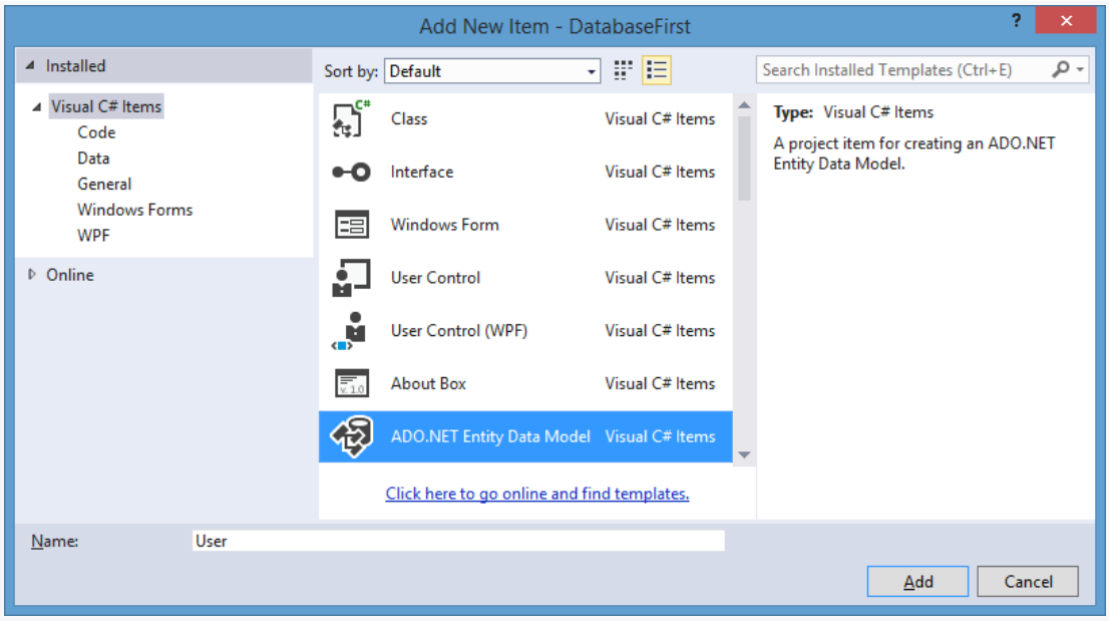
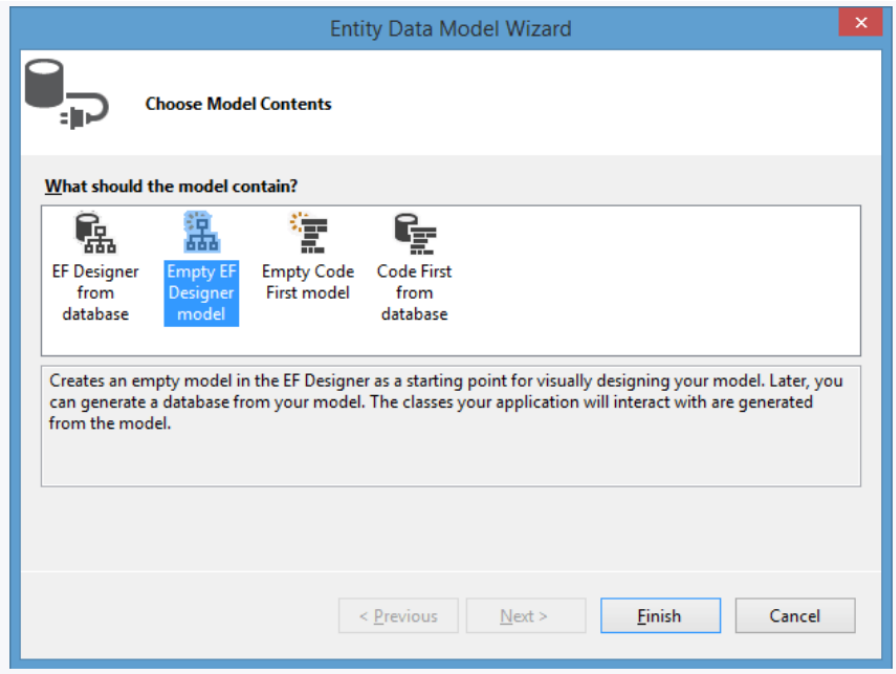
Создание модели данных

Entity Framework представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. Если традиционные средства ADO.NET позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работает с объектами.

