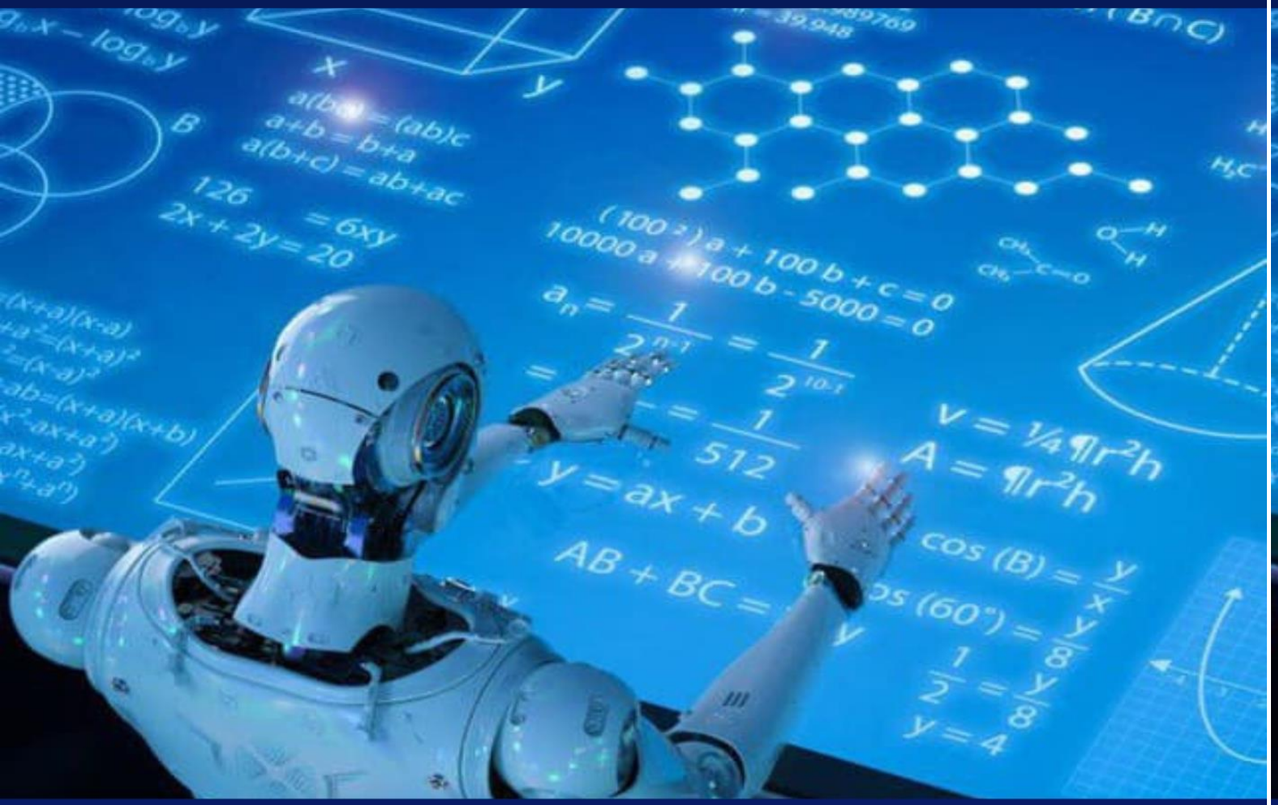
