet<Player> Players { get; set; }
        public DbSet<Team> Teams { get; set; }
    }

}
```

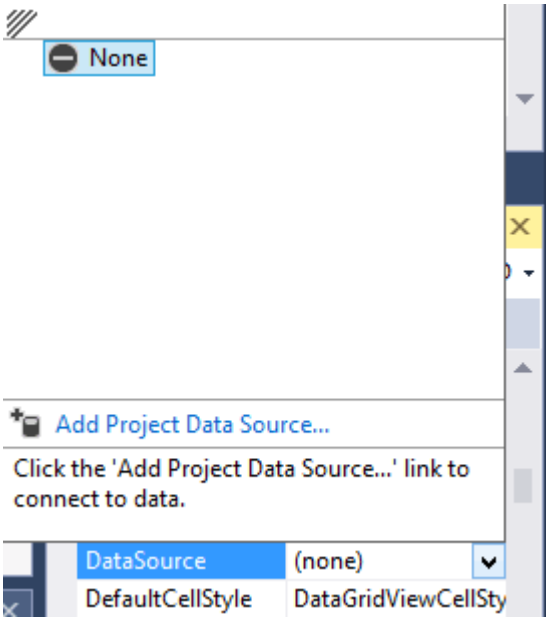
В нашем графическом приложении будет четыре формы: для списка футболистов, для списка команд, для добавления/редактирования футболиста и для добавления/изменения команды.

Пусть форма, которая уже есть по умолчанию, будет представлять футболистов и иметь следующий вид:

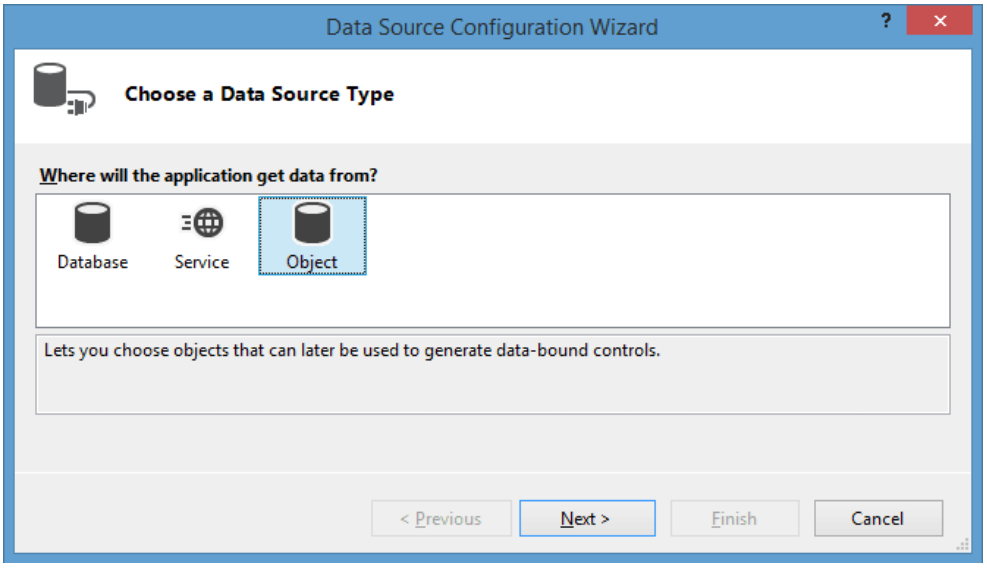


Здесь у нас элемент **DatatGridView** для отображения данных, а также три кнопки для добавления/редактирования/удаления и одна кнопка для вызова окна с футбольными командами.

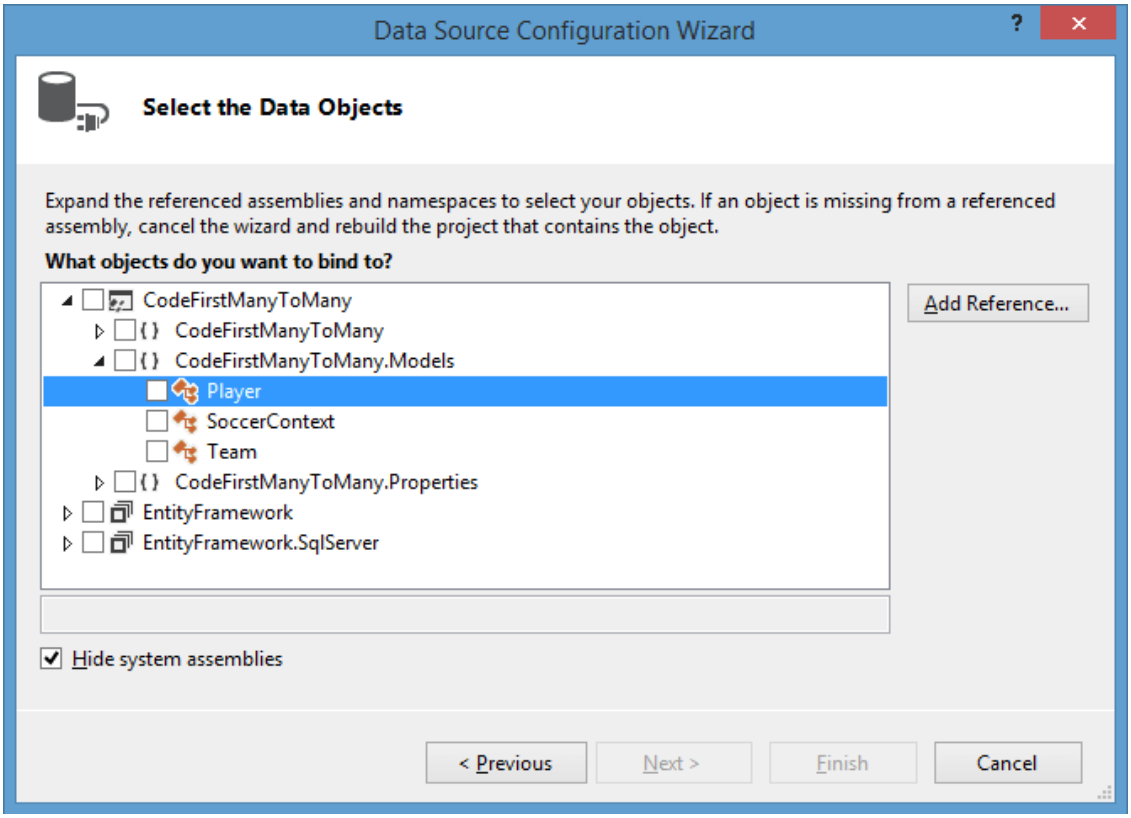
Итак, в предыдущих темах мы уже рассматривали привязку данных к **DataGridView**. При привязке для каждого свойства создается столбец. Однако свойство **TeamId** нам не нужно. Да и было бы неплохо, если бы в качестве заголовков отображались те названия, какие мы хотим, а не названия свойств. Поэтому выделим **DatatGridView** и окне свойств найдем для него свойство **DataSource**. Нажмем, чтобы установить для него значение, и нам отобразится окно выбора источника данных:



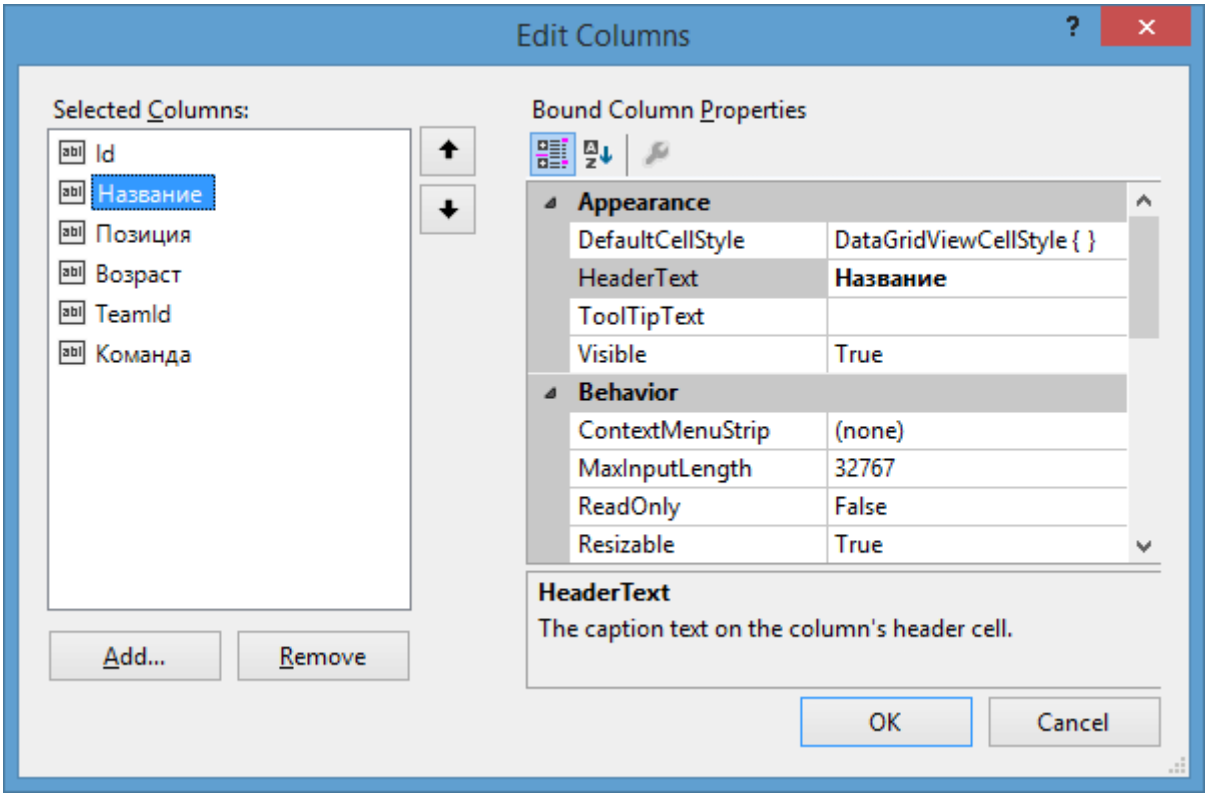
Нажмем на ссылку **Add Project Data Source...**. После этого нам откроется окно мастера настройки источника данных, в котором нам надо выбрать **Object:**



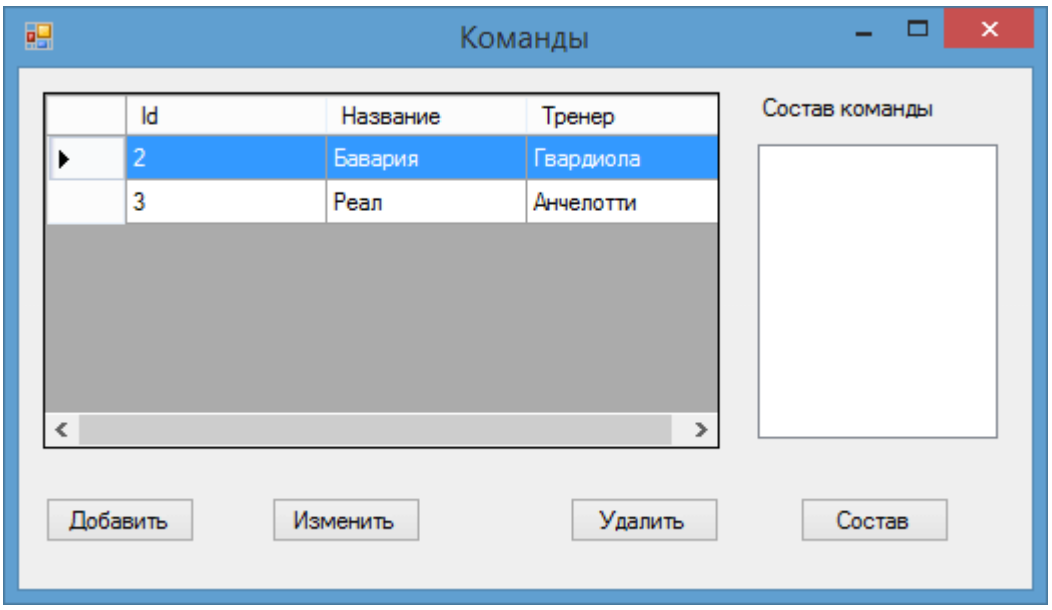
И затем на следующем шаге нам отобразится струкура проекта, в которой в одном из узлов найдем наш класс **Player**:



После этого в **DataGridView** будут добавлены заголовки по именам свойств. Перейдем в свойство **Columns** у **DataGridView**. В свойстве **HeaderText** установим для всех заголовков предпочтительное название, которое будет отображаться, а столбец **TeamId** удалим.

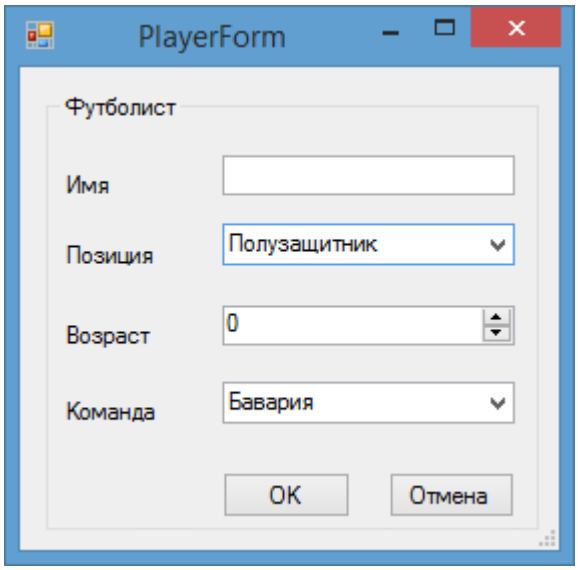


Подобным образом сделаем графический интерфейс и для формы с командами, назовем ее, к примеру, **AllTeams**:



Здесь также **DataGridView** для отображения списка команд, а также поле **ListBox** и кнопка '**Состав**' для вывода в этом поле всех игроков выбранной команды.

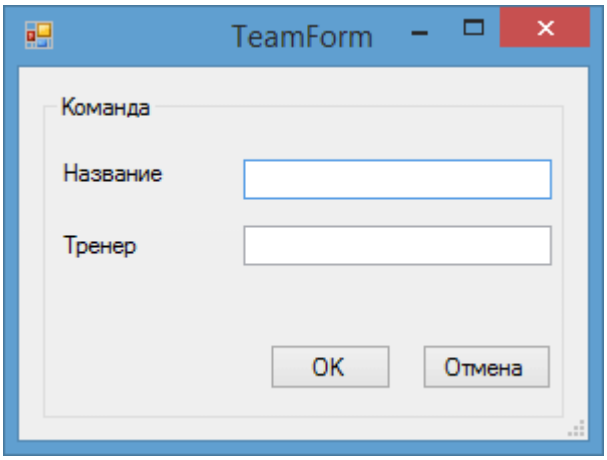
Добавим также дополнительные формы для создания и изменения игрока и команды. Форма для игрока, назовем ее **PlayerForm**:



Здесь текстовое поле для имени игрока, элемент **NumericUpDown** для указания возраста, и два элемента **ComboBox**.

Для кнопки '**OK**' у свойства **DialogResult** установим значение **OK**, а у кнокпки '**Отмена**' установим значение **Cancel**. И изменим у всех полей значение свойства **Modifiers** с **Private** на **Protected Internal**.

И форма для создания команды **TeamForm:**



Для кнопок и полей также настроим свойство **DialogResult** и **Modifiers**, как и у предыдущей формы.

Код главной формы с игроками:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.Entity;

namespace BirKups
{
    public partial class Form1 : Form
    {
        SoccerContext db;
        public Form1()
        {
            InitializeComponent();
            db = new SoccerContext();
            db.Players.Load();
            dataGridView1.DataSource = db.Players.Local.ToBindingList();
        }

        private void button1_Click(object sender, EventArgs e)
```

```
        {
            PlayerForm plForm = new PlayerForm();
            // из команд в бд формируем список
            List<Team> teams = db.Teams.ToList();
            plForm.comboBox2.DataSource = teams;
            plForm.comboBox2.ValueMember = "Id";
            plForm.comboBox2.DisplayMember = "Name";

            DialogResult result = plForm.ShowDialog(this);

            if (result == DialogResult.Cancel)
                return;

            Player player = new Player();
            player.Age = (int)plForm.numericUpDown1.Value;
            player.Name = plForm.textBox1.Text;
            player.Position = plForm.comboBox1.SelectedItem.ToString();
            player.Team = (Team)plForm.comboBox2.SelectedItem;

            db.Players.Add(player);
            db.SaveChanges();

            MessageBox.Show("Новый футболист добавлен");

        }

        private void button2_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Player player = db.Players.Find(id);

                PlayerForm plForm = new PlayerForm();
                plForm.numericUpDown1.Value = player.Age;
                plForm.comboBox1.SelectedItem = player.Position;
                plForm.textBox1.Text = player.Name;

                List<Team> teams = db.Teams.ToList();
                plForm.comboBox2.DataSource = teams;
                plForm.comboBox2.ValueMember = "Id";
                plForm.comboBox2.DisplayMember = "Name";
```

```
                if (player.Team != null)
                    plForm.comboBox2.SelectedValue = player.Team.Id;

                DialogResult result = plForm.ShowDialog(this);

                if (result == DialogResult.Cancel)
                    return;

                player.Age = (int)plForm.numericUpDown1.Value;
                player.Name = plForm.textBox1.Text;
                player.Position =
plForm.comboBox1.SelectedItem.ToString();
                player.Team = (Team)plForm.comboBox2.SelectedItem;

                db.Entry(player).State = EntityState.Modified;
                db.SaveChanges();

                MessageBox.Show("Объект обновлен");
            }

        }

        private void button3_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Player player = db.Players.Find(id);
                db.Players.Remove(player);
                db.SaveChanges();

                MessageBox.Show("Объект удален");
            }
        }

        private void button4_Click(object sender, EventArgs e)
        {
            AllTeams teams = new AllTeams();
            teams.Show();
        }
    }
}
```

Тут тот же самый функционал, что рассматривался на примере простого приложения в одной из предыдущих тем. Только появляется дополнительное поле для выбора команды, в которое нам сначала надо загрузить все команды и настроить привязку:

```
List<Team> teams = db.Teams.ToList();
plForm.comboBox2.DataSource = teams;
plForm.comboBox2.ValueMember = "Id";
plForm.comboBox2.DisplayMember = "Name";
```

А при редактировании мы устанавливаем для этого поля значение, равное **TeamId: plForm.comboBox2.SelectedValue = player.Team.Id**.

Здесь благодаря **lazy loading** мы можем получить связанную с игроком команду и обратиться к ее свойствам.