Нажмем правой кнопкой мыши на проект в окне Solution Explorer и в появившемся списке выберем **Add -> New Item**. И затем в окне добавления нового элемента выберем **ADO.NET Entity Data Model**:

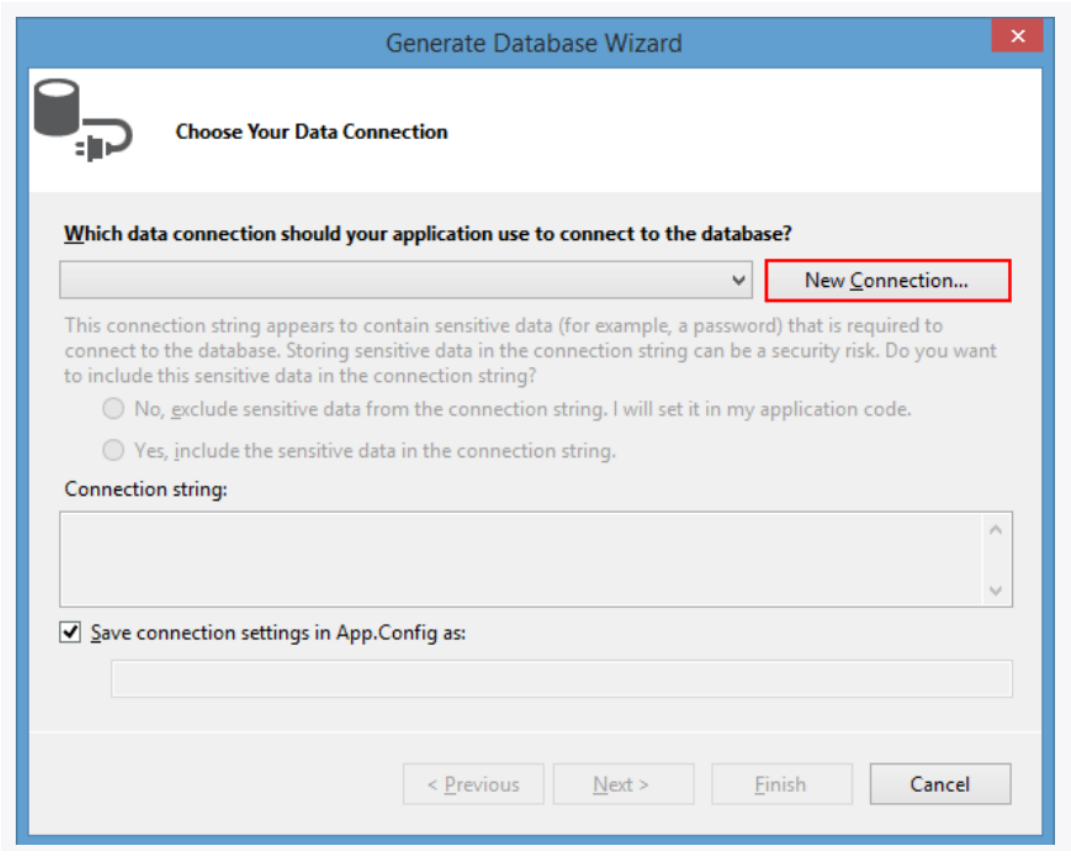


Введем название модели. Нажмем OK и нам откроется мастер создания модели. Если у нас Visual Studio с пакетами обновления SP2, SP3, то мастер создания модели выглядит следующим образом:



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

Создадим подключение к Базе Данных.



Выберем все таблицы, и создание имен таблиц во множественном числе.

CRUD-операции

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

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);
}
```

После добавления надо сохранить все изменения с помощью метода SaveChanges().

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 сохранятся. 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();
   }
}
```

Связи

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

Например, у нас определен следующий класс футбольной команды Team:

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

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

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

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

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

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

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

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.

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);
    }
}
```

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}");
    }
}
```

LINQ

Выборка

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
```

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

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

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

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

```
}
```

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

```
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);
}
```

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

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

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

В качестве альтернативы методу OrderBy можно использовать оператор orderby:

```
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()**:

```
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);
```

Группировка

Чтобы сгруппировать данные по определенным параметрам используются метод GroupBy(). Критерий, по которому проводится группировка, является ключом. Ключ мы можем получить через свойство Key, которое есть у группы.

```
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:

```
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);
```

Агрегатные операции

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

Количество элементов в выборке

Метод Count() позволяет найти количество элементов в выборке:

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

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

Минимальное, максимальное и среднее значения

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

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

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

Сумма значений

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

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