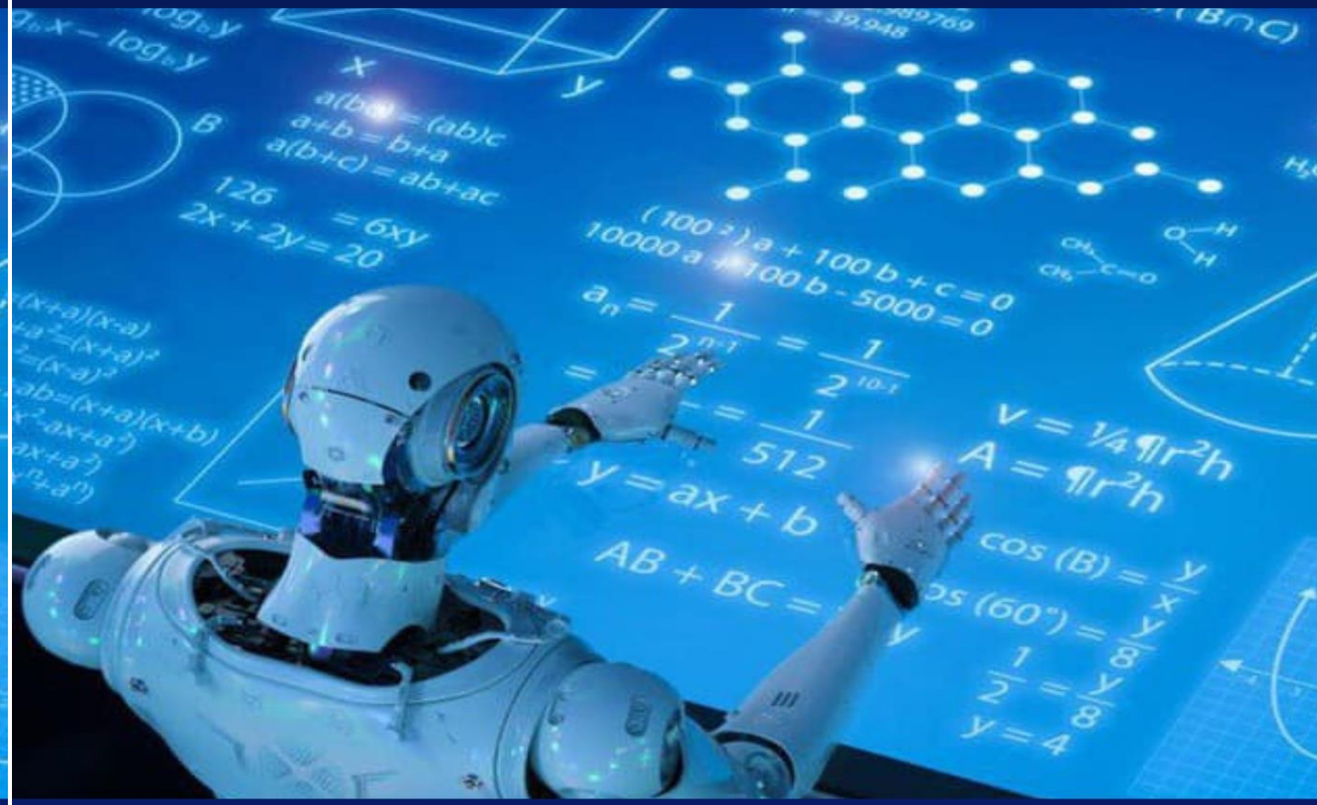


