**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 этой модели генерирует базу данных и ее таблицы

Создаем класс

public class User

{

    public int Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

}

Создаем класс с контекстом

 class UserContext : DbContext

    {

        public UserContext()

            :base("DbConnection")

        { }

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

    }

Добавляем строку подключения

После закрывающего тега </configSections> добавим следующий элемент:

|  |  |
| --- | --- |
| 1  2  3  4 | <connectionStrings>     <add name="DBConnection" connectionString="data source=(localdb)\MSSQLLocalDB;Initial Catalog=userstore.mdf;Integrated Security=True;"  providerName="System.Data.SqlClient"/>   </connectionStrings> |

Робота с базой данных

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();

        }

**Code Second**

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 Server 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 Server 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 его. Перед закрывающим тегом </configuration> добавим новую секцию **connectionStrings**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <?xml version="1.0" encoding="utf-8"?>  <configuration>      <!--остальное содержимое-->      <connectionStrings>          <add name="UserDB" connectionString="data source=(localdb)\v11.0;Initial Catalog=userstoredb;Integrated Security=True;"       providerName="System.Data.SqlClient"/>        </connectionStrings>  </configuration> |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class User  {      public int Id { get; set; }      public string Name { get; set; }      public int Age { get; set; }  } |

И также добавим класс контекста данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System;  using System.Collections.Generic;  using System.Data.Entity;    namespace CodeSecond  {      class UserContext : DbContext      {          public UserContext():              base("UserDB")          { }            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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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);      }  } 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

**П**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | namespace AutoCodeSecond  {      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; }            [Required]          [StringLength(50)]          public string Name { get; set; }            public int Age { get; set; }      }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | <connectionStrings>      <add name="UserContext" connectionString="data source=(localdb)\v11.0;initial catalog=userstoredb;          integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"          providerName="System.Data.SqlClient" />  </connectionStrings> |

Для полноценной работы нам осталось добавить класс контекста данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using System;  using System.Collections.Generic;  using System.Data.Entity;    namespace AutoCodeSecond  {      class UserContext : DbContext      {          public UserContext():base("UserContext")          { }          public DbSet<User> Users { get; set; }      }  } |

И теперь мы можем взаимодействовать с базой данных:

|  |  |
| --- | --- |
| 1  2  3  4  5 | using(UserContext db = new UserContext())  {      foreach (User u in db.Users)          Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Age);  } |

## Автоматизация Code First и EF Power Tools

**Последнее обновление: 31.10.2015**

Кроме вышеописанной функциональности 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](http://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d/). 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 ОК. 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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 сопоставление таблиц и их столбцов с классами и их свойствами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | 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:

|  |  |
| --- | --- |
| 1  2  3 | User p = new User { Name = "Имя", Age = 30 };  userstoredbContext.Users.Add(p);  userstoredbContext.SaveChanges(); |

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

**Последнее обновление: 31.10.2015**

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](http://msdn.microsoft.com/ru-RU/data/dd363565.aspx).

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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);  } |

## Основные операции с данными

**Редактирование**

Контекст данных способен отслеживать изменения объектов, поэтому для редактирования объекта достаточно изменить его свойства и после этого вызвать метод SaveChanges():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using (PhoneContext db = new PhoneContext())  {      // получаем первый объект      Phone p1 = db.Phones.FirstOrDefault();        p1.Price = 30000;      db.SaveChanges();   // сохраняем изменения  } |

Но рассмотрим другую ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Phone p1;  using (PhoneContext db = new PhoneContext())  {      p1 = db.Phones.FirstOrDefault();  }    using (PhoneContext db = new PhoneContext())  {      if(p1!=null)      {          p1.Price = 60000;          db.SaveChanges();      }  } |

Так как объект Phone получен в одном контексте, а изменяется для другого контекста, который его не отслеживает. В итоге изменения не сохранятся. Чтобы изменения сохранились, нам явным образом надо установить для его состояния значение **EntityState.Modified**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using (PhoneContext db = new PhoneContext())  {      if(p1!=null)      {          p1.Price = 60000;          db.Entry(p1).State = EntityState.Modified;          db.SaveChanges();      }  } |