Код формы команд:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.Entity;

namespace BirKups
{
    public partial class AllTeams : Form
    {
        SoccerContext db;
        public AllTeams()
        {
            InitializeComponent();
            db = new SoccerContext();
            db.Teams.Load();
            dataGridView1.DataSource = db.Teams.Local.ToBindingList();
        }

        private void dataGridView1_CellContentClick(object sender,
DataGridViewCellEventArgs e)
        {
            TeamForm tmForm = new TeamForm();
            DialogResult result = tmForm.ShowDialog(this);
```

```
            if (result == DialogResult.Cancel)
                return;

            Team team = new Team();
            team.Name = tmForm.textBox1.Text;
            team.Coach = tmForm.textBox2.Text;

            db.Teams.Add(team);
            db.SaveChanges();
            MessageBox.Show("Новый объект добавлен");

        }

        private void button1_Click(object sender, EventArgs e)
        {

        }

        private void button4_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Team team = db.Teams.Find(id);
                listBox1.DataSource = team.Players.ToList();
                listBox1.DisplayMember = "Name";
            }

        }

        private void button3_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Team team = db.Teams.Find(id);
```

```
                team.Players.Clear();
                db.Teams.Remove(team);
                db.SaveChanges();

                MessageBox.Show("Объект удален");
            }

        }

        private void button2_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count > 0)
            {
                int index = dataGridView1.SelectedRows[0].Index;
                int id = 0;
                bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
                if (converted == false)
                    return;

                Team team = db.Teams.Find(id);

                TeamForm tmForm = new TeamForm();
                tmForm.textBox1.Text = team.Name;
                tmForm.textBox2.Text = team.Coach;

                DialogResult result = tmForm.ShowDialog(this);
                if (result == DialogResult.Cancel)
                    return;

                team.Name = tmForm.textBox1.Text;
                team.Coach = tmForm.textBox2.Text;

                db.Entry(team).State = EntityState.Modified;
                db.SaveChanges();
                MessageBox.Show("Объект обновлен");
            }

        }
    }
}
```

На что здесь надо обратить внимание? Во-первых, здесь так же благодаря **lazy loading** мы можем получить связанных с командой игроков и загрузить их в **ListBox: listBox1.DataSource = team.Players.ToList();**

Во-вторых, при удалении мы предварительно очищаем данный список: **team.Players.Clear();**.

Если бы мы вручную создавали базу данных, а потом через **entity framework** через **database first** или **code first** подключали бы к проекту, то мы могли бы не очищать список, установив при создании внешнего ключа в **БД** каскадное удаление или установку в **null** поля игрока при удалении связанной команды.

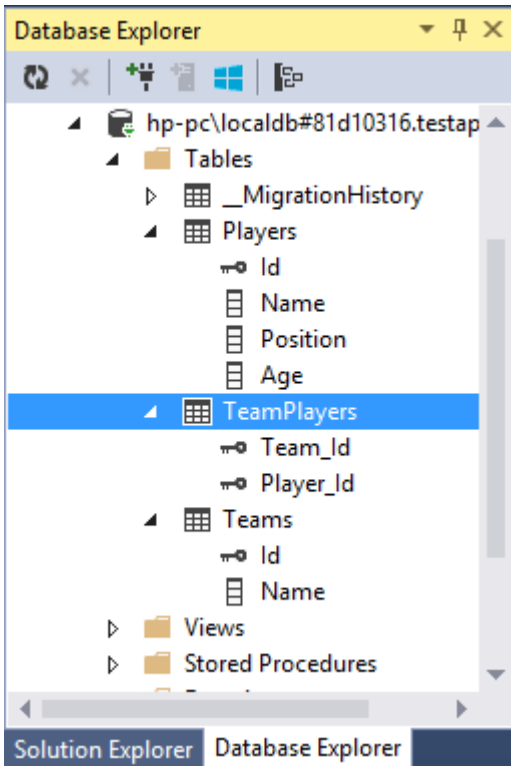
Связь многие ко многим

Еще одним способом ассоциации объектов является связь многие-ко-многим. Например, у нас есть модель футболистов и есть модель команд. На протяжении всей жизни футболист может поиграть в различных командах, а в одной команде может поиграть множество разных футболистов. На уровне моделей это выглядит так:

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

        public ICollection<Player> Players { get; set; }
        public Team()
        {
            Players = new List<Player>();
        }
    }

public class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }

        public ICollection<Team> Teams { get; set; }
        public Player()
    {
        Teams = new List<Team>();
      }
    }
```

Обе модели имеют свойства-коллекции, через которые и будет осуществляться связь многие-ко-многим. В итоге, если мы используем

CodeFirst, то автоматически будет создаваться база данных со следующей схемой:



То есть создается промежуточная таблица, которая хранит наборы пар **Player-Team**. И если бы мы использовали подход **Database First**, то нам надо было также создать эту таблицу. Используем модели.

Добавление и вывод:

```
using (SoccerContext db = new SoccerContext())
            {
                // создание и добавление моделей
                Player pl1 = new Player { Name = "Роналду", Age = 31,
Position = "Нападающий" };
                Player pl2 = new Player { Name = "Месси", Age = 28,
Position = "Нападающий" };
                Player pl3 = new Player { Name = "Хави", Age = 34,
Position = "Полузащитник" };
                db.Players.AddRange(new List<Player> { pl1, pl2, pl3 });
                db.SaveChanges();

                Team t1 = new Team { Name = "Барселона" };
                t1.Players.Add(pl2);
                t1.Players.Add(pl3);
                Team t2 = new Team { Name = "Реал Мадрид" };
                t2.Players.Add(pl1);
                db.Teams.Add(t1);
                db.Teams.Add(t2);
                db.SaveChanges();
```

```
            foreach (Team t in db.Teams.Include(t => t.Players))
            {
                Console.WriteLine("Команда: {0}", t.Name);
                foreach (Player pl in t.Players)
                {
                    Console.WriteLine("{0} - {1}", pl.Name,
pl.Position);
                }
                Console.WriteLine();
            }
        }
```

При добавление одной модели в список к другой важно помнить, что это список уже должен быть создан, иначе будет выброшено исключение. В данном случае мы создаем список в конструкторе обоих моделей. Также допустимо создание списка непосредственно в программе.

Редактирование:

```
// удаляем связи с одним объектом
Player pl_edit = db.Players.First(p => p.Name == "Месси");
Team t_edit = pl_edit.Teams.First(p => p.Name == "Барселона");
t_edit.Players.Remove(pl_edit);
```

Удаление игрока из списка команды будет означать удаление строки из таблицы **TeamPlayers**, в которой **Id** игрока сопоставляется **Id** команды.

Удаление же игрока или команды вообще из базы данных приводит к тому, что все строки в таблице **TeamPlayers**, которые содержат **Id** удаленного объекта, также будут удалены:

```
Player pl_delete = db.Players.First(p => p.Name == "Месси");
db.Players.Remove(pl_delete);
```

Связь многие ко многим. Практический пример

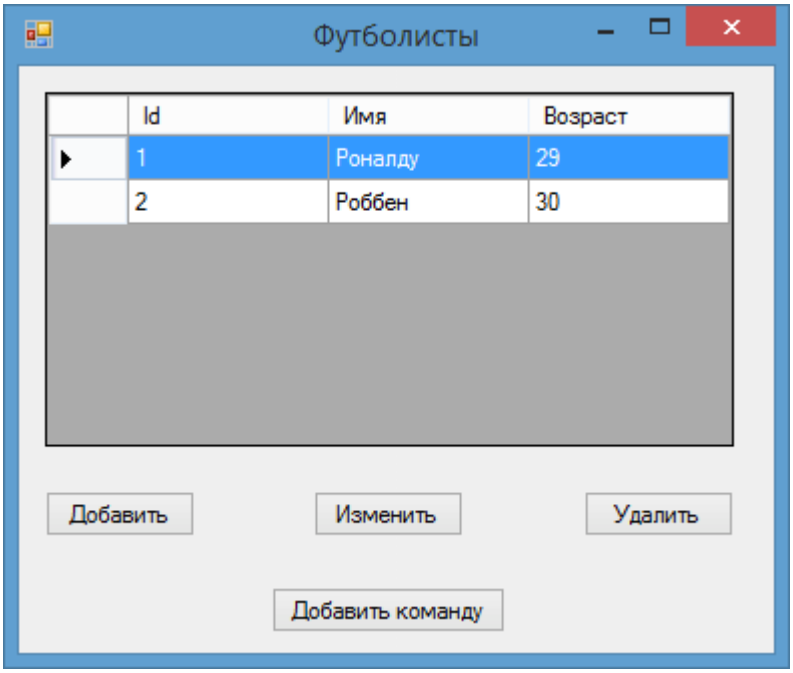
Итак, создадим новый проект по типу **Windows Forms** и добавим в него эти модели. После этого также добавим через **NuGet** пакет **Entity Framework** и следующий класс контекста данных:

```
class SoccerContext : DbContext
    {
        public SoccerContext()
            : base("SoccerDB2")
        { }

        public DbSet<Team> Teams { get; set; }
        public DbSet<Player> Players { get; set; }
    }
```

И определим модели:

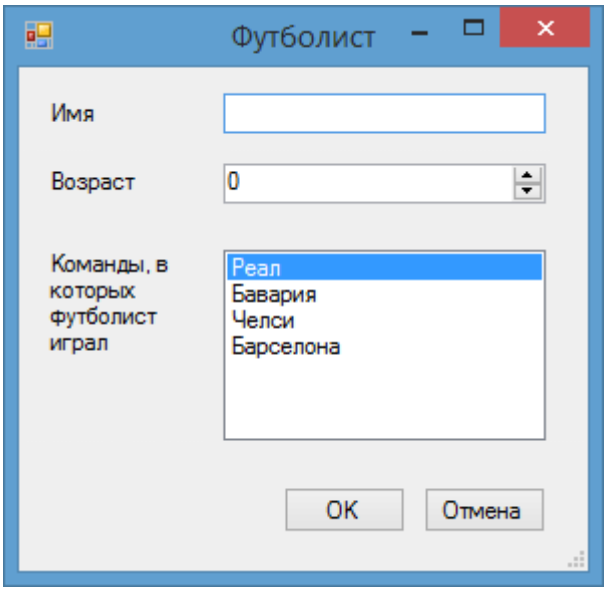
```
class Team
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Coach { get; set; }

        public virtual ICollection<Player> Players { get; set; }
        public Team()
        {
            Players = new List<Player>();
        }
    }

class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }

        public virtual ICollection<Team> Teams { get; set; }
        public Player()
        {
            Teams = new List<Team>();
        }
    }
```

Теперь само приложение. На главной форме у нас будет в таблице выводиться список игроков:



Основной функционал будет таким же, как и в прошлой теме: один элемент **DataGridView** и четыре кнопки. Кроме того нам потребуется две дополнительные формы: одна для создания команды, а другая - для создания/редактирования игрока.

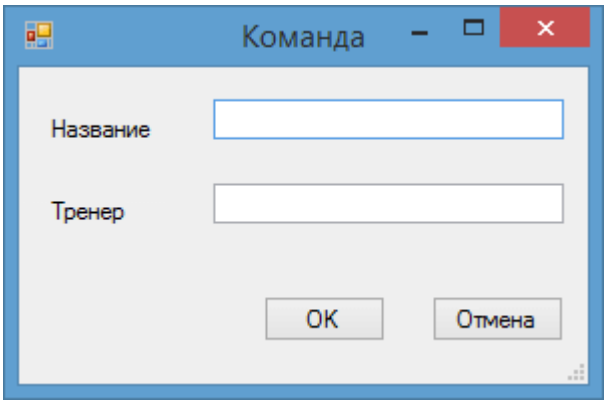
Добавим в проект новую форму. Назовем ее, к примеру, **PlayerForm**:



Здесь текстовое поле для указания имени, поле **NumericUpDown** для возраста и элемент **ListBox**, в котором будут выводиться команды. Установим у этих элементов значение свойства **Modifier** равным **Protected Internal**.

Также есть две кнопки. Установим у кнопки '**OK**' значение свойства **DialogResult** равным **OK**, а у кнопки '**Отмена**' - равным **Cancel.**

Форма для добавления команд, назовем ее **TeamForm**, будет выглядеть следующим образом:



Произведем у этой формы ту же настройку с полями и кнопками, как и у предыдущей формы.

Теперь изменим код главной формы, которая отображает игроков, на следующий:

Код формы команд:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Data.Entity;
using System.Windows.Forms;

namespace KupKups
{
    public partial class Form1 : Form
    {
        SoccerContext db;
        public Form1()
        {
            InitializeComponent();

            db = new SoccerContext();
            db.Players.Load();
            dataGridView1.DataSource = db.Players.Local.ToBindingList();
        }
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            PlayerForm plForm = new PlayerForm();

            // добавляем список команд в на форму plForm
            List<Team> teams = db.Teams.ToList();
            plForm.listBox1.DataSource = teams;
            plForm.listBox1.ValueMember = "Id";
            plForm.listBox1.DisplayMember = "Name";

            DialogResult result = plForm.ShowDialog(this);
            if (result == DialogResult.Cancel)
                return;

            Player player = new Player();
            player.Age = (int)plForm.numericUpDown1.Value;
            player.Name = plForm.textBox1.Text;

            teams.Clear(); // очищаем список и заново заполняем его
выделенными элементами
            foreach (var item in plForm.listBox1.SelectedItems)
            {
                teams.Add((Team)item);
            }
            player.Teams = teams;
            db.Players.Add(player);
            db.SaveChanges();

            MessageBox.Show("Новый игрок добавлен");
        }

        private void button4_Click(object sender, EventArgs e)
        {
            TeamForm tmForm = new TeamForm();

            DialogResult result = tmForm.ShowDialog(this);
            if (result == DialogResult.Cancel)
                return;

            Team team = new Team();
            team.Name = tmForm.textBox1.Text;
            team.Coach = tmForm.textBox2.Text;
            team.Players = new List<Player>();

            db.Teams.Add(team);
            db.SaveChanges();
            MessageBox.Show("Новая команда добавлена");

        }
```

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count < 1)
                return;

            int index = dataGridView1.SelectedRows[0].Index;
            int id = 0;
            bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
            if (converted == false)
                return;

            Player player = db.Players.Find(id);

            PlayerForm plForm = new PlayerForm();
            plForm.numericUpDown1.Value = player.Age;
            plForm.textBox1.Text = player.Name;

            // получаем список команд
            List<Team> teams = db.Teams.ToList();
            plForm.listBox1.DataSource = teams;
            plForm.listBox1.ValueMember = "Id";
            plForm.listBox1.DisplayMember = "Name";
            foreach (Team t in player.Teams)
                plForm.listBox1.SelectedItem = t;

            DialogResult result = plForm.ShowDialog(this);
            if (result == DialogResult.Cancel)
                return;

            player.Age = (int)plForm.numericUpDown1.Value;
            player.Name = plForm.textBox1.Text;

            // проверяем наличие команд у игрока
            foreach (var team in teams)
            {
                if (plForm.listBox1.SelectedItems.Contains(team))
                {
                    if (!player.Teams.Contains(team))
                        player.Teams.Add(team);
                }
                else
                {
                    if (player.Teams.Contains(team))
                        player.Teams.Remove(team);
                }
            }
            db.Entry(player).State = EntityState.Modified;
```

```
            db.SaveChanges();
            MessageBox.Show("Информация обновлена");

        }

        private void button3_Click(object sender, EventArgs e)
        {
            if (dataGridView1.SelectedRows.Count < 1)
                return;

            int index = dataGridView1.SelectedRows[0].Index;
            int id = 0;
            bool converted = Int32.TryParse(dataGridView1[0,
index].Value.ToString(), out id);
            if (converted == false)
                return;

            Player player = db.Players.Find(id);

            db.Players.Remove(player);
            db.SaveChanges();

            MessageBox.Show("Футболист удален");

        }
    }
}
```

Инициализация базы данных

Если нам необходимо, чтобы при первом обращении база данных уже была заполнена некоторыми начальными значениями, то мы можем произвести ее инициализацию.

Инициализация происходит при первом обращении к контексту данных. Для инициализации мы можем использовать один из классов инициализаторов, котоыре имеются в библиотеке **.NET**:

- **CreateDatabaseIfNotExists**: инициализатор, используемый по умолчанию. Он не удаляет автоматчески базу данных и данные, а в случае изменения структуры моделей и контекста данных выбрасывает исключение.
- **DropCreateDatabaseWhenModelChanges**: данный инициализатор проверяет на соответствие моделям определения таблиц в базе данных. И если модели не соответствуют определению таблиц, то база данные пересоздается

- **DropCreateDatabaseAlways**: этот инициализатор будет всегда пересоздавать базу данных.

Используем один из инициализаторов. Для этого нам надо переопределить метод **Seed**:

1) Phones.cs

```
using System.Data.Entity;

namespace Init
{
    class Phone
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Price { get; set; }
    }
}
```

2) MobileContext.cs

```
using System.Data.Entity;

namespace Init
{
    class MobileContext : DbContext
    {
        static MobileContext()
        {
            Database.SetInitializer<MobileContext>(new
MyContextInitializer());
        }

        public MobileContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }
    }
}
```

3) MyContextInitializer.cs

```
using System.Data.Entity;

namespace Init
{
    class MyContextInitializer : DropCreateDatabaseAlways<MobileContext>
    {
```

```
        protected override void Seed(MobileContext db)
        {
            Phone p1 = new Phone { Name = "Samsung Galaxy S5", Price =
14000 };
            Phone p2 = new Phone { Name = "Nokia Lumia 630", Price = 8000
};

            db.Phones.Add(p1);
            db.Phones.Add(p2);
            db.SaveChanges();
        }
    }

}
```

4) Programm.cs

```
using System;

namespace Init
{
    class Program
    {
        static void Main(string[] args)
        {
            using (MobileContext db = new MobileContext())
            {
                var phones = db.Phones;
                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
            }
            Console.ReadLine();
        }
    }
}
```

Собственно инициализатор наследуется от одного из выше рассмотренных классов, который типизируется классом контекста: **DropCreateDatabaseAlways<MobileContext>.**

Все действия по инициализации происходят в методе **Seed**, а сама инициализация предполагает простое сохранение данных в **БД** с помощью контекста данных.

Чтобы инициализатор сработал, надо его вызвать. Один из способов вызова инициализатора предполагет вызов его в статическом конструкторе класса контекста:

```
static MobileContext()
        {
            Database.SetInitializer<MobileContext>(new
MyContextInitializer());
        }
```

Параллелизм в Entity Framework

При работе с **Entity Framework**, когда у нас одновременно множесво пользователей имеют доступ к одинаковому набору данных и могут эти данные изменять, мы можем столкнуться с проблемой параллелизма. Например, два пользователя независимо друг от друга начнут редактировать один и тот же объект. И после сохранения объекта первым пользователем второй пользователь уже будет работать с неактуальными данными.

