

C#. Entity Framework

Оглавление

- 1) [Введение в Entity Framework ;](#)
- 2) [Пример создания БД с помощью «Code first»;](#)
- 3) [Основные операции с данными;](#)
- 4) [Связи между таблицами;](#)
- 5) [Способы загрузки и получения связанных данных;](#)
- 6) [Пример приложения с визуальными формами для работы с данными;](#)
- 7) [Пример приложения для работы с таблицами со связью один ко многим;](#)
- 8) [Инициализация БД;](#)
- 9) [LINQ to Entity Framework;](#)
- 10) [SQL to Entity Framework;](#)
- 11) [Список источников.](#)

Определение Entity Framework

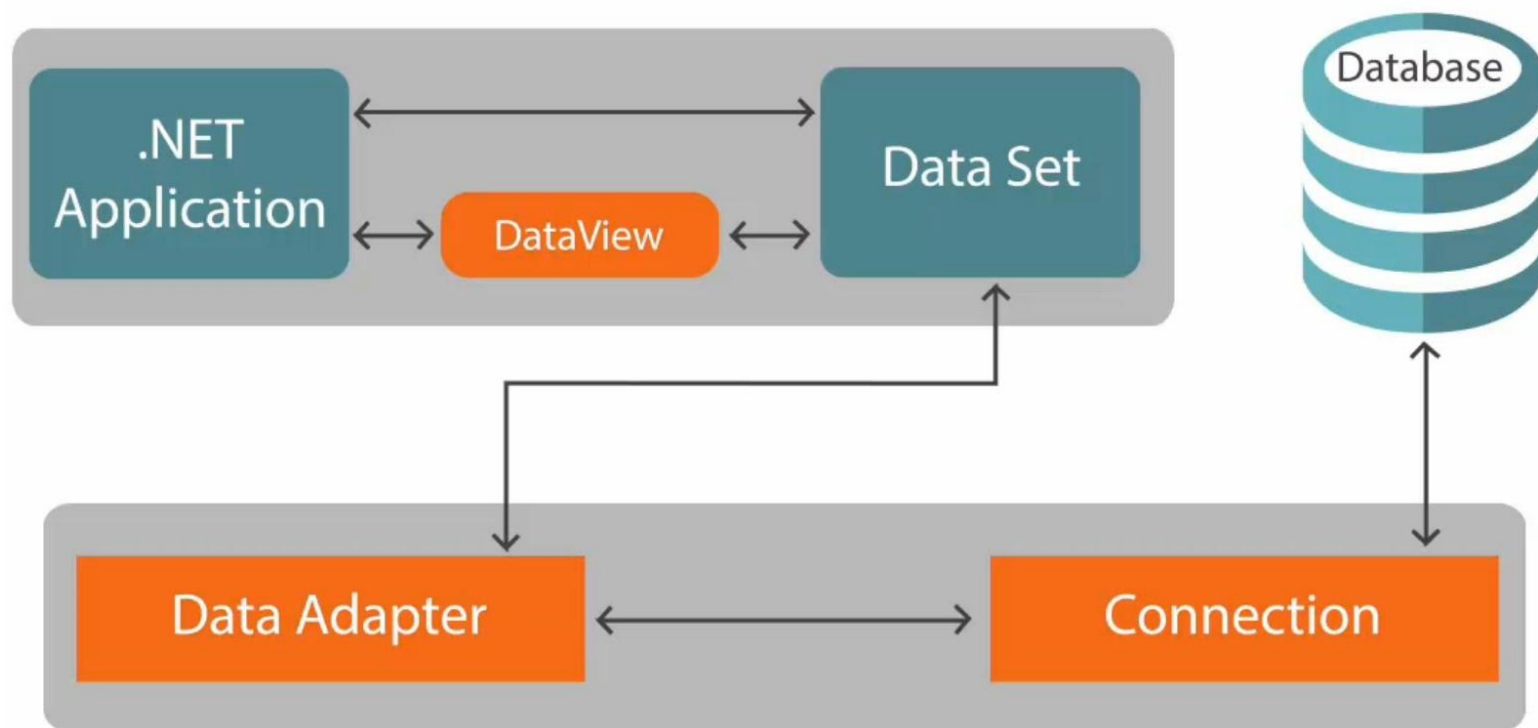
Entity Framework - объектно-ориентированная технология на базе фреймворка .NET для работы с базами данными или по другому говоря это ORM.

ORM

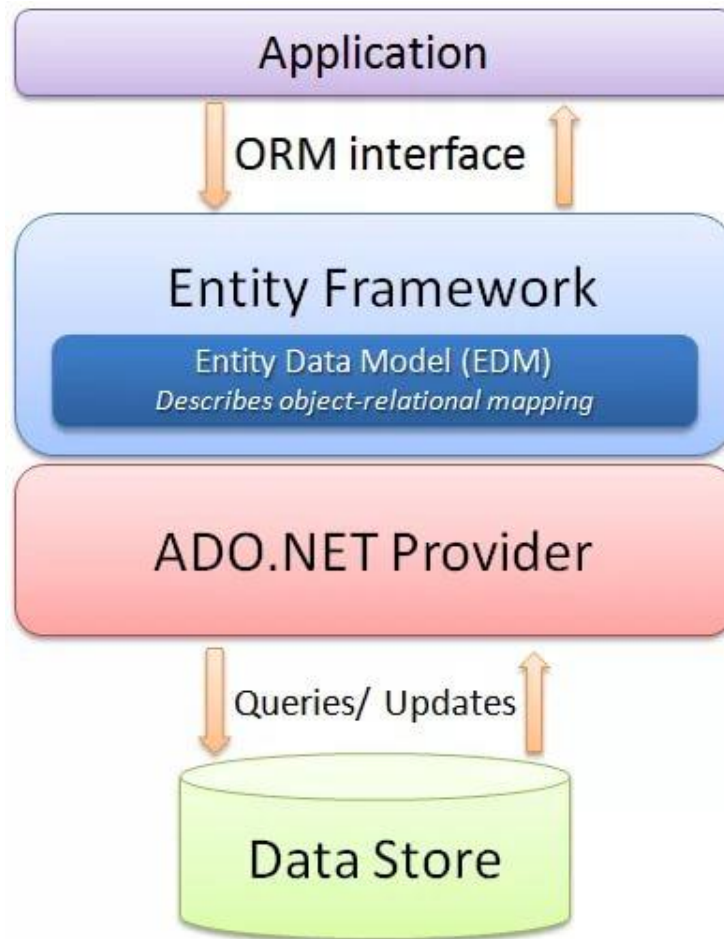
ORM - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Текущая модель работы с БД