**Удаление**

Для удаления объекта применяется метод **Remove()** объекта DbSet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using (PhoneContext db = new PhoneContext())  {      Phone p1 = db.Phones.FirstOrDefault();      if(p1!=null)      {          db.Phones.Remove(p1);          db.SaveChanges();      }  } |

Но как и в случае с обновлением здесь мы можем столкнуться с похожей проблемой, когда объект получаем из базы данных в пределах одного контекста, а пытаемся удалить в другом контексте. И в этом случае нам надо установить вручную у объекта состояние **EntityState.Deleted**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using (PhoneContext db = new PhoneContext())  {      if(p1!=null)      {          db.Entry(p1).State = EntityState.Deleted;          db.SaveChanges();      }  } |

**Метод Attach**

Если объект получен в одном контексте, а сохраняется в другом, то мы можем устанавливать у него вручную состояния EntityState.Updated или EntityState.Deleted. Но есть еще один способ: с помощью метода **Attach** у объекта DbSet мы можем прикрепить объект к текущему контексту данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | Phone p1;  using (PhoneContext db = new PhoneContext())  {      p1 = db.Phones.FirstOrDefault();  }  // редактирование  using (PhoneContext db = new PhoneContext())  {      if(p1!=null)      {          db.Phones.Attach(p1);          p1.Price = 999;          db.SaveChanges();      }  }  // удаление  using (PhoneContext db = new PhoneContext())  {      if(p1!=null)      {          db.Phones.Attach(p1); |

## Операции с данными. Практический пример

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. У всех этих трех полей установим свойство **Modifiers** равным **Protected Internal**, чтобы эти поля были доступны из главной формы.

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93 | using System;  using System.Collections.Generic;  using System.Data;  using System.Data.Entity;  using System.Windows.Forms;    namespace CRUD  {      public partial class Form1 : Form      {          SoccerContext db;          public Form1()          {              InitializeComponent();                db = new SoccerContext();              db.Players.Load();                dataGridView1.DataSource = db.Players.Local.ToBindingList();          }          // 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("Новый объект добавлен");          }          // 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.SaveChanges();                  dataGridView1.Refresh(); // обновляем грид                  MessageBox.Show("Объект обновлен");                }          }          // 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("Объект удален");              }          }      }  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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.

В данном случае контекст данных автоматически отслеживает, что объект был изменен, и при вызове метода db.SaveChanges() будет сформировано SQL-выражение UPDATE для данного объекта, которое обновит объект в базе данных.

**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.

## Навигационные свойства и загрузка данных

Например, у нас определен следующий класс футбольной команды Team:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Team  {      public int Id { get; set; }      public string Name { get; set; } // название команды      public string Coach { get; set; } // тренер        public ICollection<Player> Players { get; set; }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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.

Аналогично в классе Team также имеется навигационное свойство - Players, через которое мы можем получать игроков данной команды.

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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 загрузки и получения связанных данных

В Entity Framework есть три способа загрузки данных:

* **eager loading**("жадная загрузка")
* **explicit loading**("явная загрузка")
* **lazy loading**("ленивая загрузка")

#### Eager Loading

Суть Eager Loading заключается в том, чтобы использовать для подгрузки связанных по внешнему ключу данных метод **Include**. Например, получим всех игроков с их командами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using(SoccerContext db = new SoccerContext())  {      var players = db.Players.Include(p=>p.Team).ToList();      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

Соответственно чтобы подгрузить к командам все данные по игрокам, мы можем написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | using(SoccerContext db = new SoccerContext())  {      var teams = db.Teams.Include(t=>t.Players).ToList();      foreach (var t in teams)      {          Console.WriteLine($"{t.Name}");          foreach(var p in t.Players)              Console.WriteLine($"{p.Name}");      }  } |

#### Explicit Loading

Явная загрузка предусмативает применение метода **Load()** для загрузки данных в контекст. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using(SoccerContext db = new SoccerContext())  {      var p = db.Players.FirstOrDefault();      db.Entry(p).Reference("Team").Load();      Console.WriteLine($"{p.Name} - {p.Team.Name}");          var t = db.Teams.FirstOrDefault();      db.Entry(t).Collection("Players").Load();      Console.WriteLine($"{t.Name}");      foreach(var pl in t.Players)          Console.WriteLine($"{pl.Name}");  } |

Чтобы подгрузить данные, здесь идет обращение к методу db.Entry(), в который передается нужный объект. Для подгрузки связанного объекта, который не представляет коллекцию, используется метод **Reference()**. В этот метод переается навигационное свойство, по которому надо подгрузить данные.