Рассмотрим на примере. Например, в приложении **ASP.NET MVC** есть стандартное действие для редактирования:

```
public ActionResult Edit(int id)
        {
            Person p = db.Persons.Find(id);
            return View(p);
        }

[HttpPost]
public ActionResult Edit(Person p)
        {
            db.Entry(p).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Хотя данный код прекрасно будет работать, но если оба пользователя одновременно откроют один и тот же объект на редактирование, то после сохранения изменнного объекта первым пользователем, второй пользователь уже буде работать с устаревшими данными.

Что предлагает **Entity Framework** для решения этой проблемы?

Прежде всего стоит сказать, что есть два типа параллелизма: оптимистичный и пессимистичный.

При пессимистичном параллелизме (**pessimistic concurrency**) на базу данных накладываются ограничения по доступу. Например, строки объявляются только для чтения или обновления. Пессимистичный параллелизм предполагает работу на уровне базы данных и предполагает создание сложной программной логики, которая бы отслеживала и управляла правами доступа. В **Entity Framework** поддержки для пессимистичного параллелизма нет.

При оптимистичном параллелизме (**optimistic concurrency**) допускается проблема параллельного доступа к данным со стороны разных пользователей, как в выше приведенном случае с обновлением. И для решения оптимистичного параллелизма в **Entity Framework** есть свои методы.

Одни из методов предполагает, что в модели мы объявляем специальное свойство с атрибутом **[Timestamp]**, которое будет отслеживать модификацию строки в таблице. Например:

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

[Timestamp]
public byte[] RowVersion { get; set; }
    }
```

Атрибут **Timestamp** указывает, что значение свойства **RowVersion** будет включаться в создаваемое **Entity Framework**ом **SQL-выражение Where** при отправке в базу данных команд на обновление и удаление. В качестве типа для свойства используется массив байтов.

Также на представлении для редактирования надо добавить скрытое поле для хранения значения новго свойства:

```
@Html.HiddenFor(model => model.RowVersion)
```

Теперь, если два пользователя одновременно начнут редактировать одну и ту же модель, то после сохранения модели первым пользователем, второй пользователь получит исключение **DbUpdateConcurrencyException** (находится в

пространстве имен **System.Data.Entity.Infrastructure**), которое соответственно надо обработать:

```
[HttpPost]
public ActionResult Edit(Person p)
        {
            try
            {
                db.Entry(p).State = EntityState.Modified;
                db.SaveChanges();
            }
            catch (DbUpdateConcurrencyException ex)
            {
                ViewBag.Error = "Объект ранее уже был изменен";
                return View(p);
            }
            return RedirectToAction("Index");
        }
```

Ну и чтобы дать пользователю знать об ошибке, где-нибудь в представлении выведем **ViewBag.Error**. И теперь при одовременном редактировании одного и того же объекта второй сохраняющий пользователь получит ошибку, а его обновления не будут сохранены в базу данных.

При чем это касается не только обновления, но и удаления.

Управление транзакциями

Когда мы вызываем при добавлении, обновлении, удалении метод **SaveChanges()**, то фактически **Entity Framework** проводит транзакцию. В приложении на **C#** мы также управлять транзакциями. Когда може быть полезно ручное управление транзакциями? Транзакции применяются чаще всего для того, чтобы выполнить последовательность операций, которые должны отличаться высокой согласованностью, и при этом иметь возможность откатить все сделанные из этих операций назад, если какая-нибудь из этих операций завершилась с ошибкой

Рассмотрим на примере. Например, у нас есть в базе данных человек по имени **Bob**. У него родился сын, которого тоже назвали **Bob**. И теперь, чтобы их разграничить, отцу мы присваиваем имя **Bob Senior**, а сыну - **Bob Junior**:

```
static void Main(string[] args)
        {
            using (UserContext db = new UserContext())
            {
                using (var transaction = db.Database.BeginTransaction())
                {
                    try
                    {
                        Person p1 = db.Persons.FirstOrDefault(p => p.Name
== "Bob");
                        p1.Name = "Bob Senior";
                        db.Entry(p1).State = EntityState.Modified;
                        Person p2 = new Person { Name = "Bob Junior", Age
= 1 };
                        db.Persons.Add(p2);
                        db.SaveChanges();
                        transaction.Commit();
                    }
                    catch (Exception ex)
                    {
                        transaction.Rollback();
                    }
                }

                foreach (Person p in db.Persons.ToList())
                    Console.WriteLine("Name: {0}  Age: {1}", p.Name,
p.Age);
            }
        }
```

Для создания транзакции используется выражение `var transaction = db.Database.BeginTransaction()`, и так как класс **DbContextTransactionреализует** интерфейс **IDisposable**, то весь код транзакции обертываем в конструкцию **using**.

Далее производятся все те же обычные операции редактирования и добавления. После операций вызывается метод **transaction.Commit()** для коммита транзакции.

Однако если, допустим, у нас возникнет исключение параллелизма или любое другое исключение при редактировании или добавлении, то есть одна из операций (или обе) завершатся неудачно, то они в целом смысла уже не будут иметь. Поэтому надо будет откатить все сделанные изменения с помощью метода **transaction.Rollback()**.

4.LINQ to Entities

Введение в LINQ to Entities

Ранее мы использовали ряд операций для получения данных из **БД**. В основе подобных операций лежит технология **LINQ (Language Integrated Query)**, или точнее **LINQ to Entities**. **LINQ to Entities** предлагает простой и интуитивно понятный подход для получения данных с помощью выражений, которые по форме близки выражениям языка **SQL**.

Хотя при работе с базой данных мы оперируем запросами **LINQ**, но база данных понимает только запросы на языке **SQL**. Поэтому между **LINQ to Entities** и базой данных есть проводник, который позволяет им взаимодействовать. Этим проводником является провайдер **EntityClient**. Он создает интерфейс для взаимодействия с провайдером **ADO.NET** для **SQL Server** а.

Для начала взаимодействия с базой данных создается объект **EntityConnection**. Через объект **EntityCommand** он отправляет запросы, а с помощью объекта **EntityDataReader** считывает извлеченные из **БД** данные. Однако разработчику не надо напрямую взаимодействовать с этими объектами, фреймворк все сделает за него. Задача же разработчика сводится в основном к написанию запросов к базе данных с помощью **LINQ**.

Прежде чем приступить к обзору основных запросов в **LINQ to Entities**, для работы с материалом этой главы создадим новые модели по связи один-ко-многим:

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

        public ICollection<Phone> Phones { get; set; }
        public Company()
        {
            Phones = new List<Phone>();
        }
    }
€
class Phone
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Price { get; set; }
```

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

У нас здесь модель телефона и модель компании-производителя. Теперь создадим контекст данных и инициализатор базы данных начальными данными:

```
class PhoneContext : DbContext
    {
        static PhoneContext()
        {
            Database.SetInitializer(new MyContextInitializer());
        }
        public PhoneContext()
            : base("DefaultConnection")
        { }

        public DbSet<Company> Companies { get; set; }
        public DbSet<Phone> Phones { get; set; }
    }

 class MyContextInitializer : DropCreateDatabaseAlways<PhoneContext>
    {
        protected override void Seed(PhoneContext db)
        {
            Company c1 = new Company { Name = "Samsung" };
            Company c2 = new Company { Name = "Apple" };
            db.Companies.Add(c1);
            db.Companies.Add(c2);

            db.SaveChanges();

            Phone p1 = new Phone { Name = "Samsung Galaxy S5", Price =
20000, Company = c1 };
            Phone p2 = new Phone { Name = "Samsung Galaxy S4", Price =
15000, Company = c1 };
            Phone p3 = new Phone { Name = "iPhone5", Price = 28000,
Company = c2 };
            Phone p4 = new Phone { Name = "iPhone 4S", Price = 23000,
Company = c2 };

            db.Phones.AddRange(new List<Phone>() { p1, p2, p3, p4 });
            db.SaveChanges();
        }
    }
```

Чтобы база данных уже содержала некоторые данные, в инициализаторе **БД** создается несколько объектов. Чтобы задействовать инициализатор, он вызывается в статическом конструкторе контекста данных:

```
static PhoneContext()
      {
            Database.SetInitializer(new MyContextInitializer());
      }
```

Для создания запросов в **Linq to Entities**, так же, как и в **Linq to Objects**, мы можем применять операторы **LINQ** и методы расширения **LINQ**.
Например, используем некоторые операторы **LINQ**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = from p in db.Phones
                             where p.CompanyId == 1
                             select p;
            }
```

И тот же запрос с помощью методов расширений **LINQ**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Where(p => p.CompanyId == 1);
            }
```

Код программы:

```
static void Main(string[] args)
        {
            using (PhoneContext db = new PhoneContext())
            {
                var phones = from p in db.Phones
                             where p.CompanyId == 1
                             select p;

                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);

                Console.ReadLine();
            }


            using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Where(p => p.CompanyId == 1);
```

```
                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
                Console.ReadLine();
            }
        }
```

Оба запроса в итоге транслируются в одной выражение **sql**:

```
SELECT [Extent1].[Id] AS [Id],
        [Extent1].[Name] AS [Name],
        [Extent1].[Price] AS [Price],
        [Extent1].[CompanyId] AS [CompanyId]
FROM [dbo].[Phones] AS [Extent1]
WHERE 1 = [Extent1].[CompanyId]
```

Важно понимать различие между **Linq to Entities** и **Linq to Objects**:

```
var phones = db.Phones.Where(p => p.CompanyId == 1).ToList().Where(p =>
p.Id < 10);
```

Здесь используются два метода **Where**, но их реализация будет различной. В первом случае,

db.Phones.Where(p=> p.CompanyId == 1)

транслируется в выражение **SQL**, которое было рассмотрено выше. Далее метод **ToList()** по результатам запроса создает список в памяти компьютера. Посде этого мы уже имеем дело со списком в памяти, а не с базой данных. И далее вызов **Where(p=> p.Id<10)** будет обращаться к списку в памяти и будет представлять **Linq to Object**.

А теперь рассмотрим некоторые приемы применения **LINQ** к запросам из базы данных.

Выборка и проекция из базы данных

Для выборки применяется метод **Where**. Выберем и **БД** все модели, производитель которых - "**Samsung**":

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Where(p => p.Company.Name ==
"Samsung");
                foreach (Phone p in phones)
```

```
                Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
            }
```

Для выборки одного объекта мы можем использовать метод **Find()**. Данный метод не является методом **Linq**, он определен у класса **DbSet**:

```
// выберем элемент с id=3
Phone myphone = db.Phones.Find(3);
Console.WriteLine("{0}.{1} - {2}", myphone.Id, myphone.Name,
myphone.Price);
```

Но в качестве альтернативы мы можем использовать методы **Linq First()/FirstOrDefault()**. Они получают первый элемент выборки, который соответствует определенному условию. Использование метода **FirstOrDefault()** является более гибким, так как если выборка пуста, то он вернет значение **null**. А метод **First()** в той же ситуации выбросит ошибку.

```
Phone myphone = db.Phones.FirstOrDefault(p => p.Id == 3);
if (myphone != null)
   Console.WriteLine(myphone.Name);
```

Теперь сделаем проекцию. Допустим, нам надо добавить в результат выборки название компании. Мы можем использовать метод **Include** для подсоединения к объекту связанных данных из другой таблицы:

```
var phones = db.Phones.Include(p => p.Company);
```

Но не всегда нужны все свойства выбираемых объектов. В этом случае мы можем применить метод **Select** для проекции извлеченных данных на новый тип:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Select(p => new
                {
                    Name = p.Name,
                    Price = p.Price,
                    Company = p.Company.Name
                });

                foreach (var p in phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
            }
```

В итоге метод **Select** из полученных данных спроецирует новый тип. В даном случае мы получим данные анонимного типа, но это также может быть определенный пользователем тип. Например:

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

И спроецируем выборку на этот тип:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Select(p => new Model
                {
                    Name = p.Name,
                    Price = p.Price,
                    Company = p.Company.Name
                });
                foreach (Model p in phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
            }
```

Сортировка

Для упорядочивания полученных из **БД** данных по возрастанию служит метод **OrderBy** или оператор **orderby**. Например, отсортируем объекты по возрастанию по свойству **Name**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.OrderBy(p => p.Name);
                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
            }
```

В результате **Entity Framework** будет генерировать следующее выражение **SQL**, которое будет упорядочивать данные:

```
SELECT [Extent1].[Id] AS [Id],
           [Extent1].[Name] AS [Name],
           [Extent1].[Price] AS [Price],
```

```
            [Extent1].[CompanyId] AS [CompanyId]
FROM [dbo].[Phones] AS [Extent1]
ORDER BY [Extent1].[Name] ASC
```

В качестве альтернативы методу **OrderBy** можно использовать оператор **orderby**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = from p in db.Phones
                             orderby p.Name
                             select p;
                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
            }
```

Для сортировки по убыванию применяется метод **OrderByDescending()**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.OrderByDescending(p => p.Name);
                foreach (Phone p in phones)
                    Console.WriteLine("{0}.{1} - {2}", p.Id, p.Name,
p.Price);
            }
```

Если нам надо отсортировать данные сразу по нескольким критериям, то мы можем применить методы **ThenBy()** (для сортировки по возрастанию) и **ThenByDescending()**. Например, отсортируем результат проекции по двум столбцам:

```
var phones = db.Phones.Select(p => new { Name = p.Name, Company =
p.Company.Name, Price = p.Price }).OrderBy(p => p.Price).ThenBy(p =>
p.Company);
```

Соединение таблиц

Для объединения таблиц по определенному критерию используется метод **Join**. Например, в нашем случае таблица телефонов и таблица компаний имеет общий критерий - **Id** компании, по которому можно провести объединение таблиц:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Join(db.Companies, // второй набор
```

```
                    p => p.CompanyId, // свойство-селектор объекта из
первого набора
                    c => c.Id, // свойство-селектор объекта из второго
набора
                    (p, c) => new // результат
                    {
                        Name = p.Name,
                        Company = c.Name,
                        Price = p.Price
                    });
                foreach (var p in phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
            }
```

Метод **Join** принимает четыре параметра:

- вторую таблицу, которая соединяется с текущей
- свойство объекта - столбец из первой таблицы, по которому идет соединение
- свойство объекта - столбец из второй таблицы, по которому идет соединение
- новый объект, который получается в результате соединения

В итоге данный запрос будет транслироваться в следующее выражение **SQL**:

```
SELECT [Extent1].[Price] AS [Price],
        [Extent1].[Name] AS [Name],
        [Extent2].[Name] AS [Name1]
FROM  [dbo].[Phones] AS [Extent1]
INNER JOIN [dbo].[Companies] AS [Extent2]
ON [Extent1].[CompanyId] = [Extent2].[Id]
```

Аналогичного результата мы могли бы достигнуть, если бы использовали оператор **join**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = from p in db.Phones
                             join c in db.Companies on p.CompanyId equals
c.Id
                             select new { Name = p.Name, Company =
c.Name, Price = p.Price };

                foreach (var p in phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
            }
```

Группировка

Чтобы сгруппировать данные по определенным параметрам используются оператор **group by** или метод **GroupBy()**. Например, сгруппируем модели телефонов по производителю:

```
using (PhoneContext db = new PhoneContext())
            {
                var groups = from p in db.Phones
                             group p by p.Company.Name;
                foreach (var g in groups)
                {
                    Console.WriteLine(g.Key);
                    foreach (var p in g)
                        Console.WriteLine("{0} - {1}", p.Name, p.Price);
                    Console.WriteLine();
                }
            }
```

В данном случае критерием для объединения групп является название компании, то есть столбец **Name** из связанной таблицы **Companies**. Критерий, по которому проводится группировка, является ключом. Ключ мы можем получить через свойство **Key**, которое есть у группы.

В итоге мы получим несколько групп, каждая из которых будет содержать несколько элементов. Например, в моем случае вывод будет следующим:

Samsung
Samsung Galaxy S5 - 20000
Samsung Galaxy S4 - 15000

Apple
iPhone5 - 28000
iPhone4S - 23000

Аналогично работает метод **GroupBy():**

```
var groups = db.Phones.GroupBy(p => p.Company.Name);
```

Кроме свойства **Key** у группы есть свойство **Count**, которое хранит количество элементов в данной группе. Например, сформируем новый элемент, который будет содержать ключ группы и количество ее элементов:

```
using (PhoneContext db = new PhoneContext())
```

```
            {
                var groups = from p in db.Phones
                             group p by p.Company.Name into g
                             select new { Name = g.Key, Count = g.Count()
};
                // альтернативный способ
                // var groups = db.Phones.GroupBy(p=>p.Company.Name)
                //                       .Select(g => new { Name =
g.Key, Count = g.Count()});
                foreach (var c in groups)
                    Console.WriteLine("Производитель: {0} Кол-во моделей:
{1}", c.Name, c.Count);
            }
```

В моем случае я получу следующий вывод:

Производитель: Apple Кол-во моделей: 2

Производитель: Samsung Кол-во моделей: 2

Операции с множествами: объединение, пересечение, разность

Ряд методов **Linq** позволяют работать с результатами выборки как со множествами, производя операции на объединение, пересечение, разность двух выборок.

Но перед использованием данных методов надо учитывать, что они проводятся над однородными выборками с одинаковым определением строк, то есть которые совпадают по составу столбцов.

Объединение

Для объединения двух выборок используется метод **Union()**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Where(p => p.Price < 25000)
                    .Union(db.Phones.Where(p =>
p.Name.Contains("Samsung")));
                foreach (var item in phones)
                    Console.WriteLine(item.Name);
            }
```

Метод **Union** в качестве параметра принимает результаты второй выборки и объединяет ее с исходной.

При этом мы не можем объединить две разнородные выборки, например, таблицу, моделей телефонов и таблицу производителей телефонов. Однако уместна следующая запись:

```
var result = db.Phones.Select(p => new { Name = p.Name })
    .Union(db.Companies.Select(c => new { Name = c.Name }));
```

Первая выборка после метода **Select** будет формировать набор элементов с одним столбцом **Name**. Вторая выборка из таблицы компаний после метода **Select** также будет формировать набор элементов с одним столбцом **Name**. Поэтому строки в обоих выборках будут однородны, и мы их сможем объединять.

Пересечение

Чтобы найти пересечение выборок, то есть те элементы, которые присутствуют сразу в двух выборках, используется метод **Intersect()**:

```
using (PhoneContext db = new PhoneContext())
            {
                var phones = db.Phones.Where(p => p.Price < 25000)
                    .Intersect(db.Phones.Where(p =>
p.Name.Contains("Samsung")));
                foreach (var item in phones)
                    Console.WriteLine(item.Name);
            }
```

Разность

Если нам надо найти элементы первой выборки, которые отсутствуют во второй выборке, то мы можем использовать метод **Except**:

```
using (PhoneContext db = new PhoneContext())
            {
                var selector1 = db.Phones.Where(p => p.Price & 25000); //
Samsung Galaxy S4, Samsung Galaxy S4, iPhone S4
                var selector2 = db.Phones.Where(p =>
p.Name.Contains("Samsung")); // Samsung Galaxy S4, Samsung Galaxy S4
                var phones = selector1.Except(selector2); // результат -
iPhone S4

                foreach (var item in phones)
                    Console.WriteLine(item.Name);
            }
```

Агрегатные операции

Linq to Entities поддерживает обращение к встроенным функциями **SQL** через специальные методы **Count, Sum** и т.д.