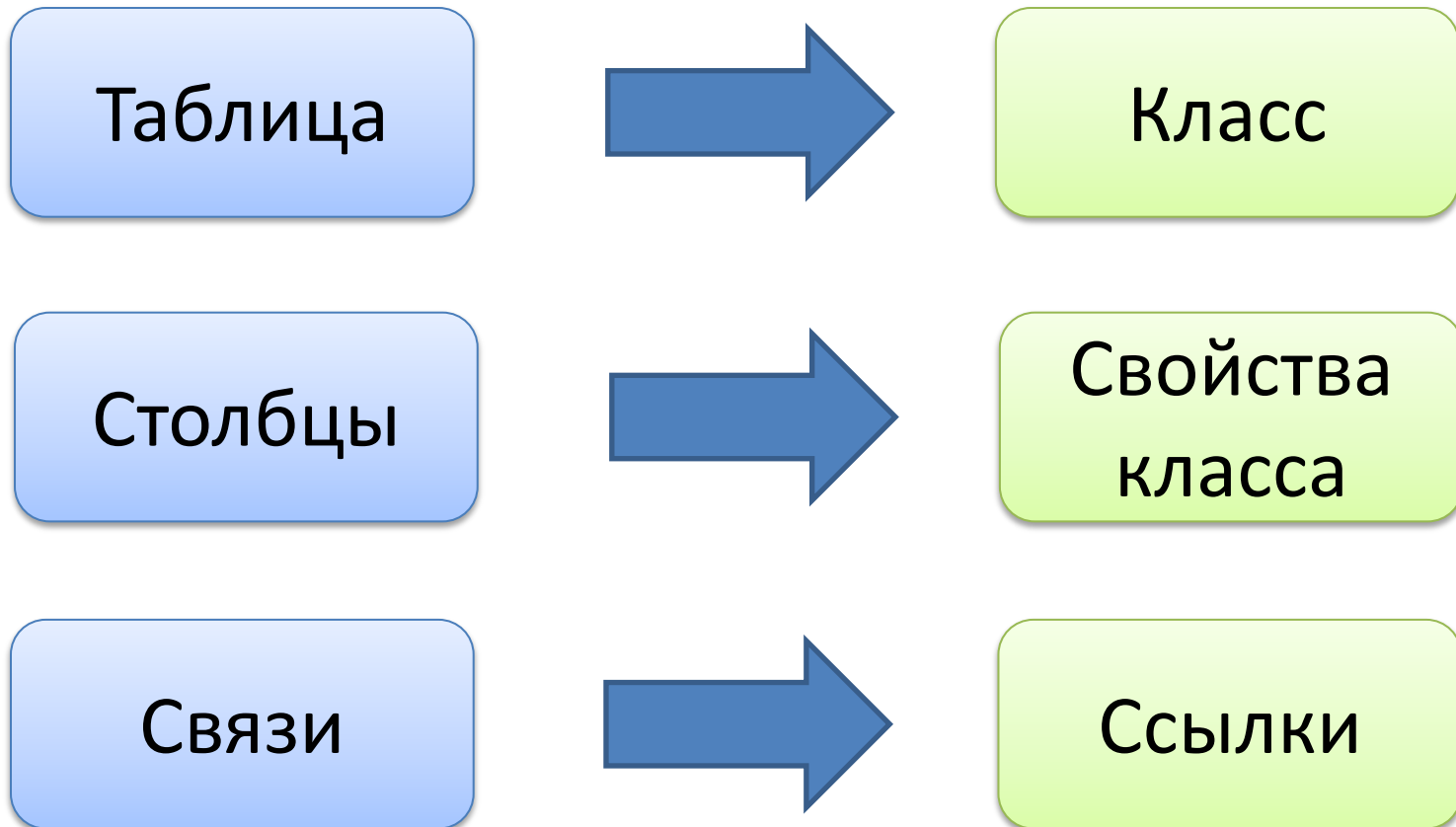
ADO.NET Disconnected Model Architecture



Предлагаемая модель работы с БД



Что поменялось?



Способы взаимодействия с БД

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

- Database;
- Model first;
- Code first.

Database

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.

Соглашения по наименованию в Code First

Типы SQL Servera и C# сопоставляются следующим образом:

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

NULL or NOT NULL вот чём вопрос!

- Все первичные ключи по умолчанию имеют определение NOT NULL.
- Столбцы, сопоставляемые со свойствами ссылочных типов (string, array), в базе данных имеют определение NULL, а все значимые типы (DateTime, bool, char, decimal, int, double, float) - NOT NULL.
- Если свойство имеет тип Nullable<T>, то оно сопоставляется со столбцом с определением NULL.


Ключи от квартиры, где деньги лежат.

- 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

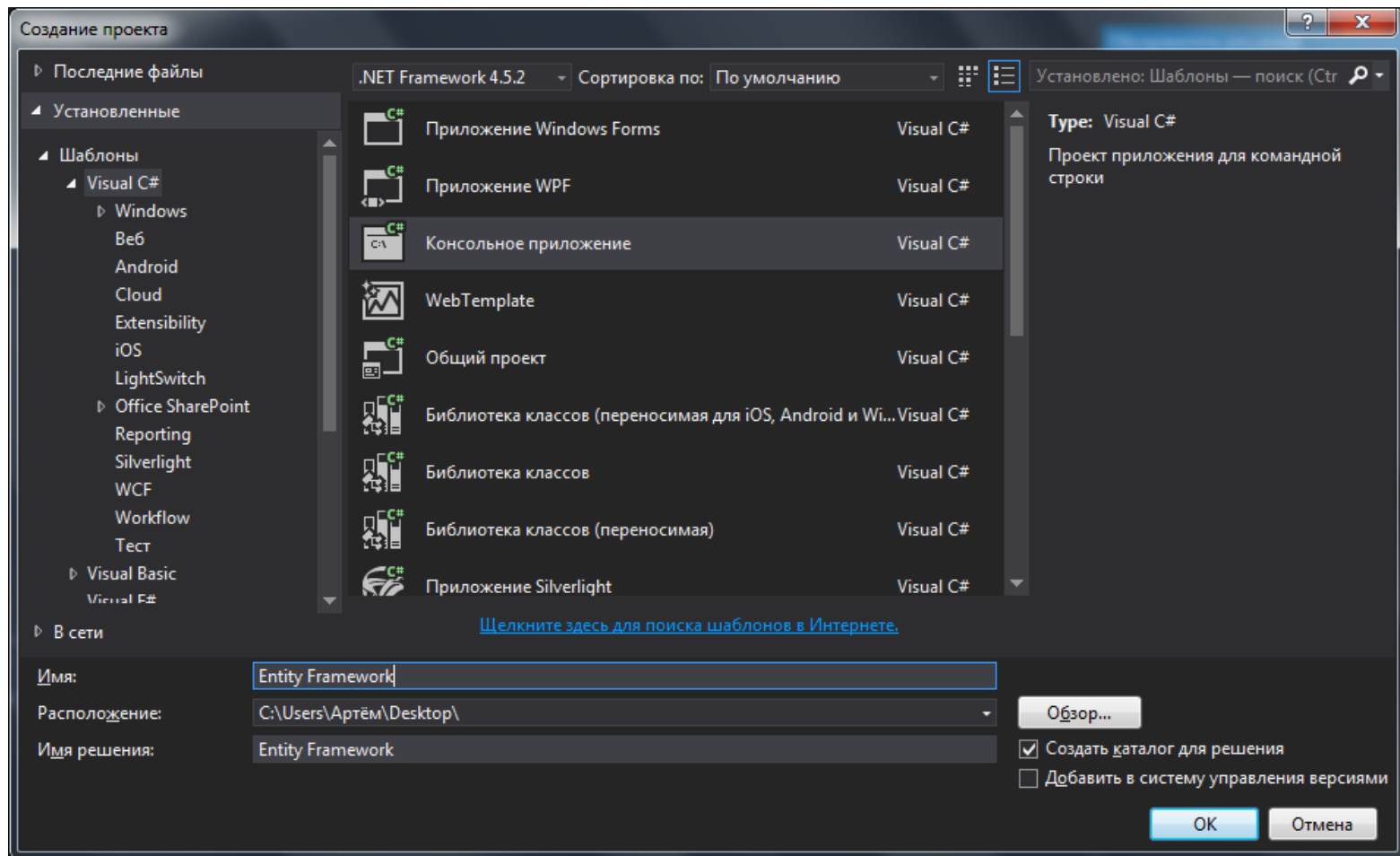
- С помощью специального класса `PluralizationService` Entity Framework проводит сопоставление между именами классов моделей и именами таблиц. При этом таблицы получают по умолчанию в качестве названия множественное число в соответствии с правилами английского языка, например, класс `User` - таблица `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.