MUHAMEDIYEVA D.T., VARLAMOVA L.P.,
BAXROMOV S.A., ELOV B.B.

DASTURIY INJINIRING. ENTITY FRAMEWORK 6

O‘QUV QO‘LLANMA



MUHAMEDIYEVA D.T., VARLAMOVA L.P.,
BAXROMOV S.A., ELOV B.B.

O’ZBEKISTON RESPUBLIKASI OLIY VA O’RTA MAXSUS TA’LIM VAZIRLIGI

MIRZO ULUG‘BEK NOMIDAGI O’ZBEKISTON MILLIY UNIVERSITETI

Muhamediyeva D.T., Varlamova L.P.,

Baxromov S.A., Elov B.B.

“Dasturiy injiniring ”

Entity framework 6

o‘quv qo‘llanma

Toshkent-2023

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

The textbook for the course "Software Engineering" was compiled for students of the specialty 5330100 - "Information systems, mathematics and software" with the aim of teaching the use of programming languages, software development and database management systems. This course is taught to students for three semesters, according to which the manual includes three parts. “The Part Two” of this tutorial covers programming in Entity Framework 6.

Mualliflar:

D.T.Muhamediyeva	- Toshkent irrigatsiya va qishloq xo’jaligini mexanizatsiyalash muhandislari instituti milliy tadqiqot universiteti professori, t.f.d.
L.P.Varlamova	- Mirzo Ulug‘bek nomidagi O‘zbekiston Milliy universiteti, Hisoblash matematikasi va axborot tizimlari kafedrasi professori, t.f.d.
S.A. Baxromov	- Mirzo Ulug‘bek nomidagi O‘zbekiston Milliy universiteti, Hisoblash matematikasi va axborot tizimlari kafedrasi dotsenti, t.f.n.
B.B.Elov	- Mirzo Ulug‘bek nomidagi O‘zbekiston Milliy universiteti, Hisoblash matematikasi va axborot tizimlari kafedrasi dotsenti, t.f.n.

Taqrizchilar:

Matyaqubov A.S – O’zMU, Amaliy matematika va computer taxlili kafedrasi mudiri

Yakubov M.S. – TATU, Axborot texnologiyalari kafedrasi professori

O’quv qo’llanma O’zbekiston Respublikasi Oliy va o’rta maxsus ta'lim vazirligining 2022 yil “09” sentyabrdagi “302” –sonli buyrug’iga asosan nashrga tavsiya etilgan. Ro’yxatga olish raqami 302-0384.

Mundarija

KIRISH

Dasturiy injiniring (Entity Framework) ni o'zlashtirish uchun talabalarda ob'ektga yo'naltirilgan dasturlash, **C#** tili sintaksisi, **.Net Framework** bibliotekasi, **T-SQL** tili, **ADO.NET** asosi, **Visual Studio 2013** muhitida ishlash ko'nikmasiga ega bo'lishlari lozim.

O'quv qo'llanmada keltirilgan o'quv materiallarini o'zlashtirish natijasida talabalar **Entity Data Model**ni generatsiya qilishni, **DataBase First, Model First, Code first** yondashuvlarini, klasslar va ob'ektlarni ishlatishni, **LINQ** vositalari yordamida ma'lumotlar bazasidagi ma'lumotlarni boshqarishni, xatoliklarga ishlov berishni va dastur kodini optimallashtirishni o'rganishadi.
O'quv qo'llanma **8** bo'limdan iborat bo'lib, ular quyidagilar:

- **Entity Framework** ga kirish;
- Ma'lumotlar bilan ishlash. Yondashuvlar;
- **Entity Framework** asoslari;
- **LINQ to Entities**;
- **EntityFramework** da **SQL**;
- **Fluent API** va annotatsiyalar;
- **Entity Framework** da «vorislash»;
- **Entity Framework** da asinxronlik.

Foydalanilgan qisqartmalar
LINQ — Language Integrated Query (tuzilmalangan so'rovlar tili)
EF – Entity Framework (ma'lumotlarga доступни ta'minlovchi ob'ektga-mo'ljallangan texnologiya)
ORM — object-relational mapping ()
EDM — Entity Data Model ()
DB — Database (ma'lumotlar bazasi)
VS 2013 — Visual Studio 2013
OOP — Object-Oriented Programming (ob'ektga yo'naltirilgan dasturlash)
UML — Unified Modeling Language (unifitsirlangan modellashtirish tili)
TPH — Table Per Hierarchy (klasslar iyerarxiyasiga mos jadval)
TPT — Table Per Type (tipga mos jadval)
TPC — Table Per Concrete Type (har bir alohida tipga mos jadval)

Foydalanilgan terminlar

Klass (класс, class) — mantiqiy tuzilma hisoblanib, maydonlar (atributlar), metodlar va hodisalar majmuasidan iborat.

Ob'ektga yo'naltirilgan dasturlash (объектно ориентированное программирование, Object-Oriented Programming) — asosiy kontseptsiyalari ob'ekt va klass tushunchalari hisoblangan dasturlash paradigmasi.

Ob'ekt (объект, object) — o'zinng holati va harakatiga ega bo'lgan virtual fazodagi bir element bo'lib, xususiyatlari(atributlar)ning qiymatlari aniqlangan bo'lib, ular ustida bir qancha amallar (метод) larni bajarish mumkin. «**Klass nusxasi**» va «**ob'ekt**» terminlari sinonim hisoblanadi.