Количество элементов в выборке

Метод **Count()** позволяет найти количество элементов в выборке:

```
using (PhoneContext db = new PhoneContext())
            {
                int number1 = db.Phones.Count();
                // найдем кол-во моделей, которые в названии содержат
Samsung
                int number2 = db.Phones.Count(p =>
p.Name.Contains("Samsung"));

                Console.WriteLine(number1);
                Console.WriteLine(number2);
            }
```

Минимальное, максимальное и среднее значения

Для нахождения минимального, максимального и среднего значений по выборке применяются функции **Min()**, **Max()** и **Average()** соответственно. Найдем минимальную, максимальную и среднюю цену по моделям:

```
using (PhoneContext db = new PhoneContext())
            {
                // минимальная цена
                int minPrice = db.Phones.Min(p => p.Price);
                // максимальная цена
                int maxPrice = db.Phones.Max(p => p.Price);
                // средняя цена на телефоны фирмы Samsung
                double avgPrice = db.Phones.Where(p => p.Company.Name ==
"Samsung")
                                           .Average(p => p.Price);

                Console.WriteLine(minPrice);
                Console.WriteLine(maxPrice);
                Console.WriteLine(avgPrice);
            }
```

Сумма значений

Для получения суммы значений используется метод **Sum()**:

```
using (PhoneContext db = new PhoneContext())
```

```
            {
                // суммарная цена всех моделей
                int sum1 = db.Phones.Sum(p => p.Price);
                // суммарная цена всех моделей фирмы Samsung
                int sum2 = db.Phones.Where(p =>
p.Name.Contains("Samsung"))
                                    .Sum(p => p.Price);
                Console.WriteLine(sum1);
                Console.WriteLine(sum2);
            }
```

IEnumerable и IQueryable в Entity Framework

Методы расширений **LINQ** могут возвращать два объекта: **IEnumerable** и **IQueryable**. С одной стороны, интерфейс **IQueryable** наследуется от **IEnumerable**, поэтому по идее объект **IQueryable** это и есть также объект **IEnumerable**. Но реальность несколько сложнее. Между объектами этих интерфейсов есть разница в плане функциональности, поэтому они не взаимозаменяемы.

Интерфейс **IEnumerable** находится в пространстве имен **System.Collections**. Объект **IEnumerable** представляет набор данных в памяти и может перемещаться по этим данным только вперед. Запрос, представленный объектом **IEnumerable**, выполняется немедленно и полностью, поэтому получение данных приложением происходит быстро.

При выполнении запроса **IEnumerable** загружает все данные, и если нам надо выполнить их фильтрацию, то сама фильтрация происходит на стороне клиента.

Интерфейс **IQueryable** располагается в пространстве имен **System.Linq**. Объект **IQueryable** предоставляет удаленный доступ к базе данных и позволяет перемещаться по данным как в прямом порядке от начала до конца, так и в обратном порядке. В процессе создания запроса, возвращаемым объектом которого является **IQueryable**, происходит оптимизация запроса. В итоге в процессе его выполнения тратится меньше памяти, меньше пропускной способности сети, но в то же время он может обрабатываться чуть медленнее, чем запрос, возвращающий объект **IEnumerable**.

Для примера возьмем два вроде бы идентичных выражения. Объект **IEnumerable**:

```
using (PhoneContext db = new PhoneContext())
            {
                IEnumerable phoneIEnum = db.Phones;
                phoneIEnum = phoneIEnum.Where(p => p.Id > id);
            }
```

Здесь запрос будет иметь следующий вид:

```
SELECT
      [Extent1].[Id] AS [Id],
      [Extent1].[Name] AS [Name],
      [Extent1].[Company] AS [Company]
      FROM [dbo].[Phones] AS [Extent1]
```

Фильтрация результата, обозначенная с помощью метода **Where(p => p.Id > id)** будет идти уже после выборки из **БД** в самом приложении.
Чтобы совместить фильтры, нам надо было сразу применить метод **Where: db.Phones.Where(p => p.Id > id);**

Объект **IQueryable**:

```
using (PhoneContext db = new PhoneContext())
            {
                IQueryable phoneIQuer = db.Phones;
                phoneIQuer = phoneIQuer.Where(p => p.Id > id);
            }
```

Здесь запрос будет иметь следующий вид:

```
SELECT
      [Extent1].[Id] AS [Id],
      [Extent1].[Name] AS [Name],
      [Extent1].[Company] AS [Company]
      FROM [dbo].[Phones] AS [Extent1]
      WHERE [Extent1].[Id] >3
```

Таким образом, все методы суммируются, запрос оптимизируется, и только потом происходит выборка из базы данных. Что же лучше использовать? Все зависит от конкретной ситуации. Если разработчику нужен весь набор возвращаемых данных, то лучше использовать **IEnumerable**, предоставляющий максимальную скорость. Если же нам не нужен весь набор, а то только некоторые отфильтрованные данные, то лучше применять **IQueryable**.

Метод AsNoTracking

Когда контекст данных извлекает данные из базы данных, **Entity Framework** помещает извлеченные объекты в кэш и отслеживает изменения, которые происходят с этими объектами вплоть до использования метода **SaveChanges()**, который фиксирует все изменения в базе данных. Но нам не всегда необходимо отслеживать изменения. Например, нам надо просто вывести данные для просмотра.

Чтобы данные не помещались в кэш, применяется метод **AsNoTracking()**. При его применении возвращаемые из запроса данные не кэшируются. А это означает, что **Entity Framework** не производит какую-то дополнительную обработку и не выделяет дополнительное место для хранения извлеченных из **БД** объектов.

Метод **AsNoTracking()** применяется к набору **IQueryable**:

```
using (BookContext db = new BookContext())
            {
                IEnumerable<Book> books1 =
db.Books.AsNoTracking().ToList();
                IEnumerable<Book> books2 = db.Books
                                .Where(b => b.Price > 200)
                                .AsNoTracking().ToList();
                IEnumerable<Book> books3 = db.Books
                                .Include(b => b.Author)
                                .AsNoTracking().ToList();
            }
```

Небольшой пример. У нас в базе данных есть есть несколько моделей **Phones**:

```
using (PhoneContext db = new PhoneContext())
            {
                db.Phones.Add(new Phone { Name = "Samsung Galaxy Note"
});
                db.Phones.Add(new Phone { Name = "iPhone 6" });
                db.SaveChanges();
            }
```

При обычном выполнении:

```
using (PhoneContext db = new PhoneContext())
            {
                Phone firstPhone = db.Phones.FirstOrDefault();
                firstPhone.Name = "Samsung Galaxy Ace 2";
```

```
            db.SaveChanges();

            List<Phone> phones = db.Phones.ToList();
        }
```

Мы увидим, что в наборе **phones** первый элемент имеет название "**Samsung Galaxy Ace 2**".

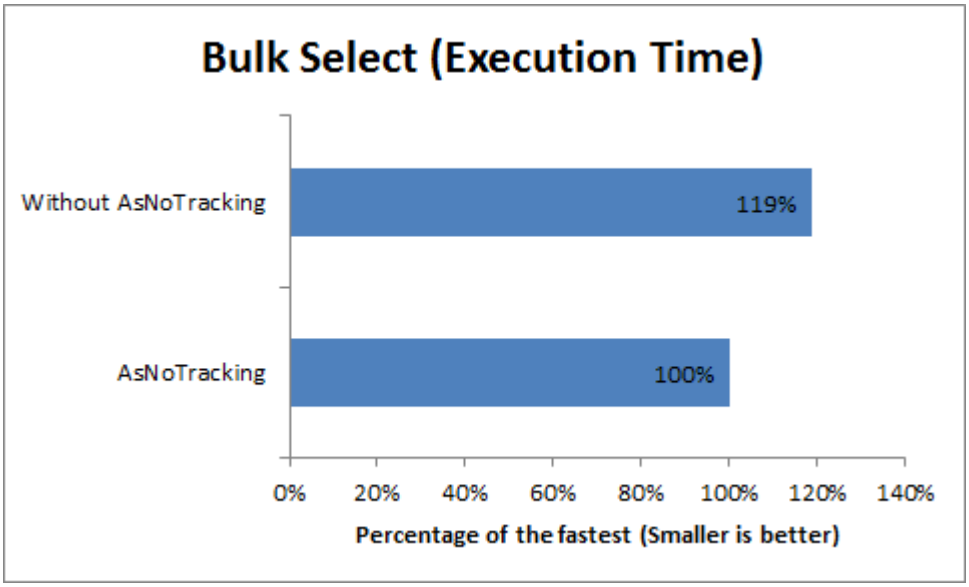
Но если бы мы использовали **AsNoTracking**, то результат был бы другой:

```
using (PhoneContext db = new PhoneContext())
        {
            Phone firstPhone =
db.Phones.AsNoTracking().FirstOrDefault();
            firstPhone.Name = "Samsung Galaxy Ace 2";
            db.SaveChanges();

            List<Phone> phones = db.Phones.AsNoTracking().ToList();
        }
```

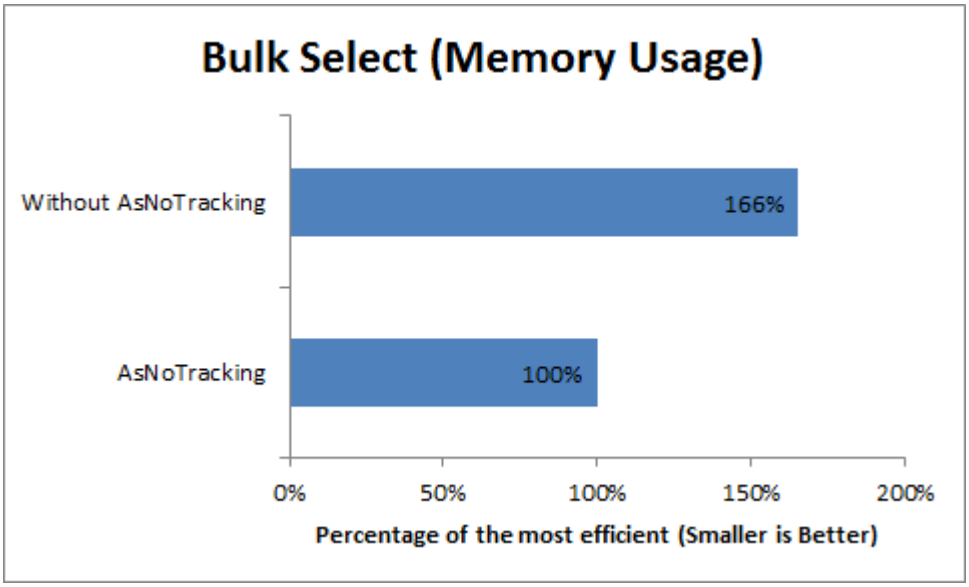
Так как при получении первого элемента используется **AsNoTracking**, он не будет отслеживаться, и поэтому вызов **db.SaveChanges()** никак не повлияет на базу данных, а первый элемент сохранит свое первоначальное значение - "**Samsung Galaxy Note**".

Если мы обратимся к тестам сравнения простого запроса и запроса с использованием **AsNoTracking**, то мы увидим преимущество как по скорости выполнения, так и по использованной памяти.

Сравнение по скорости выполнения



Сравнение по использованию памяти



Правда, различия будут заметны на больших объемах данных в сотни и тысячи объектов.

Когда следует использовать **AsNoTracking**? Если нам надо просто вывести данные для отображения без необходимости их дальнейшего обновления, тогда как раз тот случай, когда мы можем использовать **AsNoTracking**.

5.SQL в Entity Framework

Работа с SQL

В большинстве случае разработчики смогут создать эффективные запросы с помощью методов и операторов **LINQ**. Однако в **Entity Framework** доступно также прямое выполнение **sql-запросов**.

Для осуществления прямых **sql-**запросов к базе данных можно воспользоваться свойством **Database**, которое имеется у класса контекста данных. Данное свойство позволяет получать информацию о базе данных, подключении и осуществлять запросы к **БД**. Например, получим строку подключения:

```
using (PhoneContext db = new PhoneContext())
        {
          Console.WriteLine(db.Database.Connection.ConnectionString);
        }
```

Непосредственно для создания запроса нам надо использовать метод **SqlQuery**, который принимает в качестве параметра **sql-выражение**.

Для примера возьмем базу данных, созданную в прошлой теме, которая описывается следующими моделями и контекстом данных:

```
class PhoneContext : DbContext
        {
            public PhoneContext()
                : base("DefaultConnection")
            { }

            public DbSet<Company> Companies { get; set; }
            public DbSet<Phone> Phones { get; set; }
        }

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

            public ICollection<Phone> Phones { get; set; }
            public Company()
            {
                Phones = new List<Phone>();
            }
        }

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

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

Итак, получим все модели из таблицы **Companies**:

```
static void Main(string[] args)
        {
            using (PhoneContext db = new PhoneContext())
            {
                var comps = db.Database.SqlQuery<Company>("SELECT * FROM
Companies");
                foreach (var company in comps)
                    Console.WriteLine(company.Name);
            }
```

```
            Console.ReadLine();
        }
```

Выражение **SELECT** извлекает данные из таблицы. Так как эта таблица сопоставляется с моделью **Company** и хранит объекты этой модели, то данный вызов типизируется классом **Company:** `db.Database.SqlQuery<Company>()`.

Другая версия метода **SqlQuery()** позволяет использовать параметры. Например, выберем из **БД** все модели, которые в названии имеют подстроку "**Samsung**":

```
using (PhoneContext db = new PhoneContext())
            {
                System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@name", "%Samsung%");
                var phones = db.Database.SqlQuery<Phone>("SELECT * FROM
Phones WHERE Name LIKE @name", param);
                foreach (var phone in phones)
                    Console.WriteLine(phone.Name);
                 }
```

Класс **SqlParameter** из пространства имен **System.Data.SqlClient** позволяет задать параметр, который затем передается в запрос **sql**.

Метод **SqlQuery()** осуществляет выборку из **БД**, но кроме выборки нам, возможно, придется удалять, обновлять уже существующие или вставлять новые записи. Для этой цели применяется метод **ExecuteSqlCommand()**, который возвращает количество затронутых командой строк:

```
using (PhoneContext db = new PhoneContext())
            {
                // вставка
                int numberOfRowInserted =
db.Database.ExecuteSqlCommand("INSERT INTO Companies (Name) VALUES
('HTC')");
                // обновление
                int numberOfRowUpdated =
db.Database.ExecuteSqlCommand("UPDATE Companies SET Name='Nokia' WHERE
Id=3");
                // удаление
                int numberOfRowDeleted =
db.Database.ExecuteSqlCommand("DELETE FROM Companies WHERE Id=3");
            }
```

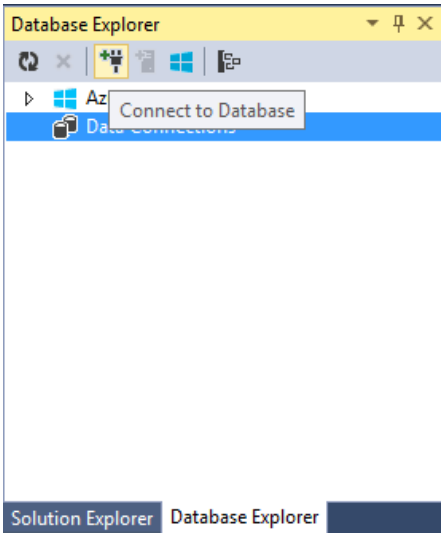
Хранимые функции

Особое внимание при работе с **sql**-запросами представляют хранимые функции и процедуры. Рассмотрим вызов хранимой функции в приложении на **C#**.

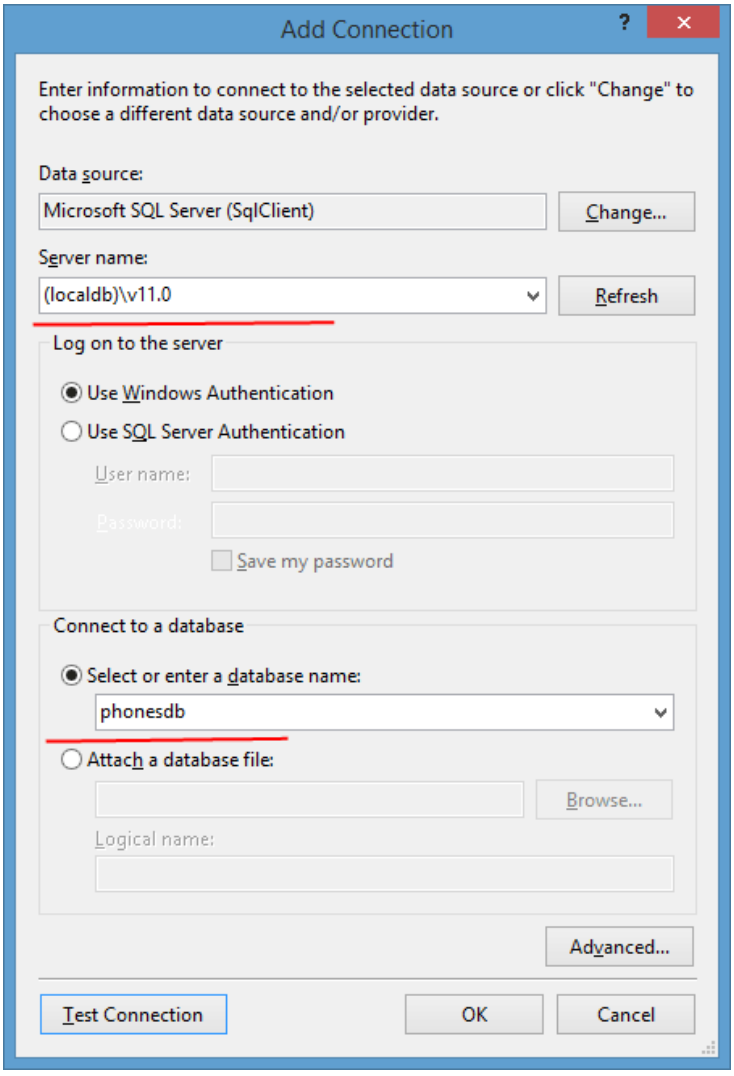
Вначале создадим функцию. Пусть наша база данных описывается следующим контекстом данных и моделями:

```
class PhoneContext : DbContext
    {
        public PhoneContext()
            : base("DBConnection")
        { }

        public DbSet<Company> Companies { get; set; }
        public DbSet<Phone> Phones { get; set; }
       }

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

        public ICollection<Phone> Phones { get; set; }
        public Company()
        {
            Phones = new List<Phone>();
        }
    }