Пример создания БД с помощью «Code first»



Перейти к
следующей теме

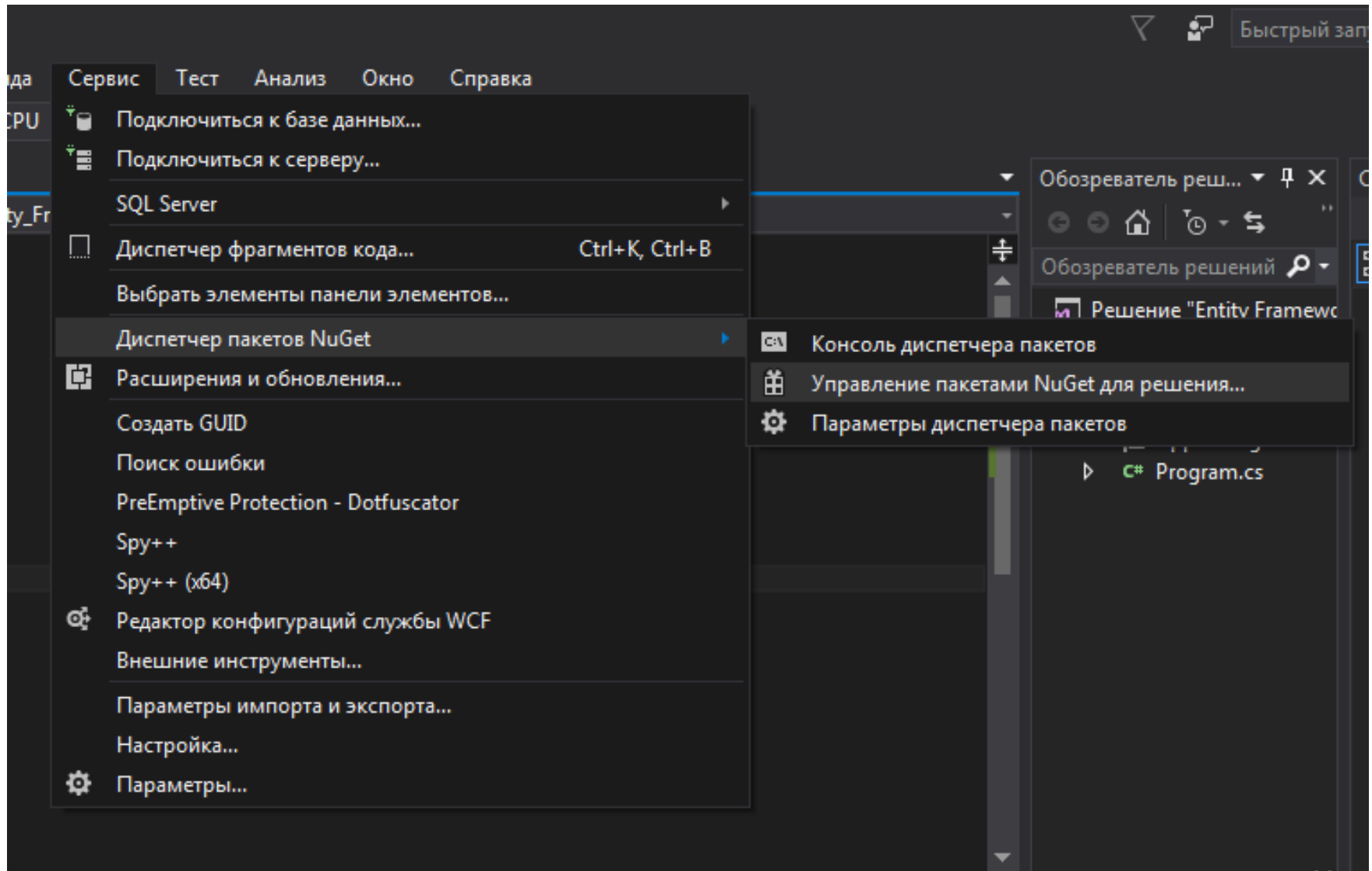
Создадим пустое консольное приложение.



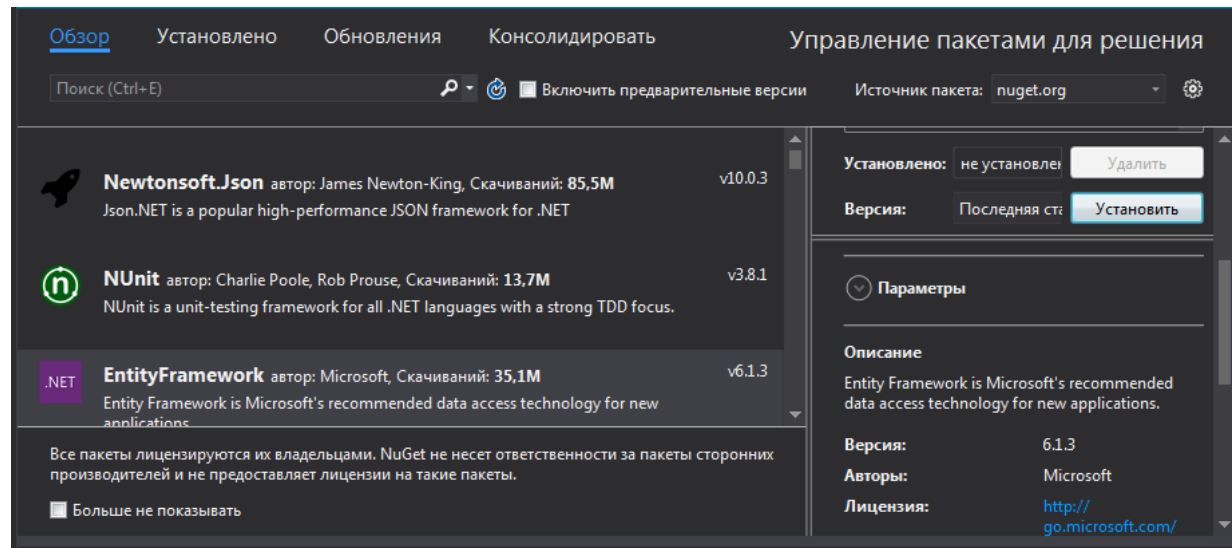
Теперь первым делом добавим новый класс, который будет описывать данные. Пусть наше приложение будет посвящено работе с пользователями. Поэтому добавим в проект новый класс User:

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

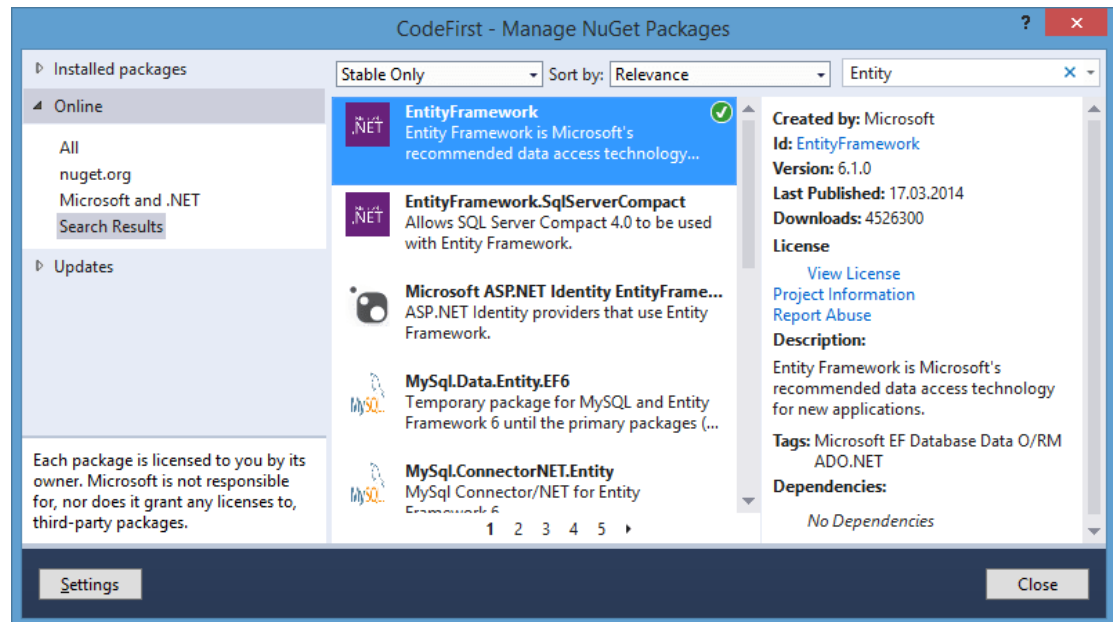
Добавление библиотеки Entity Framework



Visual Studio 2015



Visual Studio 2013



После установки пакета добавим в проект новый класс
DbContext:

```
class DbContext : DbContext
{
    public DbContext()
        : base("DbConnection")
    { }

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

И в using библиотеку:

```
using System.Data.Entity;
```

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

- **DbContext**: определяет контекст данных, используемый для взаимодействия с базой данных.
- **DbModelBuilder**: сопоставляет классы на языке C# с сущностями в базе данных.
- **DbSet/DbSet<TEntity>**: представляет набор сущностей, хранящихся в базе данных

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

В конструкторе этого класса вызывается конструктор базового класса, в который передается строка "DbConnection" - это имя будущей строки подключения к базе данных.

И теперь нам надо установить подключение к базе данных. Для установки подключения обычно используется файл конфигурации приложения. В проектах для десктопных приложений файл конфигурации называется *App.config*.

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

```
<connectionStrings>
    <add name="DbConnection"
        connectionString="Data Source=MSSQL-2K8;Initial
Catalog=(Имя базы данных);Persist Security Info=True;User
ID=(свой логин);Password=(свой пароль)"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

Именим main.

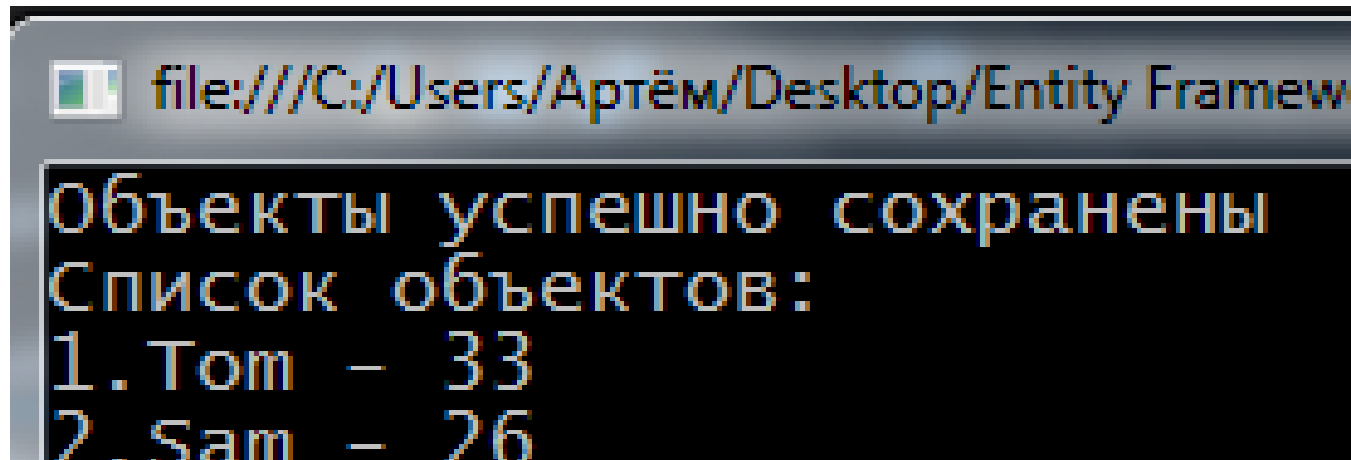
```
static void Main(string[] args)
{
    using (UserContext db = new UserContext())
    {
        // создаем два объекта User
        User user1 = new User { Name = "Tom", Age = 33 };
        User user2 = new User { Name = "Sam", Age = 26 };

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

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

Готово!

Приблизительный вывод:



```
file:///C:/Users/Артём/Desktop/Entity Framework
Объекты успешно сохранены
Список объектов:
1. Tom - 33
2. Sam - 26
```

Готовый пример можно посмотреть в примерах:

«1_Пример подключения Entity Framework и добавления строк»

Основные операции с данными

Основные операции с данными

Большинство операций с данными представляют собой CRUD-операции

(ох только не переводите это слово с английского)

(Create, Read, Update, Delete), то есть получение данных, создание, обновление и удаление. Entity Framework позволяет легко производить данные операции.

Пример

Для примеров с операциями возьмем простенькую модель Phone:

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

И следующий класс контекста данных:

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

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

Добавление

Для добавления применяется метод Add() у объекта DbSet:

```
using (PhoneContext db = new PhoneContext())
{
    Phone p1 = new Phone { Name = "Samsung Galaxy S7", Price = 20000 };
    Phone p2 = new Phone { Name = "iPhone 7", Price = 28000 };

    // добавление
    db.Phones.Add(p1);
    db.Phones.Add(p2);
    db.SaveChanges();    // сохранение изменений

    var phones = db.Phones.ToList();
    foreach (var p in phones)
        Console.WriteLine("{0} - {1} - {2}", p.Id, p.Name, p.Price);
}
```

Редактирование

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

```
using (PhoneContext db = new PhoneContext())
{
    // получаем первый объект
    Phone p1 = db.Phones.FirstOrDefault();

    p1.Price = 30000;
    db.SaveChanges();    // сохраняем изменения
}
```


Но рассмотрим другую ситуацию:

```
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 сохраняются.

Чтобы изменения сохранились, нам явным образом надо установить для его состояния значение EntityState.Modified:

```
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:

```
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:

```
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:

```
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);
        db.Remove(p1);
        db.SaveChanges();
    }
}
```

Связи между таблицами

Связь один ко многим

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

```
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:

```
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 p11 = new Player { Name = "Роналду", Age = 31, Position = "Нападающий", Team = t2 };
    Player p12 = new Player { Name = "Месси", Age = 28, Position = "Нападающий", Team = t1 };
    Player p13 = new Player { Name = "Хави", Age = 34, Position = "Полузащитник", Team = t1 };
    db.Players.AddRange(new List<Player>{p11, p12,p13});
    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();
    }
}
```

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:

```
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:

```
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 предусматривает применение метода Load() для загрузки данных в контекст. Например:

```
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 передается навигационное свойство, по которому надо подгрузить данные.
- Если связанный 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:


```
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:

```
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}");
    }
}
```

Пример создания приложения с визуальными формами для работы с данными

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

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

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

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

```
using System.Data.Entity;
```

```
class SoccerContext : DbContext
```

```
{
```

```
    public SoccerContext()
```

```
        : base("DefaultConnection")
```

```
    { }
```

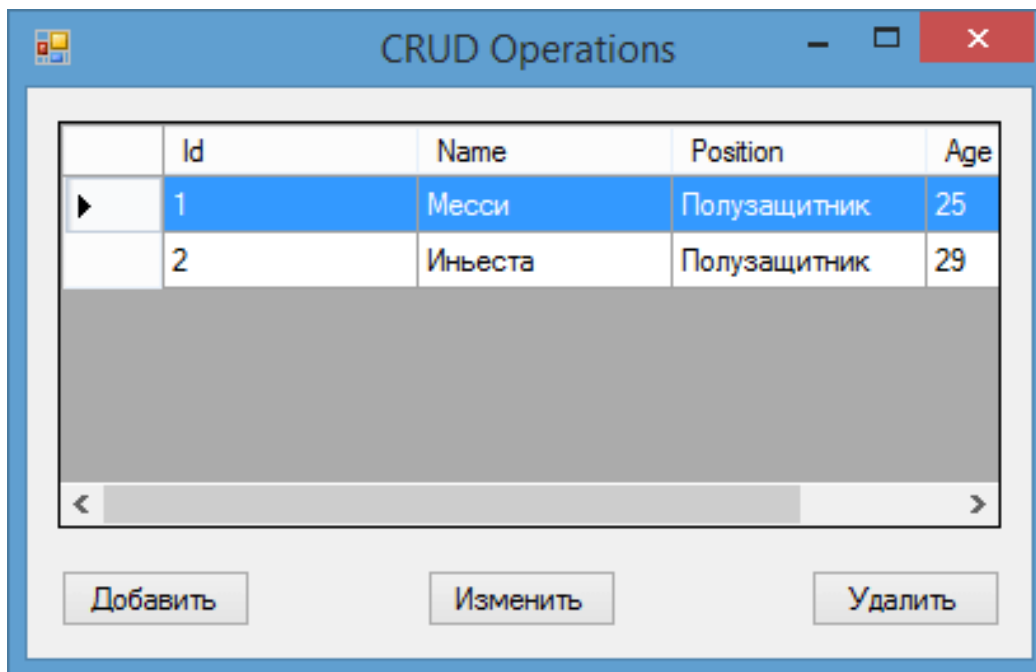
```
    public DbSet<Player> Players { get; set; }
```

```
}
```

Файл конфигурации *app.config* после секции *configSections* добавим узел *connectionStrings*, в котором определим строку подключения *DefaultConnection*:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
    http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
    EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="SoccerDb"
      connectionString="Data Source=MSSQL-2K8;Initial Catalog=LINQ;Persist Security Info=True;User
      ID=;Password="
      providerName="System.Data.SqlClient" />
  </connectionStrings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
    EntityFramework">
      <parameters>
        <parameter value="mssqllocaldb" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient"
      type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

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:

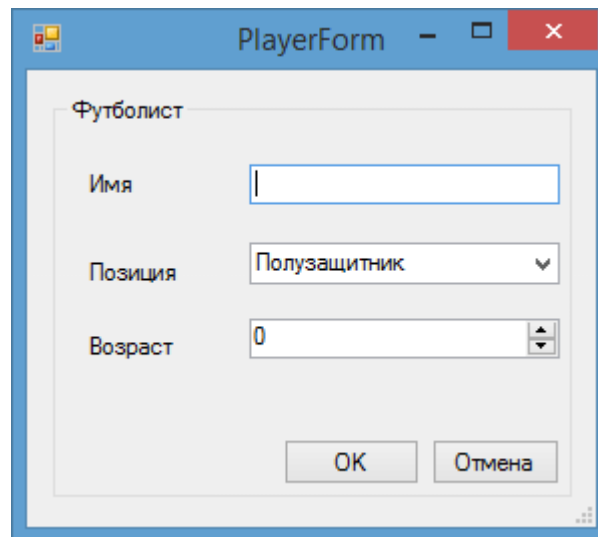


The screenshot shows a Windows application window titled "CRUD Operations". Inside the window, there is a DataGridView control displaying a table with the following data:

	Id	Name	Position	Age
▶	1	Месси	Полузащитник	25
	2	Иньеста	Полузащитник	29

Below the table, there is a horizontal scrollbar. At the bottom of the window, there are three buttons: "Добавить" (Add), "Изменить" (Edit), and "Удалить" (Delete).