Klass maydoni yoki **atribut** (***переменная-член, data member, class field, instance variable***) — klass yoki ob'ektga mansub o'zgaruvchi. Ob'ektning barcha ma'lumotlari uning maydonlarida saqlanadi. Ushbu maydonlarga murojaat uning nomi orqali amalga oshiriladi. Har bir maydon tipi klass aniqlanganda ko'rsatiladi.

Ma'lumotlar tipi (*mun, type*) — qiymatlar va ular ustida aniqlangan amallar to'plami

Metod (метод, method) — muayyan klass yoki ob'ektga tegishli funksiya yoki protsedura. Metod biror amalni bajarish uchun zarur bo'lgan bir nechta operatorlardan iborat bo'lib, kiruvchi argumentlarga ega bo'lishi mumkin..

Element (сущность, entity) — real hayotdagi biror ob'ekt.

MySQL – erkin foydalaniladigan ma'lumotlar bazasini boshqaruv tizimi. **ADO.NET** – Microsoft .NETga asoslangan ma'lumotlarga доступни

ta'minlovchi texnologiya.

Entity SQL – SQL tiliga o'xshash til hisoblanib, Entity Framework da konseptual modellarda so'rovlarni amalga oshirishga mo'ljallangan.

LINQ to Entities – LINQ vositalari orqali ma'lumotlar bazasiga murojaat qilish interfeysi.

Model First – Vizual muharrir asosida edmx model va ma'lumotlar bazasini shaklantirish texnologiyasi.

Code First – C# tilida yozilgan model kodi asosida ma'lumotlar bazasini shakllantirish texnologiyasi.

Database First – Ma'lumotlar bazasi asosida edmx modelni shallantirish texnologiyasi.

1. ENTITY FRAMEWORKGA KIRISH

Reja:

1. **Entity Framework nima.**
2. **Ma'lumotlar bazalari bilan o'zaro ta'sir qilish usullari.**
3. **Nazorat savollari.**

Entity Framework nima.

Entity Framework – .NET Framework asosida ma’lumotlar bilan ishlashni ta’minlovchi maxsus ob’ektga yo’naltirilgan yondashuvni ifodalaydi. Agar an’anaviy **ADO.NET** vositalari orqali **DB**ga ulanishlar, **sql**-buyruqlar va boshqa shu turdagi ob’ektlar orqali **DB** bilan ishlash amalga oshirilsa, **Entity Framework** orqali yuqori darajadagi abstraktsiya hosil qilinib, **DB** va undagi ma’lumotlarni saqlagichlar tipidan qat’iy nazar abstraktsiyani tashkil qilishga imkon yaratadi.

Entity Framework da fizik darajada jadval, indeks, birlamchi va ikkilamchi kalitlar bilan ish ko’rsak, konseptual darajada ob’ektlar ustida amal bajariladi.

Entity Framework - 1.0 versiya – **2008** yil yaratilgan bo’lib, chekli funksionalga ega bo’lib, oddiy **ORM**ni va **DB** bilan ishlashda faqatgina **Database First** yondashuvni amalga taklif qilgandi.

2010 yilda **4.0** versiya yaratilgach, ushbu texnologiya **DB** bilan ishlashda yetakchi texnologiya sifatida ishlatilmoqda. Ushbu versiyada **Model First** va **Code First** kabi yondashuvlar yaratildi.

2012 yilda ishlab chiqilgan **5.0** versiyada qo’shimcha imkoniyatlar taklif qilindi.. **2013** yilda esa **Entity Framework 6.0** yaratilib, ma’lumotlarga asinxron доступни taklif qiladi.

Entity Framework ning markaziy kontsepsiyasi sifatida **element** yoki **entity** ishlatiladi. Element – muayyan ob’ektni ifodalovchi ma’lumotlar to’plamidan iborat. Shuning uchun ushbu texnologiyada jadvallar o’rniga ob’ektlar va ularning to’plamlari ustida amallar bajariladi.

Ixtiyoriy element – xuddi real hayotdagi ob’ekt kabi bir qator xususiyatlardan iborat.Agar element insonni ifodalasa, biz unda **ism, familiya, bo’yi, yoshi** va **og‘irligi** kabi xususiyatlarni aniqlashimiz mumkin.