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

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

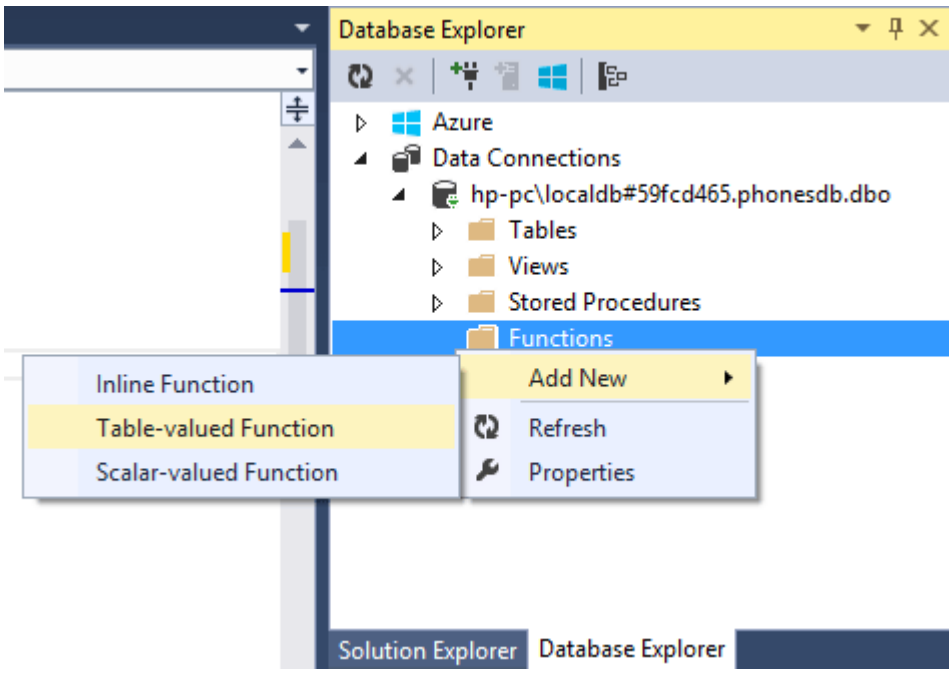
Теперь в **Visual Studio** в окне **Database Explorer** откроем базу данных. Для этого нажмем в окне **Database Explorer** на кнопку **Connect to Database**:



Выберем нужную базу данных. В моем случае это база данных **phonesdb**:



После открытия базы данных в окне **Database Explorer** найдем узел **Functions** и нажмем на него правой кнопкой мыши. В появившемся контекстном меню выберем **Add New -> Table-valued Function**:



После этого **Visual Studio** генерирует и октрывает файл скрипта со следующим содержимым:

```
CREATE FUNCTION [dbo].[Function]
(
	@param1 int,
	@param2 char(5)
)
RETURNS @returntable TABLE
(
	c1 int,
	c2 char(5)
)
AS
BEGIN
	INSERT @returntable
	SELECT @param1, @param2
	RETURN
END
```

Изменим скрипт следующим образом:

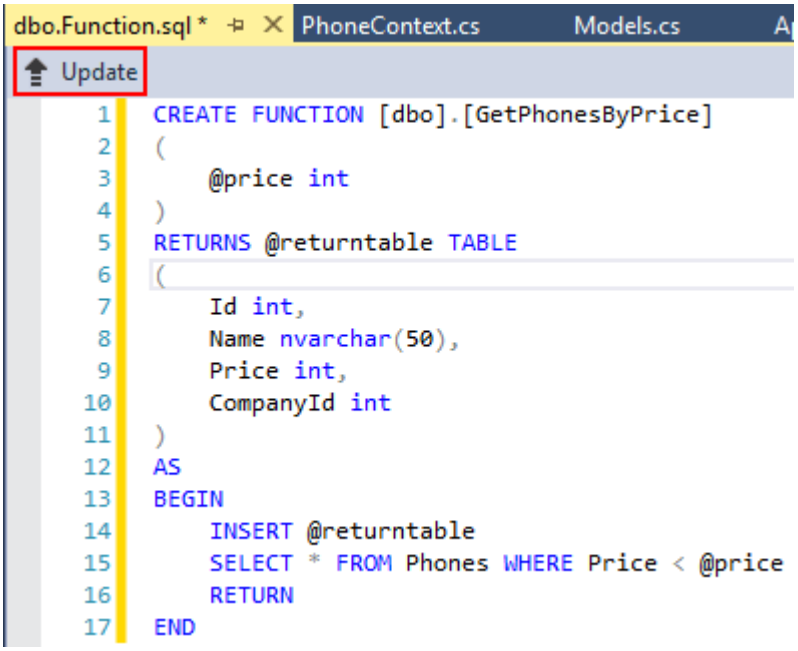
```
CREATE FUNCTION [dbo].[GetPhonesByPrice]
(
	@price int
)
RETURNS @returntable TABLE
(
	Id int,
	Name nvarchar(50),
	Price int,
	CompanyId int
)
AS
BEGIN
	INSERT @returntable
	SELECT * FROM Phones WHERE Price < @price
	RETURN
END
```

С помощью выражения **CREATE FUNCTION [dbo].[GetPhonesByPrice]** создается новая функция **GetPhonesByPrice**. Далее после названия определяется список параметров. Пусть наша функция принимает только один параметр **@price**, который имеет тип **int**, то есть целочисленное значение.

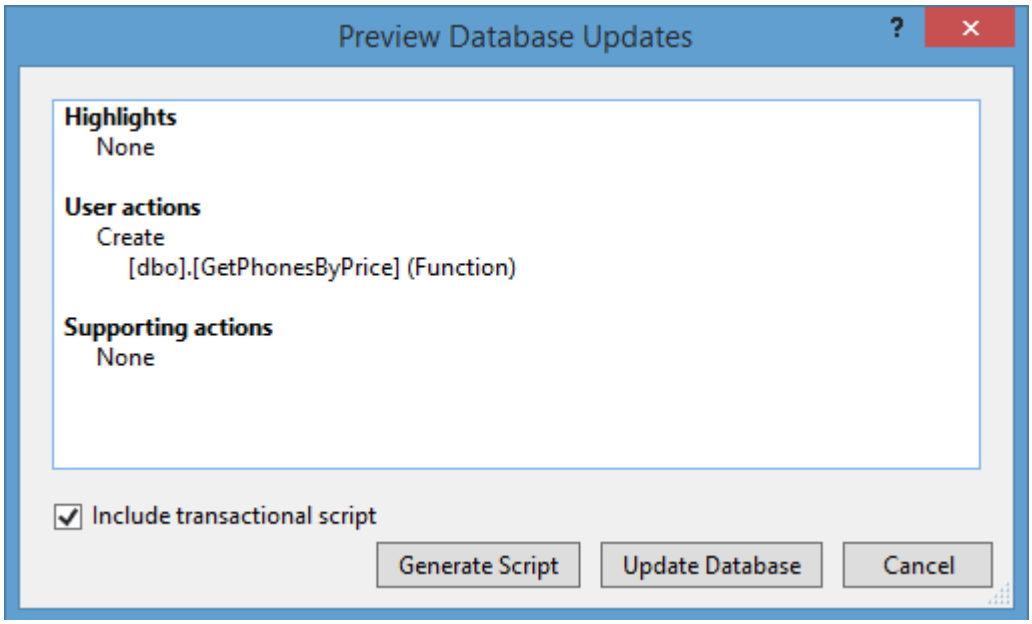
Затем идет определение возвращаемого объекта-таблицы в выражении **RETURNS @returntable TABLE(...)**. В скобках идет перечисление столбцов возвращаемой таблицы. В данном случае они совпадают с определением таблицы **Phones**. То есть таблица будет содержать объекты класса **Phone**.

Между выражениями **BEGIN** и **END** идет собственно выполнение функции. В данном случае с помощью оператора **WHERE** функция будет находить все строки, у которых столбец **Price** содержит меньшее значение, чем в параметре **@price**.

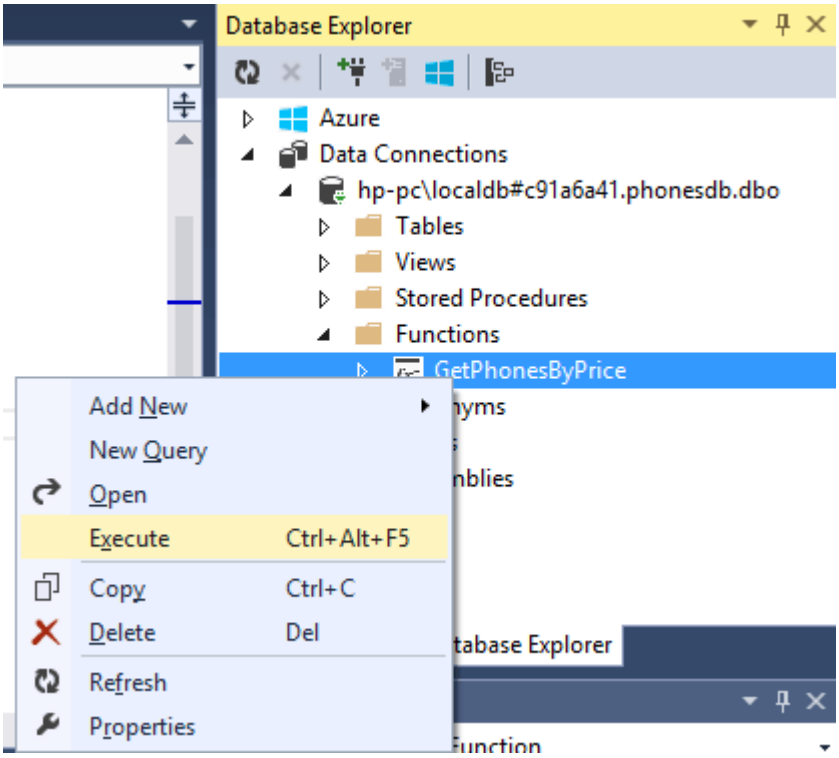
Теперь добавим функцию в базу данных. Для этого нажмем на кнопку **Update:**



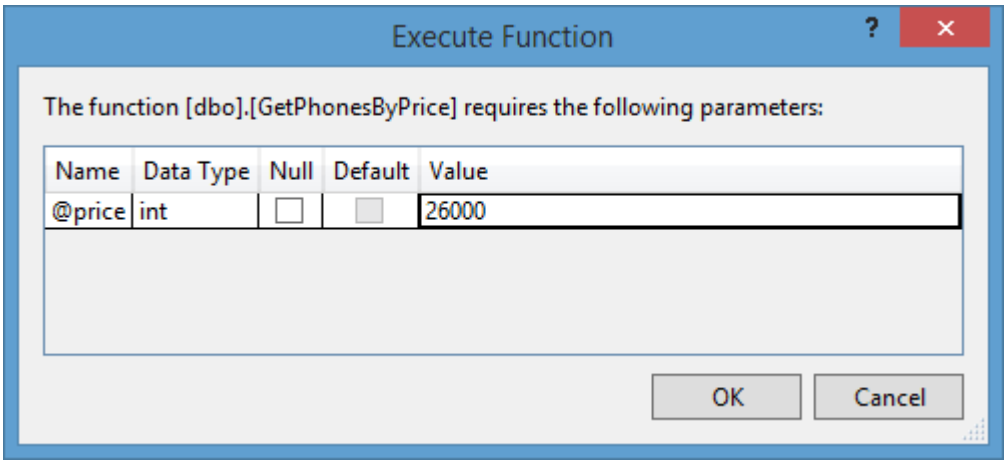
Затем в появившемся окне нажмем на кнопку **Update Database**



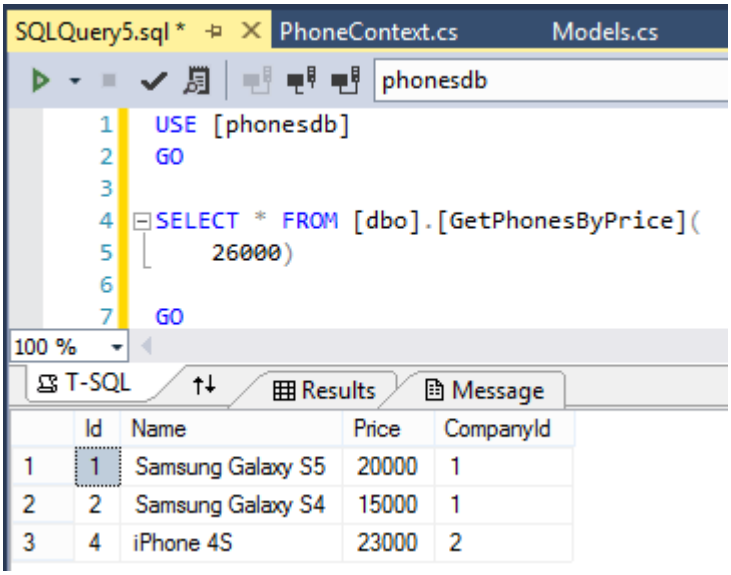
После этого в окне **Database Explorer** в узле **Functions** появится подузел добавленной функции. И мы ее можем уже использовать. Но перед обращением к ней из кода **C#** мы ее протестируем, чтобы убедиться, что она работает как надо. Для этого нажмем на функцию правой кнопкой мыши и в появившемся меню выберем пункт **Execute**:



После этого откроется окно для установки входных параметров функции. Введем в поле **Value** какое-нибудь число, которое будет передаваться в функцию в качестве параметра:



И **Visual Studio** сгенерирует и сразу же выполнит скрипт с функцией и переданным в нее параметром:



Как видно, я получил ожидаемые результаты, значит, функция работает правильно.

Теперь обратимся к ней из кода **C#**:

```
static void Main(string[] args)
        {
            using (PhoneContext db = new PhoneContext())
            {
                System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@price", 26000);
                var phones = db.Database.SqlQuery<Phone>("SELECT * FROM
GetPhonesByPrice (@price)", param);
                foreach (var phone in phones)
                    Console.WriteLine(phone.Name);
            }
            Console.ReadKey();
        }
```

В этом случае я получу те же результаты, что и при выполнении скрипта выше.

Теперь похожим образом создадим новую функцию, которая будет вычислять сумму с учетом скидки. Код функции:

```
CREATE FUNCTION [dbo].[GetPriceWithDiscount]
(
      @discount int
)
RETURNS @returntable TABLE
(
      Name nvarchar(50),
      Price decimal(8,3)
```

```
)
AS
BEGIN
      INSERT @returntable
      SELECT Name, Price - Price * @discount / 100
      FROM Phones
      RETURN
END
```

Функция принимает в качестве параметра процент скидки, например, **10%**. И на выходе она возвращает таблицу из двух полей - названия модели и цены с учетом скидки.

Так как функция фактически будет возвращать новый объект с двумя свойствами - **Name** и **Price**, при этом **Price** уже имеет тип **decimal**, то нам нужен соответствующий класс **C#**. Добавим в проект следующий класс:

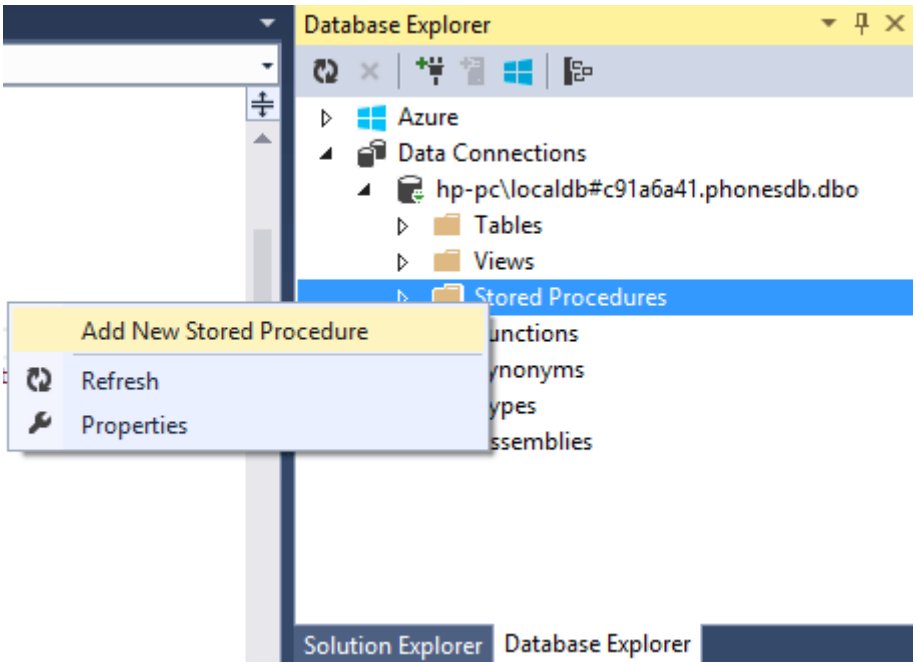
```
class DiscountPhone
    {
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
```

И теперь результат функции мы можем получить следующим образом:

```
static void Main(string[] args)
        {
            using (PhoneContext db = new PhoneContext())
            {
                // скидка - 15%
                System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@discount", 15);
                var phones = db.Database.SqlQuery<DiscountPhone>("SELECT
* FROM GetPriceWithDiscount (@discount)", param);
                foreach (var p in phones)
                    Console.WriteLine("{0} - {1}", p.Name, p.Price);
            }
            Console.ReadKey();
        }
```

Хранимые процедуры

Работа с хранимыми процедурами во многом аналогична работе с функциями базы данных. Итак, возьмем **БД** из прошлой темы и добавим в нее новую процедуру. Для этого откроем подключение к базе данных в окне **Database Explorer** и найдем в его структуре узел **Stored Procedures** (Хранимые процедуры). Нажмем на этот узел правой кнопкой мыши и в появившемся меню выберем **Add New Stored Procedure**:



После этого **Visual Studio** сгенерирует код процедуры по умолчанию:

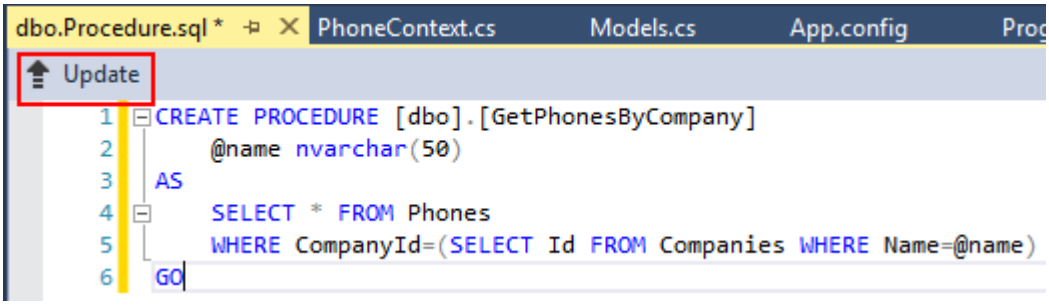
```
CREATE PROCEDURE [dbo].[Procedure]
      @param1 int = 0,
      @param2 int
AS
      SELECT @param1, @param2
RETURN 0
```

Изменим ее следующим образом:

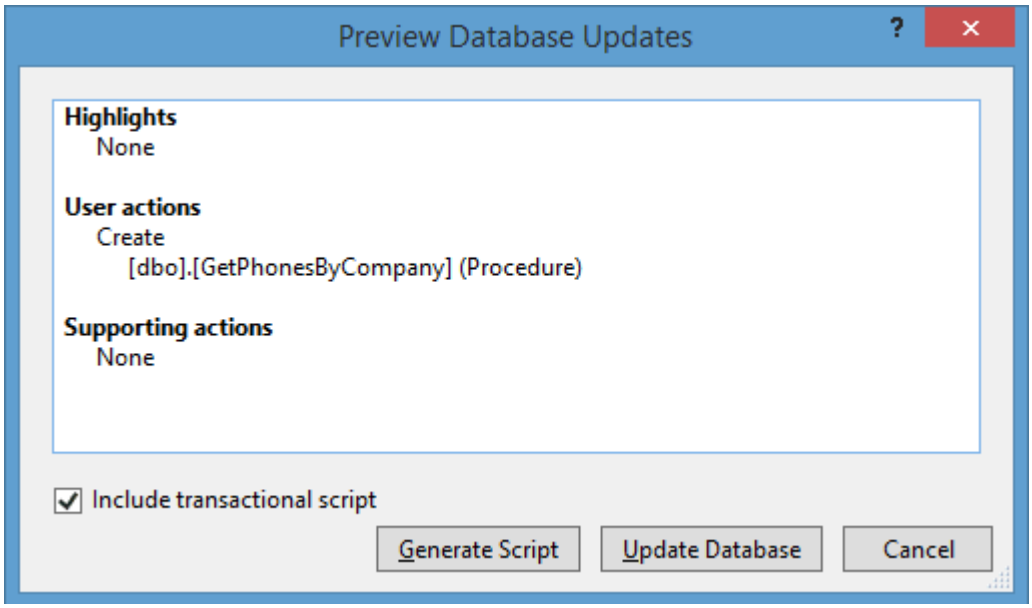
```
CREATE PROCEDURE [dbo].[GetPhonesByCompany]
    @name nvarchar(50)
AS
    SELECT * FROM Phones
    WHERE CompanyId=(SELECT Id FROM Companies WHERE Name=@name)
GO
```

Данная процедура ищет все строки, где значение столбца название компании равно строке, переданной через параметр **@name**.