Если связанные объект представляет коллекцию, то применяется метод Collection(), в который также передается навигационное свойство в виде строки.

#### Lazy Loading

Еще один 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | 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>();      }  } |

В этом случае нам не потребуется использовать какие-то дополнительные методы, как Include или Load:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using (SoccerContext db = new SoccerContext())  {      var players = db.Players.ToList();      foreach (var p in players)          Console.WriteLine($"{p.Name} - {p.Team.Name}");        var teams = db.Teams.ToList();      foreach (var t in teams)      {          Console.WriteLine($"{t.Name}");          foreach (var p in t.Players)              Console.WriteLine($"{p.Name}");      }  } |

## Связь один-к-одному

**Последнее обновление: 31.10.2015**

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  //.................................    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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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].[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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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 = "Tom" };      UserProfile profile2 = new UserProfile { Id = user2.Id, Age = 27, Name = "Alice" };      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);  } |

Редактирование:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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();      }  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | 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();      }  } |

## Связь один ко многим

**Последнее обновление: 17.12.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | 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 Team Team { get; set; }  }  public 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>();      }  }  public class SoccerContext : DbContext  {      public SoccerContext() : base("SoccerContext")      {}        public DbSet<Player> Players { get; set; }      public DbSet<Team> Teams { get; set; }  } |

Добавление и вывод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //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(); |

При удалении объектов, связанных отношением "один-ко-многим" нам надо учитывать то, что по умолчанию даже если внешний ключ допускает значение null (как в данном случае свойство TeamId в классе Player), мы не сможем просто так удалить одну модель, если она имеет ссылки на другую модель. Например, удаление команды в данном случае выльется в ошибку, если какой-то объект Player имеет ссылку на эту команду. Что делать в этом случае? В этом случае нам надо установить для внешнего ключа TeamId в таблице игроков ограничение ON DELETE SET NULL. Данное ограничение позволит при удалении связанного объекта устанавливать для внешнего ключа значение null.

Установку ограничения можно сделать вручную, изменим схему базы данных, либо программно. Рассмотрим, как это сделать программно. Для этого перед удалением нам надо выполнить соответствующую команду SQL, чтобы добавить ограничение:

|  |  |
| --- | --- |
| 1 | db.Database.ExecuteSqlCommand("ALTER TABLE dbo.Players ADD CONSTRAINT Players\_Teams FOREIGN KEY (TeamId) REFERENCES dbo.Teams (Id) ON DELETE SET NULL"); |

Здесь добавляется ограничение "Players\_Teams" в таблицу игроков, которая называется, как правило, dbo.Players или просто Players. Это ограничение указывает, что для внешнего ключа TeamId из таблицы Players, который связан со столбцом Id из таблицы dbo.Teams, устанавливается правило "ON DELETE SET NULL", то есть установка null по удалению.

И после этого мы можем выполнить удаление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //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(); |

## Связь один ко многим. Практический пример

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | using System.Data.Entity;    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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106 | public partial class Form1 : Form  {      SoccerContext db;      public Form1()      {          InitializeComponent();            db = new SoccerContext();          db.Players.Load();          dataGridView1.DataSource = db.Players.Local.ToBindingList();      }      // добавление      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");      }        // 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;                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");          }      }      // 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("Объект удален");          }      }      // открываем форму с командами      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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93 | public partial class AllTeams : Form  {      SoccerContext db;      public AllTeams()      {          InitializeComponent();            db = new SoccerContext();          db.Teams.Load();          dataGridView1.DataSource = db.Teams.Local.ToBindingList();      }      // добавление      private void button1\_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;            db.Teams.Add(team);          db.SaveChanges();          MessageBox.Show("Новый объект добавлен");      }      // просмотр списка игроков команды      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");          }      }      // 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;                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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | 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 string Position { 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | 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:

|  |  |
| --- | --- |
| 1  2  3  4 | // 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:

|  |  |
| --- | --- |
| 1  2 | Player pl\_delete = db.Players.First(p=>p.Name=="Месси");  db.Players.Remove(pl\_delete); |