- 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:



The image shows a Windows-style dialog box titled "PlayerForm". Inside the dialog, there is a group box labeled "Футболист". Within this group box, there are three input controls: a text box for "Имя" (Name), a dropdown menu for "Позиция" (Position) currently showing "Полузащитник" (Midfielder), and a spin box for "Возраст" (Age) currently showing "0". At the bottom of the dialog, there are two buttons: "ОК" and "Отмена" (Cancel).

- 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:

```
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");
    }
}
```

Комментарии к примеру

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()`)

При добавлении объекта использует вторая форма:

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

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

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

Редактирование

Редактирование имеет похожую логику. Только вначале мы передаем значения свойств объекта во вторую форму, а после получаем с нее же измененные значения для свойств объекта.

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`.

Конец примера

Готовый пример можно посмотреть в примерах:

«2_Entity_Framework_Form. Добавление, изменение, удаление записей в таблице»

«Лень трудолюбия: не ленись отдохнуть!»

Алишер Файз

Пример создания приложения с
визуальными формами для работы с
данными двух таблиц со связью
ОДИН КО МНОГИМ

- Воспользуемся теоретическим материалом из прошлой темы и создадим новое приложение, в котором будет реализована связь один-ко-многим. Для реализации подобной связи будем использовать lazy loading.
- Создадим новый проект Windows Forms. Первым делом подключим через NuGet Entity Framework и добавим все наши модели. Итак, добавим следующие классы:

```
using System.Data.Entity;

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

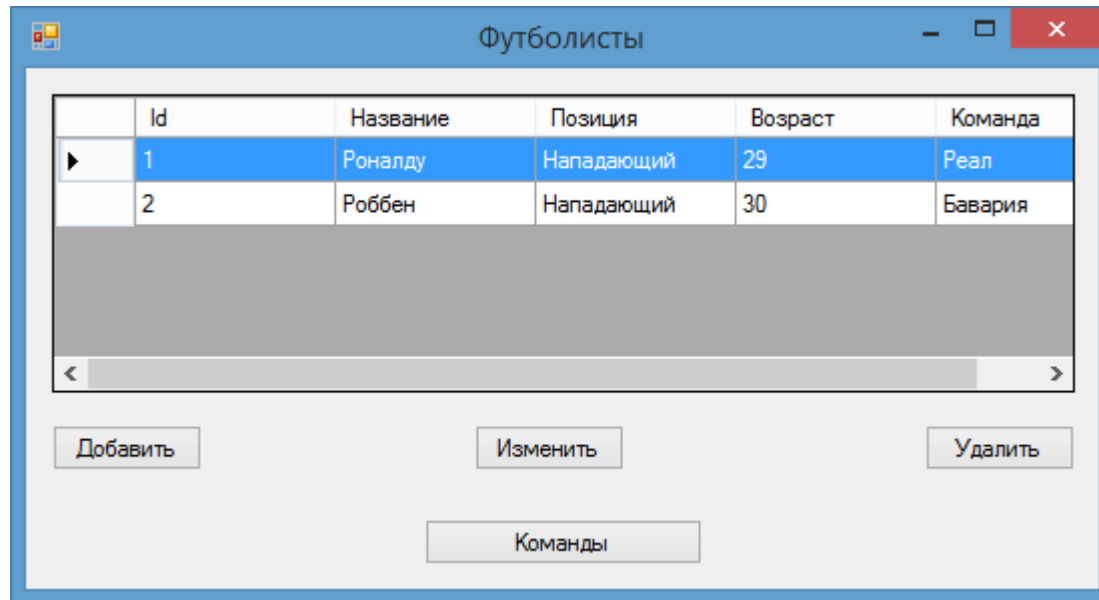
    public virtual ICollection<Player> Players { get;
set; }

    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:



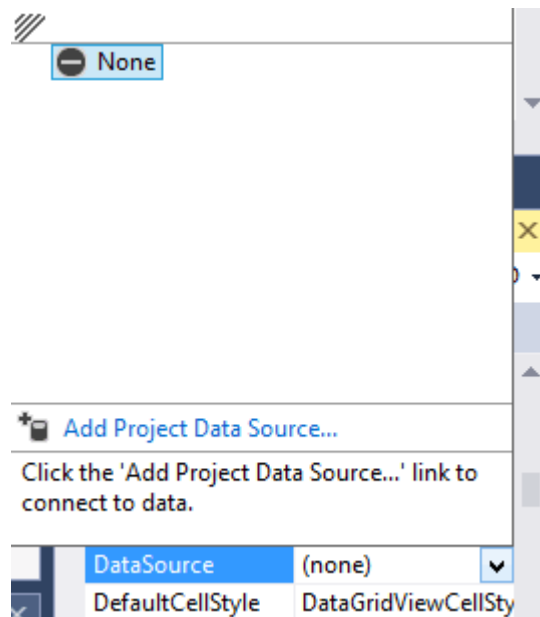
	Id	Название	Позиция	Возраст	Команда