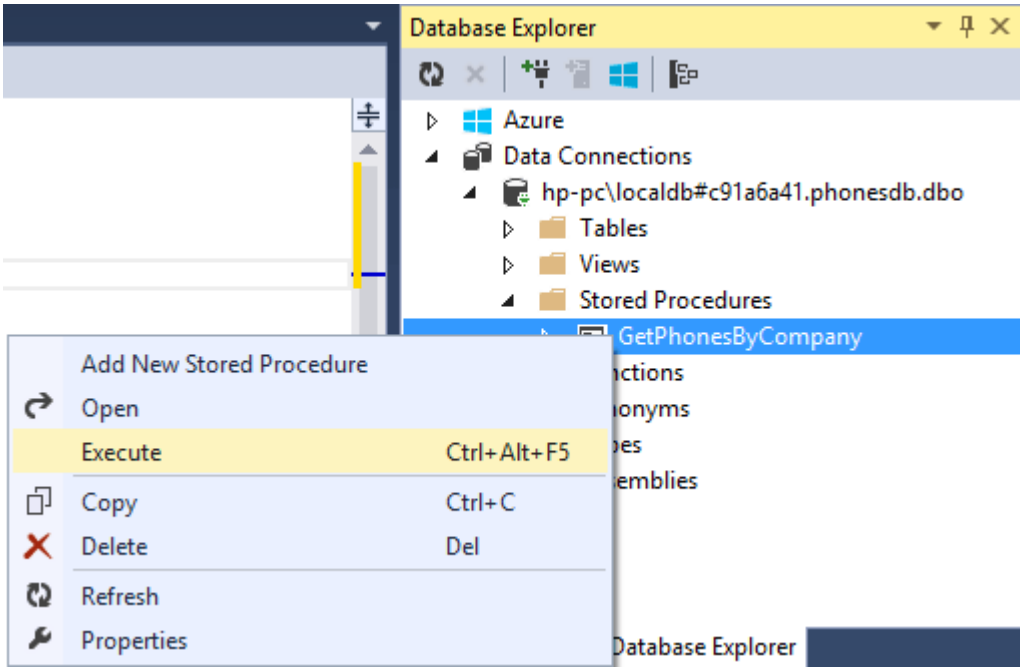
Далее нажмем на кнопку **Update** для добавления хранимой процедуры:



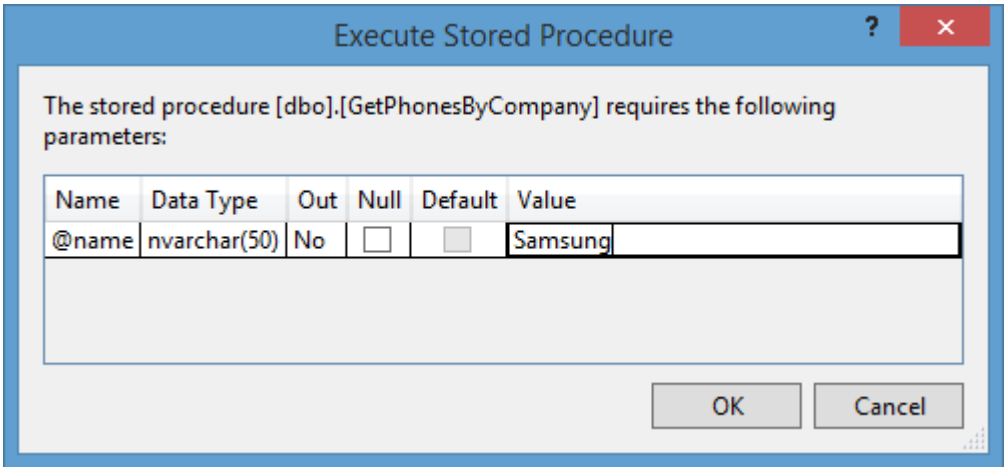
И затем в появившемся окошке нажимаем кнопку **Update Database**:



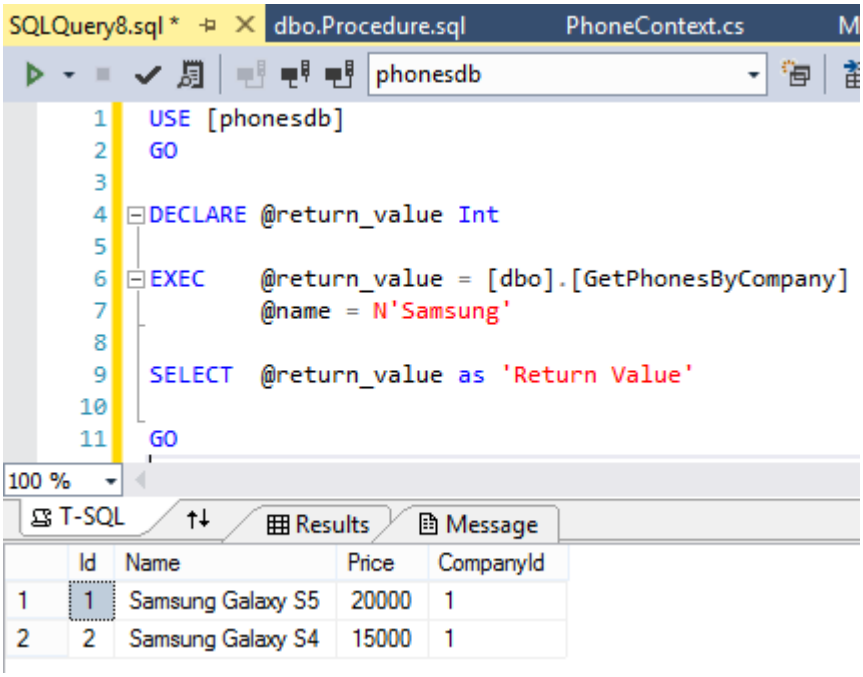
После этого в узле **Stored Procedures** появится новая хранимая процедура. Перед ее использованием мы можем проверить ее, чтобы убедиться, что она работает как надо и получает нужные данные. Для этого нажмем на нее правой кнопкой мыши и выберем в меню **Execute**:



После этого нам откроется окно настройки входного параметра. В поле **Value** введем какое-нибудь имя, по которому будем осуществлять поиск и нажмем **ОК**:



И **Visual Studio** автоматически генерирует и запускает скрипт на языке **SQL** с вызовом процедуры и передачи ей параметра:




```
CREATE PROCEDURE [dbo].[GetPhonesPrice]
    @bprice int,  @eprice int
AS
    SELECT * FROM Phones
    WHERE Price BETWEEN @bprice and @eprice
```

Теперь обратимся к процедуре в коде **C#**:

```
static void Main(string[] args)
        {
            using (PhoneContext db = new PhoneContext())
            {
                System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@name", "Samsung");
                var phones =
db.Database.SqlQuery<Phone>("GetPhonesByCompany @name", param);
                foreach (var p in phones)
                    Console.WriteLine("{0} - {1}", p.Name, p.Price);
            }
            Console.ReadKey();
        }
```

Параметр в методе **SqlQuery** принимает название процедуры, после которого идет перечисление параметров: **GetPhonesByCompany @name**

И так как процедура возвращает строки из таблицы **Phones**, то метод **SqlQuery** мы можем типизировать классом **Phone:db.Database.SqlQuery<Phone>(...)**

6.Fluent API и аннотации

Если мы используем подход **Code First**, то классы моделей сопоставляются с таблицами с помощью ряда правил в **Entity Framework**. Но иногда необходимо изменить и переопределить логику этих правил. Для этого используется **Fluent API** и аннотации данных.

Fluent API

Fluent API по большому счету представляет набор методов, которые определяются сопоставление между классами и их свойствами и таблицами и их столбцами. Как правило, функционал **Fluent API** задействуется при переопределении метода **OnModelCreating**:

```
class FluentContext : DbContext
    {
        public FluentContext()
            : base("DefaultConnection")
        { }

        public DbSet<Phone> Phones { get; set; }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            // использование Fluent API
            base.OnModelCreating(modelBuilder);
        }
    }
```

В качестве экспериментальной модели возьмем следующую модель:

```
class Phone
    {
        public int Ident { get; set; }
        public string Name { get; set; }
        public int Discount { get; set; }
        public int Price { get; set; }
    }
```

Сопоставление класса с таблицей

По умолчанию **EF** сопоставляет модель с одноименной таблицей, но мы можем переопределить это поведение с помощью метода**ToTable()**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
```

```
{
    modelBuilder.Entity<Phone>().ToTable("Mobiles");
    // использование Fluent API
    base.OnModelCreating(modelBuilder);
     }
```

Теперь все объекты **Phone** будут храниться в таблице **Mobiles**. Но мы также с ними сможем работать через свойство **db.Phones**.

Если по какой-то сущности нам не надо создавать таблицу, то мы можем ее проигнорировать с помощью метода **Ignore()**:

modelBuilder.Ignore<Company>();

Переопределение первичного ключа

По умолчанию в **Entity Framework** первичный ключ должен представлять свойство модели с именем **Id** или **[Имя_класса]Id**, например, **PhoneId**. Чтобы переопределить первичный ключ через **Fluent API**, надо использовать метод **HasKey()**:

```
modelBuilder.Entity<Phone>().HasKey(p => p.Ident);
```

В данном случае первичным ключом будет свойство **Ident** класса **Phone**. Чтобы настроить составной первичный ключ, мы можем указать два свойства:

```
modelBuilder.Entity<Phone>().HasKey(p => new { p.Ident, p.Name });
```

Сопоставление свойств

Чтобы сопоставить свойство с определенным столбцом, используется метод **HasColumnName()**:

```
modelBuilder.Entity<Phone>().Property(p =>
p.Name).HasColumnName("PhoneName");
```

В данном случае свойство **Name** будет сопоставляться со столбцом **PhoneName**.

Если мы не хотим, чтобы с каким-то свойством вообще шло сопоставление, то мы можем его исключить с помощью метода **Ignore()**:

```
modelBuilder.Entity<Phone>().Ignore(p => p.Discount);
```

Теперь свойство **Discount** класса **Phone** не будет сопоставляться ни с никаким столбцом из таблицы в **БД**.

Столбцы в таблице в **БД** могут допускать значение **NULL**, которое указывает, что значение не определено. По умолчанию все столбцы при **Code First**, если не применяются аннотации данных, за исключением идентификатора допускают значение **NULL**. Но мы можем указать с помощью метода **IsRequired()**, что значение для этого столбца и свойства требуется обязательно:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsRequired();
```

Если нам, наоборот, надо указать, чтобы столбец мог принимать значения **NULL**, то мы можем использовать метод **IsOptional()**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsOptional();
```

Настройка строк

Для строк мы модем указать максимальную длину с помощью метода **HasMaxLength()**. Например, длина не более **50** символов:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).HasMaxLength(50);
```

Также для строк можно определить, будут ли они храниться в кодировке **Unicode**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsUnicode(false);
```

Параметр **false** указывает, что строки будут храниться не в **Unicode**-кодировке.

Настройка чисел decimal

Если у нас есть свойство с типом **decimal**, то мы можем указать для него точность число цифр в числе и число цифр после запятой:

```
// допустим, свойство Price - decimal
modelBuilder.Entity<Phone>().Property(p => p.Price).HasPrecision(15, 2);
```

Теперь число **decimal** может содержать до **15** цифр и **2** цифры после запятой. Если же мы не указываем, то действуют значения по умолчанию **- 18** и **2**.

Настройка типа столбцов

По умолчанию **EF** сам выбирает тип данных в **БД**, исходя из типа данных свойства. Но мы также можем явно указать, какой тип данных в **БД** должен использоваться для столбца с помощью метода **HasColumnType()**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).HasColumnType("varchar");
```

Сопоставление модели с несколькими таблицами

С помощью **Fluent API** мы можем поместить ряд свойств модели в одну таблицу, а другие свойства связать со столбцами из другой таблицы:

```
modelBuilder.Entity<Phone>().Map(m =>
            {
                m.Properties(p => new { p.Ident, p.Name });
                m.ToTable("Mobiles");
            })
.Map(m =>
{
    m.Properties(p => new { p.Ident, p.Price, p.Discount });
    m.ToTable("MobilesInfo");
});
```

Таким образом, данные для свойства **Name** будут храниться в таблице **Mobiles**, а данные для свойств **Price** и **Discount** - в таблице **MobilesInfo**. И столбец идентификатора будет общим.

Отношения между моделями в Fluent API

Связь один-к нулю или - к одному (One-to–Zero-or-One)

При такой связи для одной модели наличие другой необязательно. Например:

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

        public Company Company { get; set; }
    }

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

```
        public Phone BestSeller { get; set; }
    }
```

Смартфон обязательно имеет производителя, но производитель может не иметь наиболее продаваемого телефона. То есть в данном случае связь один-к нулю или ко многим. В **Fluent API** она устанавливается следующим образом:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Phone>()
            .HasRequired(c => c.Company)
            .WithOptional(c => c.BestSeller);

            // использование Fluent API
            base.OnModelCreating(modelBuilder);
        }
```

Метод **HasRequired()** указывает, что для сущности **Phone** обязательно должно быть указано навигационное свойство **Company**. А метод **WithOptional()**, наоборот, устанавливает необязательную связь между объектом предыдущего выражения - **Company** и его свойством **BestSeller**.

Связь один-к-одному (One-to-One)

В данной конфигурации уже оба объекта связи должны иметь ссылку друг на друга:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Phone>()
            .HasRequired(c => c.Company)
            .WithRequiredPrincipal(c => c.BestSeller);
            //или так
            //modelBuilder.Entity<Company>()
            //    .HasRequired(c => c.BestSeller)
            //    .WithRequiredPrincipal(c => c.Company);


            // использование Fluent API
            base.OnModelCreating(modelBuilder);
        }
```

Метод **WithRequiredPrincipal()** настраивает обязательную связь и устанавливает одну из сущностей в качестве основной. Так, в данном случае

основной сущность устанавливается модель **Phone: WithRequiredPrincipal(c => c.BestSeller)**. А таблица, на которую отображается модель **Company**, будет содержать внешний ключ к таблице **Phones**.

Связь многие-ко-многим (many-tomany)

Пусть у нас есть ситуация, когда любая из моделей содержит список объектов другой модели. Например, компания может производить несколько телефонов, а над одним телефоном могут работать несколько компаний:

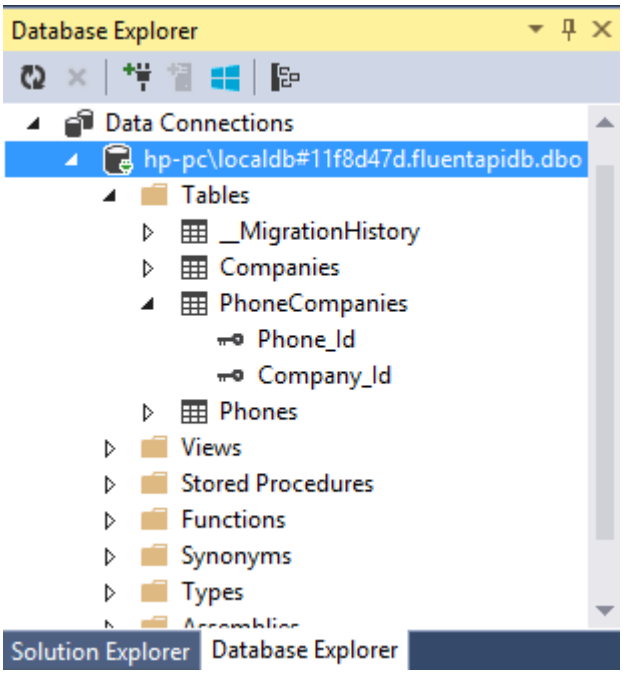
```
public class Phone
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public ICollection<Company> Companies { get; set; }

        public Phone()
        {
            Companies = new List<Company>();
        }
    }

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

        public ICollection<Phone> Phones { get; set; }

        public Company()
        {
            Phones = new List<Phone>();
        }
    }
```

Тогда настройка связи между ними будет выглядеть следующим образом:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Phone>()
            .HasMany(p => p.Companies)
            .WithMany(c => c.Phones);
            base.OnModelCreating(modelBuilder);
        }
```

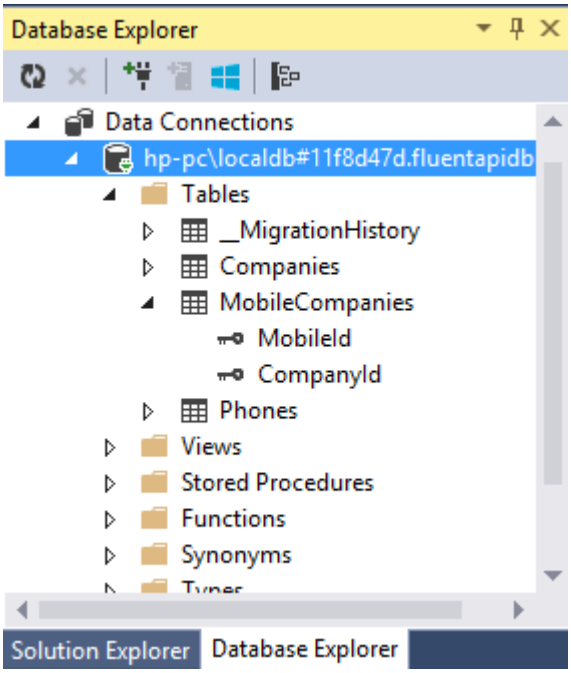
HasMany() устанавливает множественную связь между объектом **Phone** и объектами **Company**. А метод **WithMany()** добавляет обратную множественную связь между объектом **Company** и объектами **Phone**.

В результате при работе с базой данных будет сформирована третья таблица-посредник между двумя сущностями:



Но нас может не устраивать подобное название таблицы и ее столбцов, и мы можем стандартное поведение переопределить следующим образом:

```
modelBuilder.Entity<Phone>()
            .HasMany(p => p.Companies)
            .WithMany(c => c.Phones)
            .Map(m =>
            {
                m.ToTable("MobileCompanies");
                m.MapLeftKey("PhoneId");
                m.MapRightKey("CompanyId");
            });
```




```
using (FluentContext db = new FluentContext())
            {
                var items = from i in db.Phones
                            join fi in db.Companies on i.Id equals fi.Id
                            where fi.Id == 1
                            select i;
                foreach (var item in items)
                {
                    Console.WriteLine(item.Name);
                }
            }
```

Связь один-ко-многим (One-to-Many)

При связи один-ко-многим одна модель может ссылаться на множество объектов другой модели. Например, одна компания производит множество телефонов:

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

        public Company Company { get; set; }
    }

    public class Company
```

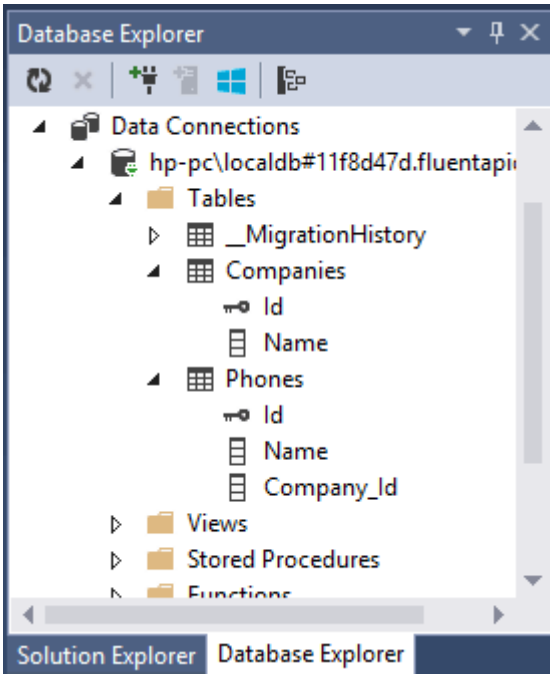
```
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public ICollection<Phone> Phones { get; set; }
        public Company()
        {
            Phones = new List<Phone>();
        }
    }
```

А сама связь через **Fluent API** будет выглядеть так:

```
modelBuilder.Entity<Company>()
.HasMany(p => p.Phones)
.WithRequired(p => p.Company);
```

Метод **HasMany()** устанавливает множественную связь между объектом **Company** и объектами **Phone**, а метод **WithRequired()** требует обязательной установки свойства **Company** у класса **Phone**.