Xususiyatlar oddiy **int** tipiga mansub qiymatni ifodalamasligi ham mumkin. Xususiyat sifatida murakkab tuzilmaga ega bo’lgan kompleks tuzilma ham bo’lishi mumkin. Har bir elementda bir yoki bir qancha xususiyatlar mavjud bo’lib, xususiyatlar orasida elementni bir qiymatli aniqlovchi xususiyat ham mavjud bo’ladi. Ushbu turdagi xususiyatlarni **kalitlar** deb yuritiladi.

Shuningdek elementlar bir-biri bilan birga-bir, birga-ko'p va ko'pga- ko'p aloqada bo'lishi mumkin. Elementlar o'zaro birlamchi va ikkilamchi kalitlar orqali bog'lanadi.

Entity Framework ning muhim jihatlaridan biri **DB** ma'lumotlar ustida **LINQ** so'rovlarini amalga oshirish imkoniyati hisoblanadi. **LINQ** orqali nafaqat **DB**dagi jadvallarida saqlanayotgan ma'lumotlarni, balki turli assotsiativ aloqaga ega bo'lgan ob'ektlarni ham olishimiz mumkin.

Entity Framework ning asosiy tushunchalaridan biri **Entity Data Model** hisoblanadi. Ushbu model loyihadagi elementlarga mos klasslarni **DB**dagi jadvallarga mos qo'yadi.

Entity Data Model uchta darajadan iborat: **konseptual, saqlagich** va **moslashtirish**. Konseptual darajada dasturda ishlatiladigan **element** klasslari aniqlanadi. Saqlagich darajasida jadvallar, ustunlar, jadvallar va ma'lumotlar tiplari o'rtasidagi munosabat aniqlanad. Mos qo'yish darajasi yuqoridagi ikkita darajani o'zaro bog'lovchi komponent bo'lib, element klass xususiyatlari va jadval ustunalari o'rtasida moslik o'rnatiladi.

Natijada biz loyihadagi klasslar orqali **DB**dagi jadvallar bilan ishlashimiz mumkin.

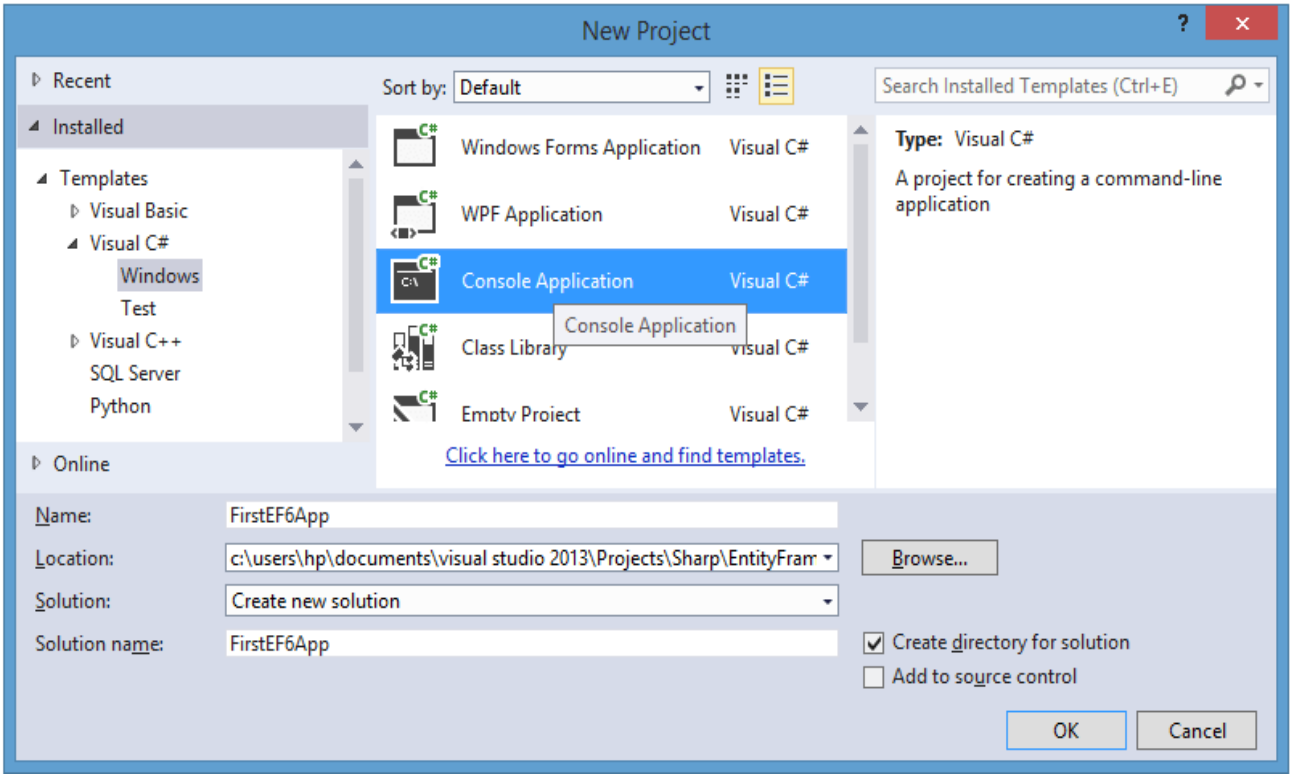
Ma'lumotlar bazasi bilan bog'lanish usullari