## Инициализация базы данных

**Последнее обновление: 31.10.2015**

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, которые 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.
* **DropCreateDatabaseIfModelChanges**: 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | 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();      }  }  public class Phone  {      public int Id { get; set; }      public string Name { get; set; }      public int Price { get; set; }  }    class MobileContext : DbContext  {      static MobileContext()      {          Database.SetInitializer<MobileContext>(new MyContextInitializer());      }        public MobileContext() : base("DefaultConnection")      { }      public DbSet<Phone> Phones { get; set; }  } |

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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public ActionResult Edit(int id)  {      Person p = db.People.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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using System.ComponentModel.DataAnnotations;  //..................................  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 Frameworkом 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:

|  |  |
| --- | --- |
| 1 | @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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using(UserContext db = new UserContext())  {      using(var transaction = db.Database.BeginTransaction())      {          try          {              Person p1 = db.People.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.People.Add(p2);              db.SaveChanges();              transaction.Commit();          }          catch(Exception ex)          {              transaction.Rollback();          }      }        foreach(Person p in db.People.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**DbContextTransaction** реализует 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()

## Миграции

**Последнее обновление: 27.09.2016**

Миграции позоляют вносить изменения в базу данных при изменениях моделей и контекста данных. Так, пусть у нас есть следующая модель Phone и контекст данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class PhoneContext : DbContext  {      public DbSet<Phone> Phones { get; set; }      public PhoneContext() : base("DefaultConnection")      { }  }    public class Phone  {      public int Id { get; set; }      public string Name { get; set; }      public int Price { get; set; }  } |

Мы можем использовать этот контекст данных для работы с БД, добавлять и удалять данные. Но в какой-то момент, возможно, нам захочется что-то изменить, например, добавить в модель Phone новое свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Phone  {      public int Id { get; set; }      public string Name { get; set; }      public string Company { get; set; }      public int Price { get; set; }  } |

И если мы попытаемся обратиться к базе данных после изменения модели, то мы получим ошибку:

Для решения этой проблемы нам надо изменить базу данных таким образом, чтобы она вновь соответствовала моделям и контексту данных. И для этого мы можем применить миграции.

Для этого в Visual Studio перейдем к окну **Package Manager Console**, которое можно найти внизу VS. Если такого окна нет, то его можно открыть, перейдя к меню **View**->**Other Window**->**Package Manager Console**

Для добавления функционала миграций введем в это окно следующую команду:

enable-migrations

После ввода команды нажмем на Enter. И в результате выполнения данной команды в проект будет добавлена папка *Migrations*, в которой будут два файла: *Configration.cs* (содержит базовую конфигурацию миграций) и файл начальной миграции, название которого может отличаться. Файл начальной миграции устанавливает, как база данных определяется на данный момент. То есть в моем случае он будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | namespace CodeFirstApp.Migrations  {      using System;      using System.Data.Entity.Migrations;        public partial class InitialCreate : DbMigration      {          public override void Up()          {              CreateTable(                  "dbo.Phones",                  c => new                      {                          Id = c.Int(nullable: false, identity: true),                          Name = c.String(),                          Price = c.Int(nullable: false),                      })                  .PrimaryKey(t => t.Id);            }            public override void Down()          {              DropTable("dbo.Phones");          }      }  } |

Далее выполним в Package Manager Console следующую команду:

add-migration "AddCompanyMigration"

И после выполнения этой команды в папку Migrations будет добавлена новая миграция:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System;  using System.Data.Entity.Migrations;    public partial class AddCompanyMigration : DbMigration  {      public override void Up()      {          AddColumn("dbo.Phones", "Company", c => c.String());      }        public override void Down()      {          DropColumn("dbo.Phones", "Company");      }  } |

В миграции определяются два метода: Up() и Down(). В методе Up с помощью вызова метода AddColumn добавляется новый столбец Company в уже имеющуюся таблицу dbo.Phones. Метод Down удаляет столбец на случай, если они существуют. Фактически эти методы равнозначные выражению ALTER в языке SQL, которое меняет структуру базы данных и ее таблиц.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

update-database

Эта команда обновит базу данных, добавив в нее новый столбец. Причем данные, которые уже были в таблицы, сохранятся.