После генерации таблиц таблица для моделей **Phone** будет содержать столбец-внешний ключ **Company_Id** для связи с таблицей **Companies**:



Настройка внешнего ключа

Возможно, нас не устраивает такое название столбца и внешнего ключа, которое дается **EF** по умолчанию. С помощью метода **HasForeignKey()** мы

можем переопределить действие по умолчанию. Для этого определим свойство, которое будет представлять внешний ключ:

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

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

Теперь настроим свойство **Manufacturer** в качестве внешнего ключа к таблице **Companies**:

```
modelBuilder.Entity<Company>()
.HasMany(p => p.Phones)
.WithRequired(p => p.Company)
.HasForeignKey(s => s.Manufacturer);
```

Отключение каскадного удаления

По умолчанию, если свойство-внешний ключ зависимой сущности не представляет собой тип **Nullable**, то в таблице для этой сущности устанавливается каскадное удаление, по удалению главной сущности. То есть в предыдущем примере свойство **Manufacturer**, которое выполняет роль внешнего ключа, имеет тип **int**. Поэтому при генерации таблицы будет действовать правило **ON DELETE CASCADE**.

Если бы у нас свойство **Manufacturer** представляло бы тип **int?**, а вместо метода **WithRequired** использовался бы метод **`WithOptional()`**

```
(modelBuilder.Entity<Company>().HasMany(p =>
p.Phones).WithOptional(p=>p.Company)),
```

который не требует наличия внешнего ключа, то каскадное удаление бы не добавлялось в таблицу.

И чтобы через **Fluent API** отключить каскадное удаление, надо использовать метод **WillCascadeOnDelete(false)**:

```
modelBuilder.Entity<Company>()
.HasMany(p => p.Phones)
.WithRequired(p => p.Company)
.HasForeignKey(s => s.Manufacturer)
.WillCascadeOnDelete(false);
```

Соответственно, если используется **true**, то каскадное удаление, наоборот, включается: **WillCascadeOnDelete(true)**

А также можно использовать дополнительные методы для отключения каскадного удаления при отдельных видах отношений:

```
using System.Data.Entity.ModelConfiguration.Conventions;
……
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();
```

Аннотации

Аннотации представляют настройку сопоставления моделей и таблиц с помощью атрибутов. Большинство классов аннотаций располагаются в пространстве **System.ComponentModel.DataAnnotations**, которое нам надо подключить в файл **C#** перед использованием аннотаций.

Настройка ключа

Для установки свойства в качестве первичного ключа применяется атрибут **[Key]**:

```
public class Phone
    {
        [Key]
        public int Ident { get; set; }
        public string Name { get; set; }
    }
```

Теперь свойство **Ident** будет рассматриваться в качестве первичного ключа. Чтобы установить ключа в качестве идентификатора, можно использовать атрибут **DatabaseGenerated(DatabaseGeneratedOption.Identity):**

```
[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int Ident { get; set; }
```

Если свойство, помеченное атрибутом **Key**, представляет тип **int**, то к нему атрибут **DatabaseGenerated(DatabaseGeneratedOption.Identity)** будет применяться автоматически, и его можно в принципе не указывать.

Атрибут Required

Атрибут **Required** указывает, что данное свойство обязательно для установки, то есть будет иметь определение **NOT NULL** в **БД**:

```
public class Phone
    {
        [Key]
        public int Ident { get; set; }
        [Required]
        public string Name { get; set; }
    }
```

Если мы не установим свойство **Name** у объекта **Phone** и попытаемся добавить этот объект в **БД**, то получим ошибку. А столбец **Name** будет определен как **NOT NULL**.

MaxLength и MinLength

Атрибуты **MaxLength** и **MinLength** устанавливают максимальное и минимальное количество символов в строке-свойстве:

```
public class Phone
    {
        [Key]
        public int Ident { get; set; }
        [MaxLength(20)]
        public string Name { get; set; }
    }
```

Атрибут NotMapped

По умолчанию все публичные свойства сопоставляются с определенными столбцами в таблицами. Но такое поведение не всегда необходимо. Иногда требуется, наоборот, исключить определенное свойство, чтобы для него не создавался столбец в таблице. И для этих целей есть атрибут **NotMapped**:

```
public class Phone
    {
        [Key]
        public int Ident { get; set; }
        [Required]
```

```
        public string Name { get; set; }
        [NotMapped]
        public int Discount { get; set; }
    }
```

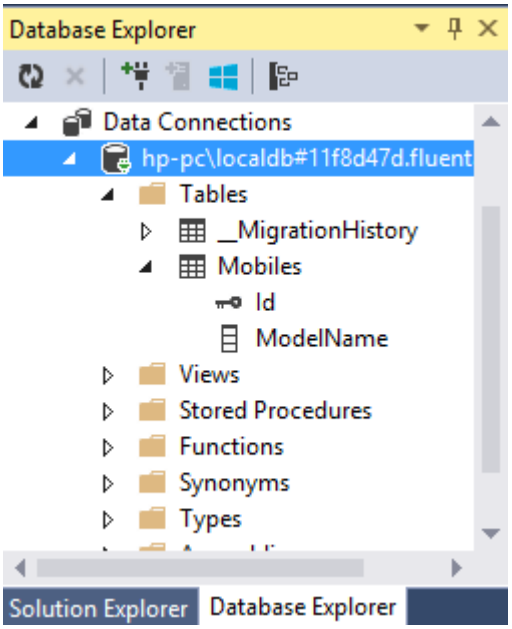
Чтобы задействовать атрибут **NotMapped**, надо подключить пространство имен **System.ComponentModel.DataAnnotations.Schema.**

Сопоставление с таблицей и столбцами

Entity Framework при создании и сопоставлении таблиц и столбцов использует имена моделей и их свойств. Но мы можем переопределить это поведение с помощью атрибутов **Table** и **Column**:

```
[Table("Mobiles")]
public class Phone
    {
        public int Id { get; set; }
        [Column("ModelName")]
        public string Name { get; set; }
    }
```

Теперь сущность **Phone** будет сопоставляться с таблицей **Mobiles**, а свойство **Name** со столбцом **ModelName**:



Установка внешнего ключа

Чтобы установить внешний ключ для связи с другой сущностью, используется атрибут **ForeignKey**:

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

        public int? CompId { get; set; }
        [ForeignKey("CompId")]
        public Company Company { get; set; }
    }

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

В данном случае внешним ключом для связи с моделью **Company** будет служить свойство **CompId**.

ConcurrencyCheck

Атрибут **ConcurrencyCheck** позволяет решить проблему параллелизма, когда с одной и той же записью в таблице могут работать одновременно несколько пользователей. Например:

```
public class Phone
    {
        public int Id { get; set; }
        [ConcurrencyCheck]
        public string Name { get; set; }
    }
```

Теоретически возможна ситуация, когда два пользователя пытаются изменить значение, например:

```
using (FluentContext db = new FluentContext())
            {
                Phone phone = db.Phones.Find(1);
                phone.Name = "Nokia N9";
```

```
            db.Entry(phone).State = EntityState.Modified;
            db.SaveChanges();
        }
```

В обычном режиме **Entity Framework** при обновлении смотрит на **Id** и если **Id** записи в таблице совпадает с **Id** в передаваемой модели **phone**, то строка в таблице обновляется. При использовании атрибута **ConcurrencyCheck EF** смотрит не только на **Id**, но и на исходное значение свойства **Name**. И если оно совпадает с тем, что имеется в таблице, то запись обновляется. Если же не совпадает (то есть кто-то уже успел обновить), то **EF** генерирует исключение **DbUpdateConcurrencyException**.

Работа с комплексными типами

Каждый тип сопоставляется **Entity Framework** с отдельной таблицей. Однако бывают случаи, когда определенный класс существует не сам по себе, а несет некоторую дополнительную информацию по отношению к другой главной модели. Например:

```
public class PhoneInfo
    {
        public string Company { get; set; }
        public int Price { get; set; }
    }

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

        public PhoneInfo Info { get; set; }

        public Phone()
        {
            Info = new PhoneInfo { Price = 400000 };
        }
    }
```

Класс **PhoneInfo** содержит некоторую дополнительную информацию по отношению к модели **Phone**, а модель **Phone** создает объект **PhoneInfo** с данными по умолчанию. Таким типы еще называют комплексными.

Комплексные типы имеют ряд ограничений:

- Они не имеют ключей
- Они могут содержать свойства только примитивных типов
- При использовании моделями модель может использовать только один объект типа, но не коллекцию (то есть коллекцию **PhoneInfo** модель **Phone** содержать не может)

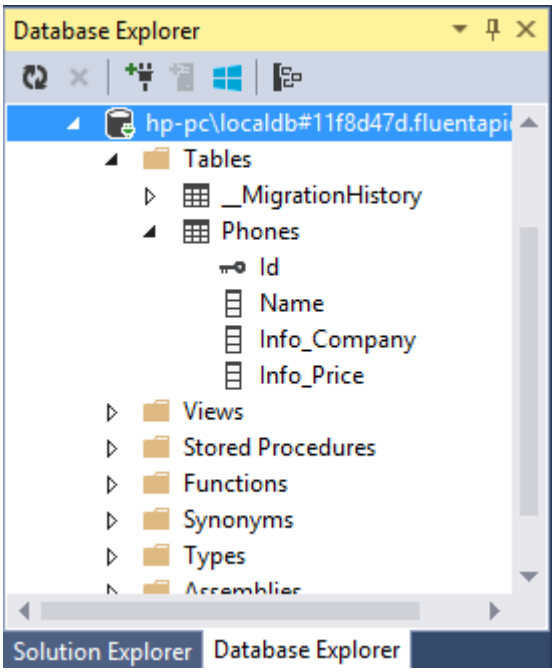
Чтобы обозначить связь между двумя типами, необходимо использовать атрибут **ComplexType**. Он позволяет создать из разных классов одну сущность. Например:

```
[ComplexType]
public class PhoneInfo
    {
        public string Company { get; set; }
        public int Price { get; set; }
    }
```

Класс **Phone** остается без изменений. Тогда мы можем использовать данные типы следующим образом:

```
using (FluentContext db = new FluentContext())
            {
                db.Phones.Add(new Phone
                {
                    Name = "Samsung Galaxy S5",
                    Info = new PhoneInfo { Company = "Samsung", Price =
17000 }
                });
                db.Phones.Add(new Phone
                {
                    Name = "Nokia Lumia 930",
                    Info = new PhoneInfo { Company = "Nokia", Price =
15000 }
                });
                db.SaveChanges();
                foreach (Phone p in db.Phones)
                    Console.WriteLine("{0} - {1}", p.Name, p.Info.Price);
            }
```

Тогда при создании таблицы она будет объединять столбцы для обоих моделей, представляя как бы единую сущность:



Альтернативный вариант применению атрибута представляет использование метода **ComplexType** в :

```
class FluentContext : DbContext
    {
        public FluentContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.ComplexType<PhoneInfo>();
            base.OnModelCreating(modelBuilder);
        }
    }
```

Две модели в одной таблице

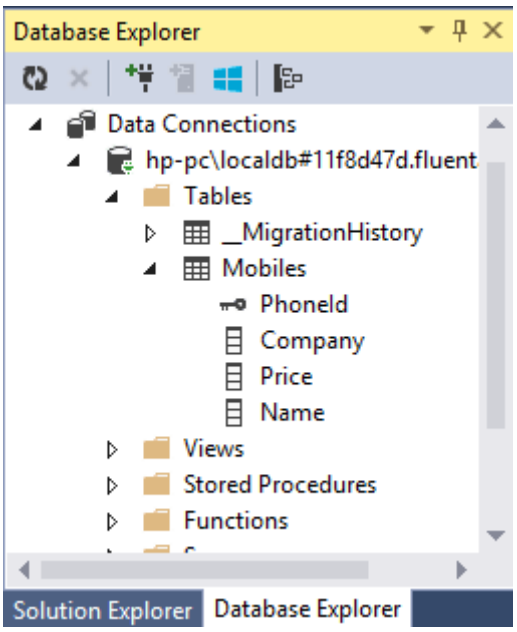
В предыдущей теме два класса по сути относились к одной модели и хранились в одной таблице. Рассмотрим другую конфигурацию. Две модели связаны между собой, и было бы логично хранить их в одной таблице. В то же

время мы можем оставить возможность работать с ними независимо друг от друга.

Чтобы две модели сохранялись в одну таблицу, они должны иметь между собой связь одни-ко-одному. А также иметь общий ключ. Например:

```
using System;
using System.Text;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace FluentAPIApp
{
    [Table("Mobiles")]
    public class PhoneInfo
    {
        [Key, ForeignKey("Phone")]
        public int PhoneId { get; set; }
        public string Company { get; set; }
        public int Price { get; set; }

        public Phone Phone { get; set; }
    }

    [Table("Mobiles")]
    public class Phone
    {
        [Key, ForeignKey("Info")]
        public int PhoneId { get; set; }
        public string Name { get; set; }

        public PhoneInfo Info { get; set; }
    }

    class MobileContext : DbContext
    {
        public MobileContext()
            : base("DefaultConnection")
        { }

        public DbSet<Phone> Phones { get; set; }
        public DbSet<PhoneInfo> Infos { get; set; }
    }
}
```

Модели **Phone** и **PnoneInfo** имеют один и тот же первичный ключ, который также выполняет роль внешнего ключа. В итоге сформируется следующая таблица:



Использование моделей:

```
using (MobileContext db = new MobileContext())
            {
                PhoneInfo pi1 = new PhoneInfo { PhoneId = 5, Company =
"Samsung", Price = 14000 };
                PhoneInfo pi2 = new PhoneInfo { PhoneId = 6, Company =
"Nokia", Price = 8000 };

                Phone p1 = new Phone { PhoneId = 5, Name = "Samsung
Galaxy S5", Info = pi1 };
                Phone p2 = new Phone { PhoneId = 6, Name = "Nokia Lumia
630", Info = pi2 };

                db.Infos.Add(pi1);
                db.Infos.Add(pi2);
                db.Phones.Add(p1);
                db.Phones.Add(p2);
                db.SaveChanges();

                foreach (Phone p in db.Phones.Include(p => p.Info))
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Info.Company, p.Info.Price);
            }
```

```
Результатом работы программы будет следующий вывод:
Samsung Galaxy S5 (Samsung) - 14000
Nokia Lumia 630 (Nokia) - 8000
```

Разделение сущности на несколько таблиц

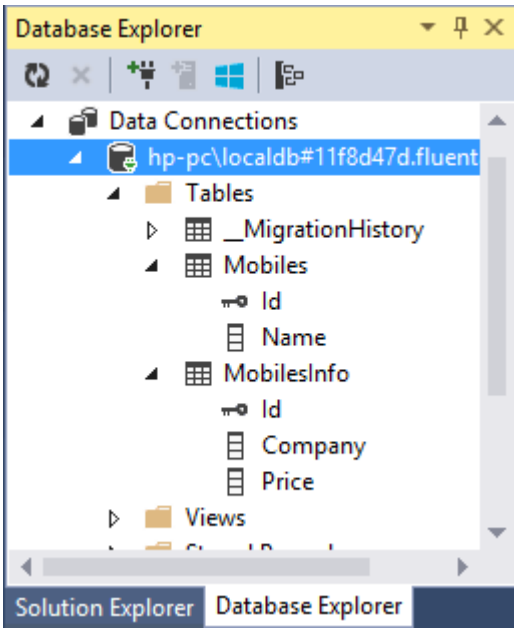
В предыдущей теме два класса объединялись в одну таблицу. Но мы также можем сделать и обратное действие - сохранить разные свойства одного класса в разных таблицах. Например, нам потребовалось важную информацию о модели сохранить в одной таблице, а вспомогательную информацию - в другой.