Entity Framework da **DB** bilan ishlashda uchta usuldan foydalanish mumkin:

- **Database first**: **Entity Framework** orqali muayyan **DB** modeliga mos klasslar to'plami generatsiya qilinadi;
- **Model first**: dasturchi avvalo ma'lumotlar modelini hosil qiladi, so'ngra **Entity Framework** serverda real **DB**ni hosil qiladi;
- **Code first**: dasturchi **DB**da saqlanish lozim bo'lgan ma'lumotlar modelini hosil qiladi. So'ngra **Entity Framework** ushbu model asosida DBda mos jadvallarni generatsiya qiladi.

Entity Framework asosida yaratilgan birinchi loyiha. Code First yondashuvi.

Entity Framework bilan ishni boshlashdan avval, **VS 2013** da birinchi loyihani yaratishimiz lozim. Buning uchun **VS 2013** muhitini ishga tushirib, loyiha turidan **Console Application**ni tanlaymiz:



1-Rasm

So’ngra loyihamizga ma’lumotlarni ifodalovchi yangi klassni qo’shamiz. Biz yaratayotgan loyiha dastur foydalanuvchilari bilan bog‘liq bo’lsin. Shuning uchun loyihamizga yangi **User** klassini hosil qilamiz:

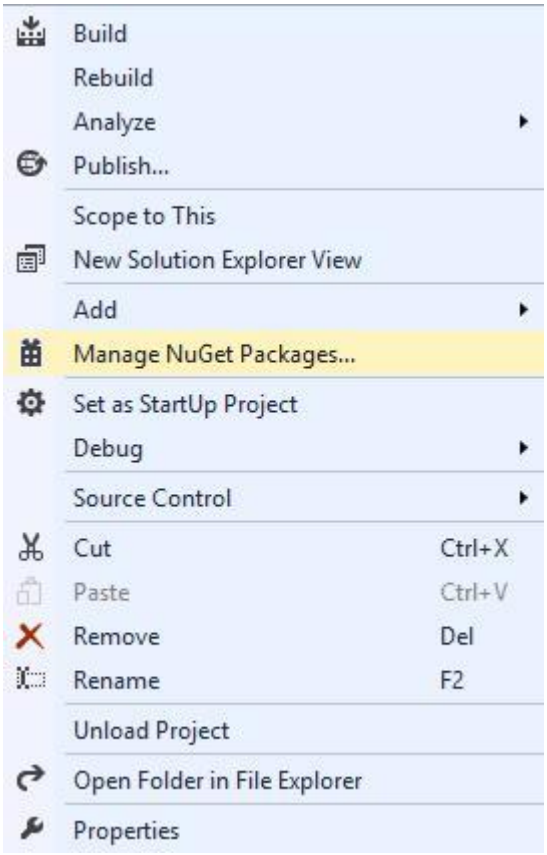
```
public class User
{
public int Id { get; set; }
public string Name { get; set; }
public int Age { get; set; }
}
```

Ushbu klass bir nechta avto xususiyatlarni o’zida saqlaydi. Har bir xususiyat **DB**dagi jadvalning muayyan ustuniga mos tarzda shakllantirilishi lozim.