Если база данных уже используется в производстве, развернута на сервере, где бы не можем произвести миграции, то мы можем сгенерировать по миграции скрипт. Для этого надо ввести следующую команду:

update-database -script

В итоге в моем случае будет сгенерирован следующий скрипт SQL:

|  |  |
| --- | --- |
| 1  2  3 | ALTER TABLE [dbo].[Phones] ADD [Company] [nvarchar](max)  INSERT [dbo].[\_\_MigrationHistory]([MigrationId], [ContextKey], [Model], [ProductVersion])  VALUES (N'201609271310287\_AddCompanyMigration', N'CodeFirstApp.PhoneContext',  0x1F8B0800000000000400CD57DB6EDB38107D2FB0FF207CC35C38C0E0000 , N'6.1.3-40302') |

Затем данный скрипт можно выполнить для используемой базы данных.

### Методы миграций

Основу миграции составляют ряд методов, которые позволяют удалять, добавлять столбцы и таблицы, изменять настройки столбцов и так далее. Основные методы:

* **CreateTable**: добавляет таблицу
* **DropTable**: удаляет таблицу
* **AddColumn**: добавляет столбец
* **DropColumn**: удаляет столбец
* **AlterColumn**: изменяет настройки столбца
* **AddForeignKey**: добавляет внешний ключ
* **DropForeignKey**: удаляет внешний ключ
* **AddPrimaryKey**: добавляет первичный ключ
* **DropPrimaryKey**: удаляет первичный ключ
* **CreateIndex**: добавляет индекс
* **DropIndex**: удаляет индекс
* **CreateStoredProcedure**: создает хранимую процедуру
* **DropStoredProcedure**: удаляет хранимую процедуру
* **AlterStoredProcedure**: изменяет хранимую процедуру

Далее от этих методов можно строить цепочки дополнительных методов. Например, создание таблицы:

|  |  |
| --- | --- |
|  | CreateTable(      "dbo.Companies",   // название таблицы      c => new    // столбцы      {          Id = c.Int(nullable: false, identity: true),          Name = c.String(),      })      .PrimaryKey(t => t.Id); // первичный ключ |

# LINQ to Entities

## Введение в LINQ to Entities

**Последнее обновление: 31.10.2015**

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 Serverа.

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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; }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | 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: 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1  2  3  4 | using(PhoneContext db = new PhoneContext())  {      var phones = db.Phones.Where(p=> p.CompanyId == 1);  } |

Оба запроса в итоге транслируются в одной выражение sql:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1 | 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. После 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

**Последнее обновление: 31.10.2015**

Для выборки применяется метод Where. Выберем из бд все модели, производитель которых - "Samsung":

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1 | Phone myphone = db.Phones.Find(3); // выберем элемент с id=3 |

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.

|  |  |
| --- | --- |
| 1  2  3 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Model  {      public string Name { get; set; }      public string Company { get; set; }      public int Price { get; set; }  } |

И спроецируем выборку на этот тип:

|  |  |
| --- | --- |
|  | 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

**Последнее обновление: 31.10.2015**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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()**:

|  |  |
| --- | --- |
| 1 | var phones = db.Phones.OrderByDescending(p=>p.Name); |

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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1  2  3 | 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 }; |

### Соединение трех таблиц

Допустим, у нас есть три таблицы, которые связаны между собой и которые описываются следующими моделями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class Country  {      public int Id { get; set; }      public string Name { get; set; }  }  public class Company  {      public int Id { get; set; }      public string Name { get; set; }      public int CountryId { get; set; }      public Country Country { get; set; }  }  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; }  } |