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

class MobileContext : DbContext
    {
        public MobileContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Phone>().Map(m =>
            {
                m.Properties(d => new { d.Name, d.Id });
                m.ToTable("Mobiles");
            }).Map(m =>
            {
                m.Properties(d => new { d.Company, d.Price });
                m.ToTable("MobilesInfo");
            }); ;
            base.OnModelCreating(modelBuilder);
        }
    }
```

С помощью метода **Map** осуществляется сопоставление отдельных свойств с таблицами. Так, данные для свойства **Name** будут храниться в

таблице **Mobiles**, а данные для свойств **Company** и **Price** - в таблице **MobilesInfo**. **Id** мы можем опустить, так как данное поле будет указываться для обеих таблиц для их связи.

В итоге сформируются две таблицы:



Работа с моделью будет проходить аналогично стандартным моделям:

```
using (MobileContext db = new MobileContext())
            {
                Phone p1 = new Phone { Id = 1, Name = "Samsung Galaxy
S5", Company = "Samsung", Price = 14000 };
                Phone p2 = new Phone { Id = 2, Name = "Nokia Lumia 630",
Company = "Nokia", Price = 8000 };

                db.Phones.Add(p1);
                db.Phones.Add(p2);
                db.SaveChanges();

                foreach (Phone p in db.Phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
            }
```

7. Наследование в Entity Framework

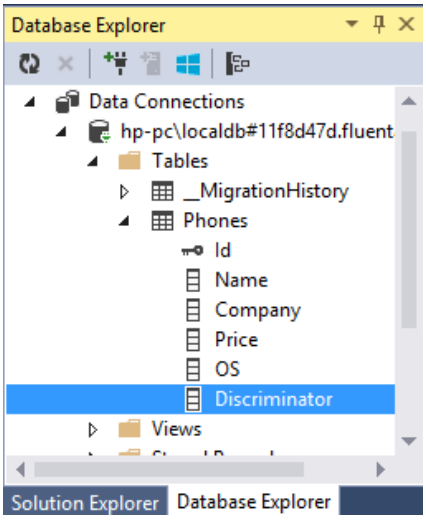
Подход TPH

При использовании подхода **TPH** (**Table Per Hierarchy** / **Таблица** на одну иерархию классов) для одной иерархии классов используется одна таблица. Данные базовых и производных классов сохраняются в одну таблицу, а для их отличия создается специальный столбец.

Например, создадим мини-иерархию из двух классов:

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

    public class Smartphone : Phone
    {
        public string OS { get; set; }
    }

    class MobileContext : DbContext
    {
        public MobileContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }
        public DbSet<Smartphone> Smarts { get; set; }
    }
```

Здесь класс **Smartphone** наследуется от **Phone**, определяя одно свойство в дополнение к унаследованным. И при работе будет создана такая таблица:

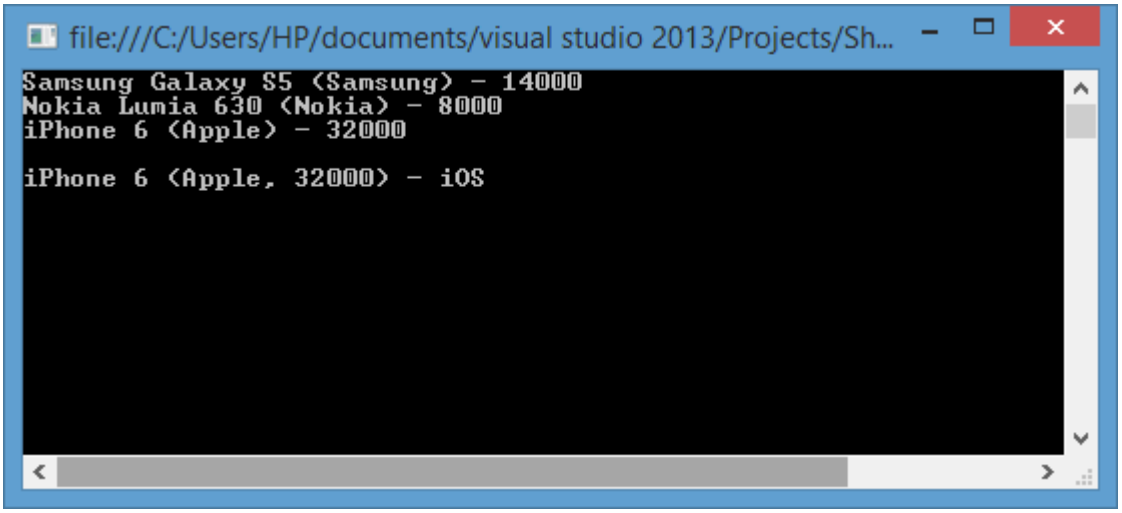


Кроме всех свойств классов **Phone** и **Smartphone** здесь также появляется еще один столбец - **Discriminator**. Он имеет тип **nvarchar** (то есть строка) и имеет длину в **128** символов. Данный столбец и будет определять относится строка к типу **Phone** или **Smartphone**.

Теперь используем в программе:

```
using (MobileContext db = new MobileContext())
            {
                db.Phones.Add(new Phone { Name = "Samsung Galaxy S5",
Company = "Samsung", Price = 14000 });
                db.Phones.Add(new Phone { Name = "Nokia Lumia 630",
Company = "Nokia", Price = 8000 });

                Smartphone s1 = new Smartphone { Name = "iPhone 6",
Company = "Apple", Price = 32000, OS = "iOS" };
                db.Smarts.Add(s1);
                db.SaveChanges();

                foreach (Phone p in db.Phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
                Console.WriteLine();
                foreach (Smartphone p in db.Smarts)
                    Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name,
p.Company, p.Price, p.OS);
            }
```

Также обратите внимание на вывод консоли:



Так как объект **Smartphone** также является и объектом **Phone**, то он также извлекается через **db.Phones**.

Подход TPT

Подход TPT (Table Per Type / Таблица на тип) предполагает сохранение в общей таблице только тех свойств, которые общие для всех классом-наследников, то есть которые определены в базовом классе. А те свойства, которые относятся только к производному классу, сохраняются в отдельной таблице.

Чтобы применить подход, возьмем из предыдущего примера систему классов и добавим к классу **Smartphone** атрибут **Table**:

```
public class Phone
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public int Price { get; set; }
    }
[Table("Smartphones")]
public class Smartphone : Phone
    {
        public string OS { get; set; }
    }

class MobileContext : DbContext
    {
        public MobileContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }
```

```
        public DbSet<Smartphone> Smarts { get; set; }
    }
```

Все остальное остается также, как и при подходе **TPH**. Но теперь база данных будет содержать следующие таблицы:

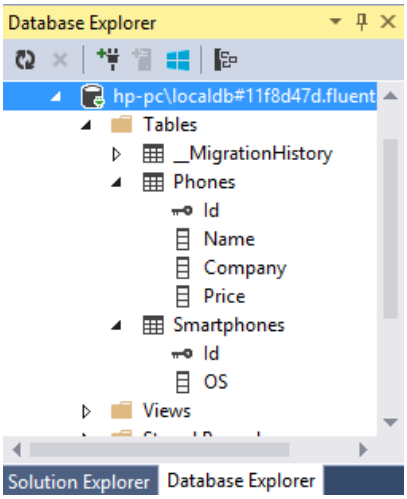


Таблица для смартфонов содержит только оно поле **OS**, а также ключ **Id** для связи с таблицей **Phones**.

Применение моделей будет аналогично подходу **TPH**:

```
using (MobileContext db = new MobileContext())
            {
                db.Phones.Add(new Phone { Name = "Samsung Galaxy S5",
Company = "Samsung", Price = 14000 });
                db.Phones.Add(new Phone { Name = "Nokia Lumia 630",
Company = "Nokia", Price = 8000 });

                Smartphone s1 = new Smartphone { Name = "iPhone 6",
Company = "Apple", Price = 32000, OS = "iOS" };
                db.Smarts.Add(s1);
                db.SaveChanges();

                foreach (Phone p in db.Phones)
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
                Console.WriteLine();
                foreach (Smartphone p in db.Smarts)
                    Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name,
p.Company, p.Price, p.OS);
            }
```

Подход TPC

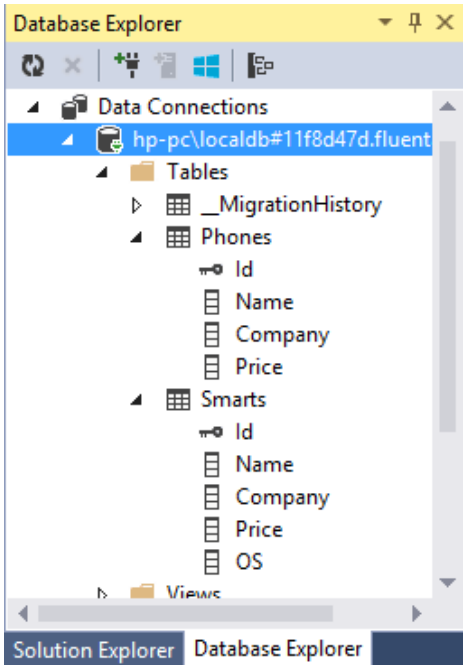
Подход **TPC (Table Per Concrete Type / Таблица на каждый отдельный тип)** предполагает создание для каждой модели по отдельной таблицы. Столбцы в каждой таблице создаются по всем свойствам, в том числе и унаследованным.

Чтобы применить подход, изменим объявления моделей и контекст следующим образом:

```
public class Phone
    {
        [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public Guid Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public int Price { get; set; }
    }

public class Smartphone : Phone
    {
        public string OS { get; set; }
    }

    class MobileContext : DbContext
    {
        public MobileContext()
            : base("DefaultConnection")
        { }
        public DbSet<Phone> Phones { get; set; }
        public DbSet<Smartphone> Smarts { get; set; }
        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Phone>()
            .Map(m =>
            {
                m.MapInheritedProperties();
                m.ToTable("Phones");
            });
            modelBuilder.Entity<Smartphone>().Map(m =>
            {
                m.MapInheritedProperties();
                m.ToTable("Smarts");
            });
        }
    }
```

Во-первых, обратите внимание, что у класса **Phone** в качестве типа ключа используется не **int**, а **Guid**. Это поможет нам избежать некоторых проблем с ключами. Хотя также можно было бы использовать **int** с ручной установкой **Id** при создании объекта.

Во-вторых, при настройке сопоставления моделей и таблиц у каждой модели вызывается метод **MapInheritedProperties()**, который указывает **Entity Frameworky**, что надо переопределить связи между моделями при наследовании. При генерации базы данных у нас будут созданы две таблицы с полным набором столбцов:



Применение моделей:

```
using (MobileContext db = new MobileContext())
            {
                db.Phones.Add(new Phone { Name = "Samsung Galaxy S5",
Company = "Samsung", Price = 14000 });
                db.Phones.Add(new Phone { Name = "Nokia Lumia 630",
Company = "Nokia", Price = 8000 });

                Smartphone s1 = new Smartphone { Name = "iPhone 6",
Company = "Apple", Price = 32000, OS = "iOS" };
                db.Smarts.Add(s1);
                db.SaveChanges();

                foreach (Phone p in db.Phones)
```

```
                    Console.WriteLine("{0} ({1}) - {2}", p.Name,
p.Company, p.Price);
                Console.WriteLine();
                foreach (Smartphone p in db.Smarts)
                    Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name,
p.Company, p.Price, p.OS);
            }
```

Несмотря на то, что объект **Smartphone** никак не связан с таблицей **Phones**, при извлечении данных он также будет находится в наборе **db.Phones**, потому что наследование все равно будет действовать.

8. Асинхронность в Entity Framework

Асинхронные операции

Начиная с версии **6.0 Entity Framework** поддерживает асинхронные операции. Для сохранения результатов в базе данных в асинхронном режиме используется метод **SaveChangesAsync**.

Чтобы получить объект по **Id** в асинхронном режиме, в классе **DbSet** определен метод **FindAsync**.

Некоторые методы **Linq to Entities** также имеют асинхронных двойников для осуществления запросов в асинхронном режиме:

- **ForEachAsync**: асинхронное извлечение данных и выполнение над ними определенных действий
- **AllAsync**: удовлетворяет ли все элементы в выборке определенному условию
- **AnyAsync**: удовлетворяет ли хотя бы один элемент выборки определенному условию
- **AverageAsync**: асинхронное получение среднего значения
- **ContainsAsync**: содержит ли выборка определенный элемент
- **CountAsync**: получение размера выборки
- **FirstAsync**: получение первого элемента
- **FirstOrDefaultAsync**: получение первого элемента или значения по умолчанию
- **LoadAsync**: асинхронная загрузка данных в кэш

- **MaxAsync**: получение максимального значения
- **MinAsync**: получение минимального значения
- **SingleAsync**: получение одного элемента
- **SingleOrDefaultAsync**: получение одного элемента или значения по умолчанию
- **SumAsync**: асинхронное получение суммы значений

Все методы возвращают объект задачи **Task** или **Task<T>**.

Например, выполним асинхронное сохранение и асинхронную выборку из **БД**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        Phone p = new Phone { Name = "Nokia Lumia 930", Price = 13000 };

        SaveObjectsAsync(p).Wait();

        Task t = GetObjectsAsync();
        t.Wait();

        Console.Read();
    }
    public static async Task GetObjectsAsync()
    {
        using (MobileContext db = new MobileContext())
        {
            await db.Phones.ForEachAsync(p =>
            {
                Console.WriteLine("{0} ({1})", p.Name, p.Price);
            });
        }
    }

    private static async Task SaveObjectsAsync(Phone p)
    {
        using (MobileContext db = new MobileContext())
        {
            db.Phones.Add(p);
```

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

Кроме асинхронных операций **Linq to Entities** нам доступно асинхронное осуществление команд в **БД** с помощью метода **ExecuteSqlCommandAsync**:

```
private static async Task DbCommandAsync(Phone p)
    {
        using (MobileContext db = new MobileContext())
        {
            System.Data.SqlClient.SqlParameter name = new
System.Data.SqlClient.SqlParameter("name", p.Name);
            System.Data.SqlClient.SqlParameter price = new
System.Data.SqlClient.SqlParameter("price", p.Price);
            await db.Database.ExecuteSqlCommandAsync("INSERT INTO Phones
(Name, Price) VALUES (@name, @price)", name, price);
        }
    }
```

Применение:

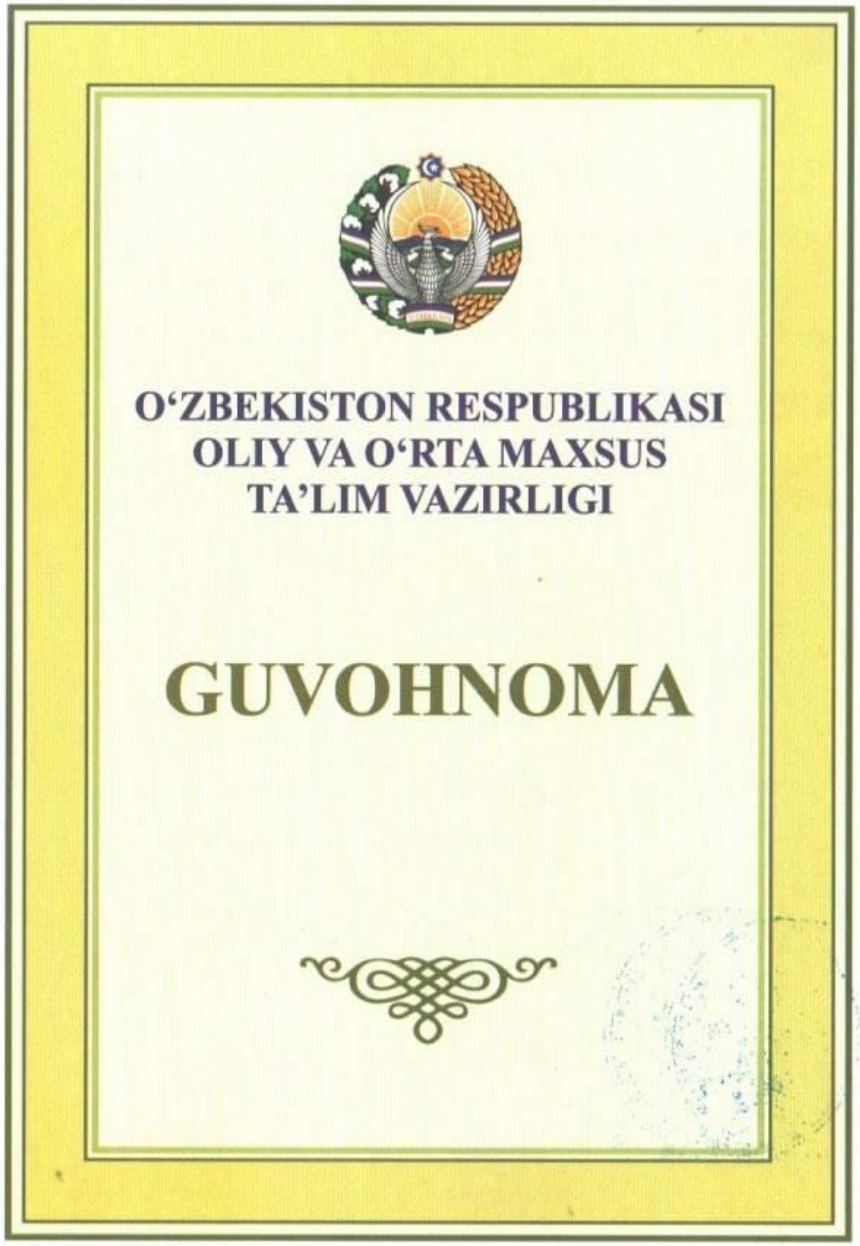
```
Phone p2 = new Phone { Name = "iPhone 6", Price = 33000 };
bCommandAsync(p2).Wait();
```

Заключение

Предмет программной инженерии учит синтаксису программной инженерии и работе на необходимых платформах для создания практических программ на нем, для создания программного обеспечения произвольной сложности, для создания современного распределенного программного обеспечения. Далее в рамках этого курса студенты изучают технологии программирования для работы MVC 5 и проектами в них и получают возможность эффективно использовать их в практической работе, исследованиях, а также в системе образования.

Использованная литература

1. Roger Pressman, Bruce Maxim, Software Engineering: A Practitioner's Approach, John Wiley & Sons, USA 2014.
2. Ian Sommerville. Software Engineering Hardcover. Pearson 2010 USA
3. Robert W. Sebesta, Concepts of Programming Languages, John Wiley & Sons, USA 2015.
4. Fundamentals of Computer Programming With C# (The Bulgarian C# Programming Book). Svetlin Nakov & Co., 2013.
5. Шилдт, Герберт. C# 4.0: полное руководство. : Пер. с англ. — М. : ООО "И.Д. Вильяме", 2011.



O‘ZBEKISTON RESPUBLIKASI
OLIY VA O‘RTA MAXSUS
TA’LIM VAZIRLIGI

GUVOHNOMA

O‘QUV ADABIYOTINING
NASHR RUXSATNOMASI

O‘zbekiston Respublikasi Oliy va o‘rta maxsus ta’lim vazirligining 2022 yil “ 9 ” sentabr dagi “ 302 ” -sonli buyrug‘iga asosan

Д.Т.Мухуммадиева, Л.П.Варламова, С.А.Бахромов, Б.Б.Елов
(muallifning familiyasi, ismi-sharifi)

5330100-Axborot tizimlarining matematik va dasturiy ta’minoti

(ta’lim yo’nalishi (mutaxassisligi))

ning

talabalari (o‘quvchilari) uchun tavsiya etilgan

Программная инженерия. Entity freamwork 6
(o‘quv adabiyotining nomi va turi: darslik, o‘quv qo‘llanma)

nomli o‘quv qo‘llanmasi

ga

O‘zbekiston Respublikasi Vazirlar Mahkamasi tomonidan litsenziya berilgan nashriyotlarda nashr etishga ruxsat berildi.

Vazir _____ A.Toshkulov
(imzo)

M.O.

Ro‘yxatga olish raqami
302-0397