Entity Framework da **Code First** yondashuv asosida ish ko’rilganda, **DB** jadvalining birlamchi kalitini ko’rsatishni talab qilinadi. **DB** generatsiya qilinayotganda **Entity Framework** birlamchi kalit sifatida **Id** yoki **[Klass_nomi]Id** kabi formadagi xususiyatlarni tushunadi. Bizning misolda ushbu turdagi xususiyat **Id**

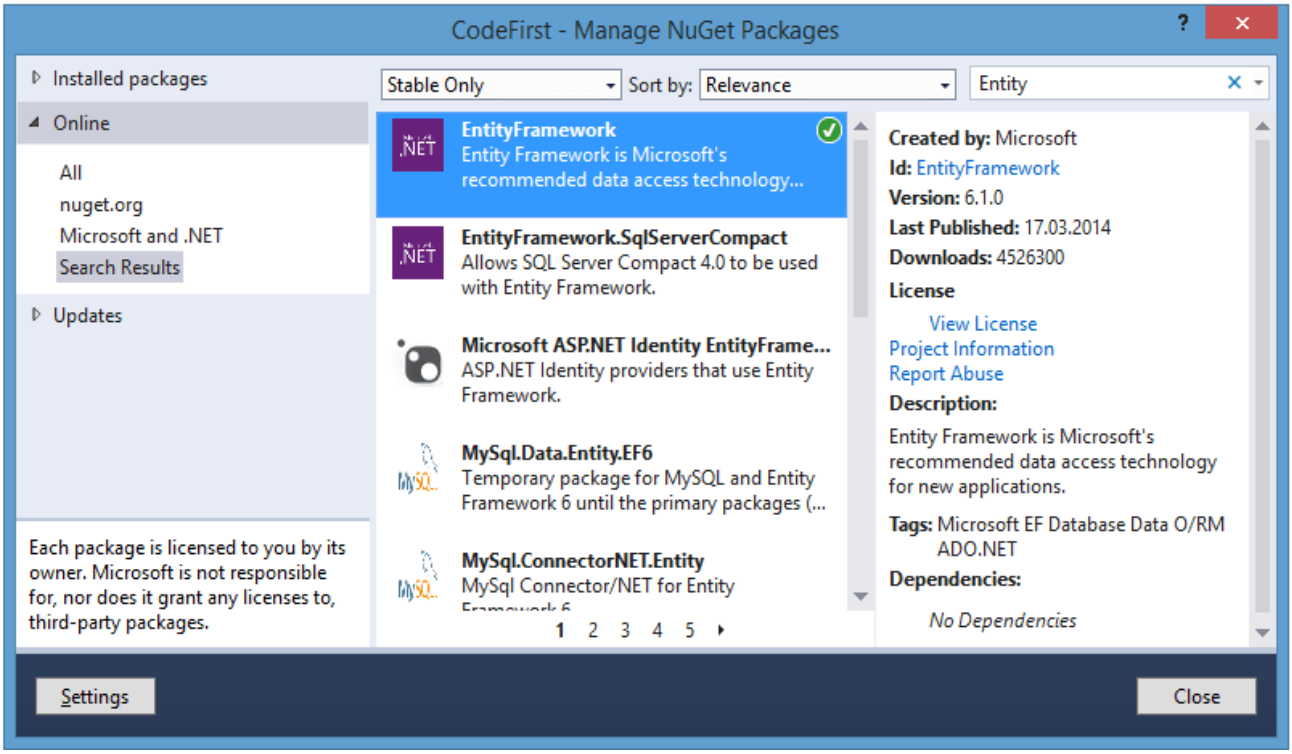
hisoblanadi. Agar biz kalitni boshqa ustun deb ko'rsatmoqchi bo'lsak, **C#** tilida qo'shimcha mantiqni tashkil qilishimiz lozim.