Объединим три таблицы в один набор:

|  |  |
| --- | --- |
|  | var result = from phone in db.Phones               join company in db.Companies on phone.CompanyId equals company.Id               join country in db.Countries on company.CountryId equals country.Id               select new               {                  Name = phone.Name,                  Company = company.Name,                  Price = phone.Price,                  Country = country.Name               }; |

## Группировка

**Последнее обновление: 31.10.2015**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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():

|  |  |
| --- | --- |
| 1 | var groups = db.Phones.GroupBy(p=>p.Company.Name); |

Кроме свойства Key у группы есть метод Count(), который возвращает количество элементов в данной группе. Например, сформируем новый элемент, который будет содержать ключ группы и количество ее элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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

**Последнее обновление: 31.10.2015**

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()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2 | 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()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2 | IEnumerable<Phone> phoneIEnum = db.Phones;  phoneIEnum=phoneIEnum.Where(p => p.Id > id); |

Здесь запрос будет иметь следующий вид:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

|  |  |
| --- | --- |
| 1  2 | IQueryable<Phone> phoneIQuer = db.Phones;  phoneIQuer=phoneIQuer.Where(p => p.Id > id); |

Здесь запрос будет иметь следующий вид:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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:

|  |  |
| --- | --- |
| 1  2  3 | db.Phones.Add(new Phone { Name = "Samsung Galaxy Note" });  db.Phones.Add(new Phone { Name = "iPhone 6"});  db.SaveChanges(); |

При обычном выполнении:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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.

# SQL в EntityFramework

## Работа с SQL

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using(PhoneContext db = new PhoneContext())  {      var comps = db.Database.SqlQuery<Company>("SELECT \* FROM Companies");      foreach (var company in comps)          Console.WriteLine(company.Name);  } |

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":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 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

**Последнее обновление: 31.10.2015**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | 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; }  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | 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#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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);  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class DiscountPhone  {      public string Name { get; set; }      public decimal Price { get; set; }  } |

И теперь результат функции мы можем получить следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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);  } |

# Fluent API и аннотации

**Последнее обновление: 31.10.2015**

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**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public 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()**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | protected override void OnModelCreating(DbModelBuilder modelBuilder)  {      modelBuilder.Entity<Phone>().ToTable("Mobiles");      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()**:

|  |  |
| --- | --- |
| 1 | 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()**:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | modelBuilder.Entity<Phone>().HasKey(p => new { p.Ident, p.Name }); |

### Сопоставление свойств

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

|  |  |
| --- | --- |
| 1 | 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()**:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | 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()**:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | modelBuilder.Entity<Phone>().Property(p => p.Name).HasMaxLength(50); |

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

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2 | // 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()**:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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

**Последнее обновление: 31.10.2015**

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | protected override void OnModelCreating(DbModelBuilder modelBuilder)  {      modelBuilder.Entity<Phone>()          .HasRequired(c => c.Company)          .WithOptional(c => c.BestSeller);        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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | modelBuilder.Entity<Phone>()      .HasRequired(c => c.Company)      .WithRequiredPrincipal(c => c.BestSeller);  //или так  //modelBuilder.Entity<Company>()  //    .HasRequired(c => c.BestSeller)  //    .WithRequiredPrincipal(c => c.Company); |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | 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:

|  |  |
| --- | --- |
| 1  2  3 | modelBuilder.Entity<Phone>()      .HasMany(p => p.Companies)      .WithMany(c => c.Phones); |

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | modelBuilder.Entity<Phone>()      .HasMany(p => p.Companies)      .WithMany(c => c.Phones)      .Map(m =>      {          m.ToTable("MobileCompanies");          m.MapLeftKey("MobileId");          m.MapRightKey("CompanyId");      }); ; |

### 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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:

|  |  |
| --- | --- |
| 1  2  3 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public 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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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)**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

|  |  |
| --- | --- |
| 1  2  3  4 | using System.Data.Entity.ModelConfiguration.Conventions;  ...........................................................  modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();  modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>(); |