▶	1	Роналду	Нападающий	29	Реал
	2	Роббен	Нападающий	30	Бавария

< ————— >

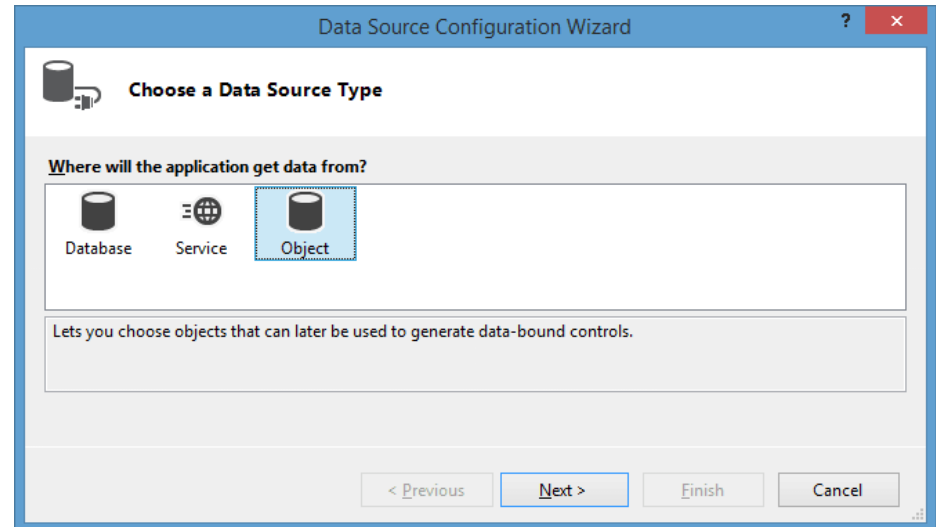
Добавить Изменить Удалить

Команды

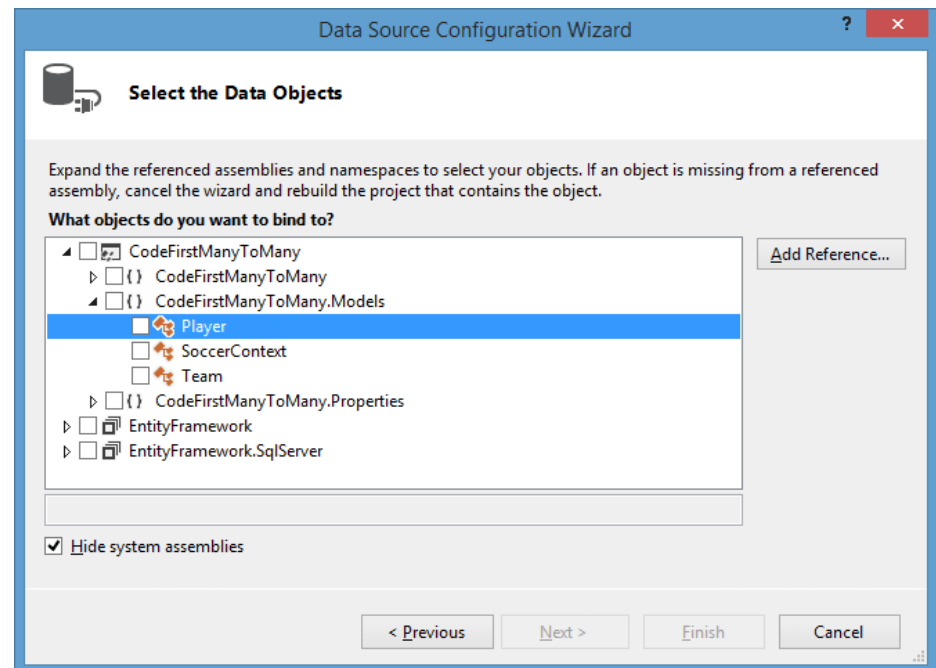
- Здесь у нас элемент `DatatGridView` для отображения данных, а также три кнопки для добавления/редактирования/удаления и одна кнопка для вызова окна с футбольными командами.
- Итак, в предыдущих темах мы уже рассматривали привязку данных к `DataGridView`. При привязке для каждого свойства создается 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:



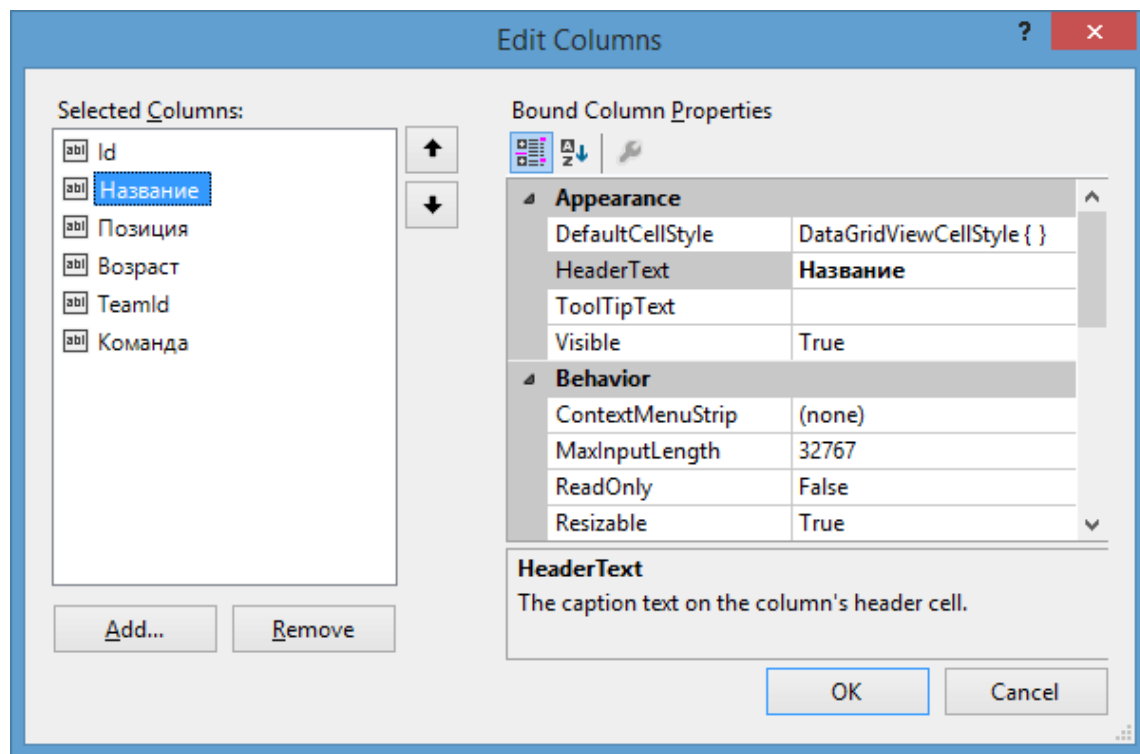
Нажмем на ссылку Add Project Data Source.... После этого нам откроется окно мастера настройки источника данных, в котором нам надо выбрать Object:



И затем на следующем шаге нам отобразится структура 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:

	Id	Название	Тренер
▶	2	Бавария	Гвардиола
	3	Реал	Анчелотти

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:

Футболист

Имя

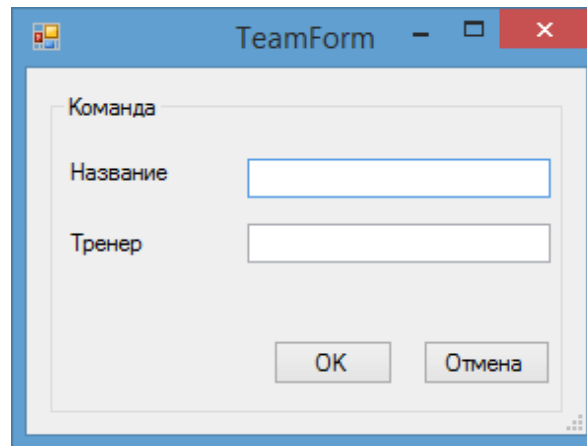
Позиция

Возраст

Команда

OK Отмена

- Здесь текстовое поле для имени игрока, элемент NumericUpDown для указания возраста, и два элемента ComboBox.
- Для кнопки 'ОК' у свойства DialogResult установим значение OK, а у кнопки '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:

```
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();
}
```


Код формы команд:

```
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");
    }
}
```

Конец примера

Готовый пример можно посмотреть в примерах:

«3_Entity_Framework_Form. Создание таблиц со связью один ко многим»

Инициализация базы данных

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

Инициализация происходит при первом обращении к контексту данных.

Для инициализации мы можем использовать один из классов инициализаторов, которые имеются в библиотеке .NET:

CreateDatabaseIfNotExists: инициализатор, используемый по умолчанию. Он не удаляет автоматически 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:

```
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 предполагает вызов его в статическом конструкторе класса контекста.

LINQ в Entity Framework

Определение

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс `IEnumerable` (например, стандартные коллекции, массивы), набор данных `DataSet`, документ XML.

Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Существует несколько разновидностей LINQ:

- **LINQ to Objects:** применяется для работы с массивами и коллекциями
- **LINQ to Entities:** используется при обращении к базам данных через технологию Entity Framework
- **LINQ to Sql:** технология доступа к данным в MS SQL Server
- **LINQ to XML:** применяется при работе с файлами XML
- **LINQ to DataSet:** применяется при работе с объектом DataSet
- **Parallel LINQ (PLINQ):** используется для выполнения параллельной запросов

Простейшее определение запроса LINQ выглядит следующим образом:

```
from переменная in набор_объектов  
select переменная;
```

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.Linq.

Пример кода

```
string[] teams = { "Бавария", "Боруссия", "Реал Мадрид",  
"Манчестер Сити", "ПСЖ", "Барселона" };  
  
var selectedTeams = from t in teams // определяем каждый объект  
из teams как t  
                    where t.ToUpper().StartsWith("Б")  
//фильтрация по критерию  
                    orderby t // упорядочиваем по возрастанию  
                    select t; // выбираем объект  
  
foreach (string s in selectedTeams)  
    Console.WriteLine(s);
```

- Итак, что делает этот запрос LINQ? Выражение `from t in teams` проходит по всем элементам массива `teams` и определяет каждый элемент как `t`. Используя переменную `t` мы можем проводить над ней разные операции.
- Несмотря на то, что мы не указываем тип переменной `t`, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор `teams` состоит из объектов `string`, поэтому переменная `t` будет рассматриваться в качестве строки.
- Далее с помощью оператора `where` проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.
- Оператор `orderby` упорядочивает по возрастанию, то есть сортирует выбранные объекты.
- Оператор `select` передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

Методы расширения LINQ

Кроме стандартного синтаксиса `from .. in .. select` для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса `IEnumerable`. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа `where` или `orderby`.


```
string[] teams = { "Бавария", "Боруссия", "Реал Мадрид",  
"Манчестер Сити", "ПСЖ", "Барселона" };
```

```
var selectedTeams = teams.Where(t =>  
t.ToUpper().StartsWith("Б")).OrderBy(t => t);
```

```
foreach (string s in selectedTeams)  
    Console.WriteLine(s);
```

- Запрос `teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t)` будет аналогичен предыдущему. Он состоит из цепочки методов `Where` и `OrderBy`. В качестве аргумента эти методы принимают делегат или лямбда-выражение.
- Не каждый метод расширения имеет аналог среди операторов LINQ, но в этом случае можно сочетать оба подхода. Например, используем стандартный синтаксис `linq` и метод расширения `Count()`, возвращающий количество элементов в выборке.

Список используемых методов расширения LINQ

- **Select**: определяет проекцию выбранных значений
- **Where**: определяет фильтр выборки
- **OrderBy**: упорядочивает элементы по возрастанию
- **OrderByDescending**: упорядочивает элементы по убыванию
- **ThenBy**: задает дополнительные критерии для упорядочивания элементов по возрастанию
- **ThenByDescending**: задает дополнительные критерии для упорядочивания элементов по убыванию
- **Join**: соединяет две коллекции по определенному признаку
- **GroupBy**: группирует элементы по ключу
- **ToLookup**: группирует элементы по ключу, при этом все элементы добавляются в словарь
- **GroupJoin**: выполняет одновременно соединение коллекций и группировку элементов по ключу
- **Reverse**: располагает элементы в обратном порядке.

- All: определяет, все ли элементы коллекции удовлетворяют определенному условию
- Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
- Contains: определяет, содержит ли коллекция определенный элемент
- Distinct: удаляет дублирующиеся элементы из коллекции
- Except: возвращает разность двух коллекций, то есть те элементы, которые содержатся только в одной коллекции
- Union: объединяет две однородные коллекции
- Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обеих коллекциях
- Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
- Sum: подсчитывает сумму числовых значений в коллекции
- Average: подсчитывает среднее значение числовых значений в коллекции
- Min: находит минимальное значение
- Max: находит максимальное значение
- Take: выбирает определенное количество элементов

- Skip: пропускает определенное количество элементов
- TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
- SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
- Concat: объединяет две коллекции
- Zip: объединяет две коллекции в соответствии с определенным условием
- First: выбирает первый элемент коллекции
- FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
- Single: выбирает единственный элемент коллекции, если коллекция содержит больше или меньше одного элемента, то генерируется исключение
- SingleOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
- ElementAt: выбирает элемент последовательности по определенному индексу
- ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
- Last: выбирает последний элемент коллекции
- LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию

Выборка и проекция из базы данных

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

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

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

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

Теперь сделаем проекцию. Допустим, нам надо добавить в результат выборки название компании. Мы можем использовать метод Include для подсоединения к объекту связанных данных из другой таблицы: `var phones = db.Phones.Include(p=>p.Company)`. Но не всегда нужны все свойства выбираемых объектов. В этом случае мы можем применить метод Select для проекции извлеченных данных на новый тип:

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

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

```
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

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

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


Соединение таблиц

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

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

Метод Join принимает четыре параметра:

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

Агрегатные операции

Количество элементов в выборке

Метод Count() позволяет найти количество элементов в выборке:

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

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

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

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

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

SQL в Entity Framework

- В большинстве случаев разработчики смогут создать эффективные запросы с помощью методов и операторов LINQ. Однако в Entity Framework доступно также прямое выполнение sql-запросов.
- Для осуществления прямых sql-запросов к базе данных можно воспользоваться свойством Database, которое имеется у класса контекста данных. Данное свойство позволяет получать информацию о базе данных, подключении и осуществлять запросы к БД.

Например, получим строку подключения:

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

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

Получим все модели из таблицы Companies:

```
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":

```
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:

```
// вставка
int numberOfRowsInserted =
db.Database.ExecuteSqlCommand("INSERT INTO Companies
(Name) VALUES ('HTC')");
// обновление
int numberOfRowsUpdated =
db.Database.ExecuteSqlCommand("UPDATE Companies SET
Name='Nokia' WHERE Id=3");
// удаление
int numberOfRowsDeleted =
db.Database.ExecuteSqlCommand("DELETE FROM Companies
WHERE Id=3");
```

Обращение к функции

```
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);
}
```

Обращение к хранимой процедуре

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

СПИСОК ИСТОЧНИКОВ

- <https://metanit.com/sharp/entityframework/1.1.php>
- <https://professorweb.ru/my/entity-framework/6/level1/>