Endi **DB** bilan ishlash uchun ma'lumotlar kontekstidan foydalanamiz. Ma'lumotlar konteksti **DB** va loyiha klasslari o'rtasida vositachi hisoblanadi. Bizning loyihada **Entity Framework** uchun biblioteka qo'shib qo'yilmagan. **Entity Framework** bibliotekasini qo'shish uchun loyiha nomiga sichqonchaning o'ng tugmasini bosib, menyudan **Manage NuGet Packages...** qismni tanlash lozim:



2-Rasm

So'ngra, hosil qilingan muloqot oynasidan **NuGet**-paketlari bilan ishlash uchun "**Entity**" kalit so'zini izlash qismiga kiritib, hosil qilinga ro'yxatdan **Entity Framework** ni tanlaymiz va o'rnatamiz:



3-Rasm

Loyihamizda **Entity Framework** paketi o'rnatilgach, yangi **UserContext** klassini hosil qilamiz:

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

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

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

Entity Framework funksionalligini klasslar tashkil qiladi. Ushbu klasslar **System.Data.Entity** nomlar fazosida joylashgan. Ushbu nomlar fazosidan eng ko'p ishlatiladigan klasslar quyidagilar:

- **DbContext**: **DB** bilan aloqani tashkil qilish uchun zarur bo'lgan ma'lumotlar kontekstini ifodalaydi.
- **DbModelBuilder**: **C#** tilidagi klasslar bilan **DB** dagi elementlarni moslashtiradi.
- **DbSet/DbSet<TEntity>**: **DB**da saqlanayotgan elementlar to'plamini ifodalaydi.

DB bilan **Entity Framework** orqali ishlayotgan ixtiyoriy dasturda bizga ma'lumotlar konteksti (**DbContext** dan vorislangan) va **DbSet** ma'lumotlar to'plami (**DB** jadvallari bilan aloqa uchun) kerak bo'ladi. Bizning misolda ma'lumotlar konteksti sifatida **UserContext** klassidan foydalanamiz.

Ushbu klass konstruktorida bosh klass konstruktori chaqiriladi. Ushbu konstruktorda "**DbConnection**" satri parameter sifatida uzatilgan bo'lib, bu qiymat **DB**ga ulanishni ifodalaydi. Biz konstruktordan foydalanmasak, ulanish satri nomi ma'lumotlar konteksti bilan bir xil nomda bo'lishi shart.

Shuningdek klassda bitta **Users** xususiyati aniqlangan bo'lib, unda **User** ob'ektlar to'plami saqlanadi. Ushbu kalssda ma'lumotlar konteksti to'plami uchun **DbSet<T>** klassidan foydalanilgan. Ushbu xususiyat orqali **DB**dagi **User** jadvali bilan aloqa o'rnatilinadi.

Keyingi qadamda **DB**ga ulanishni hosil qilishimiz lozim. Buning uchun loyihadagi konfiguratsiya faylidan foydalanamiz. Oddiy loyihalarda konfiguratsiya fayli sifatida - **App.config** (bizning hol uchun), veb-loyihalrda esa - **web.config** faylidan foydalaniladi. **Entity Framework** loyihaga qo'shilgach **App.config** fayli quyidagi mazmunga ega:

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

```
</startup>

<entityFramework>

<defaultConnectionFactory

type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework"
/>

<providers>

    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer"
/>
</providers>

</entityFramework>

</configuration>
```

</configSections> yopiluvchi tegdan so'ng, quyidagi elementni qo'shib qo'yamiz:

```
<connectionStrings>

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

Loyihadagi barcha ulanishlar **connectionStrings** sektsiyasida aniqlanishi lozim va har bir ulanish **add** elementi orqali shakllantiriladi. **UserContext** klass konstruktorlarida ulanish satri sifatida "**DbConnection"**dan foydalanilgan. Shuning uchun ulanish satridagi **name** atributi qiymati name="DBConnection"ga teng.

Ulanish satridagi ulanish satri **connectionString** atributi orqali aniqlanadi. Bizning loyihada ulanish satrida **userstore.mdf DB** bilan ishlash ko'rsatilgan. Endi loyihamizdagi **Program.cs** faylida quyidagi o'zgartirishlarni amalga oshiramiz:

```
using System;
namespace FirstEF6App
{
class Program
{
static void Main(string[] args)
{
using (UserContext db = new UserContext())

{
// Ikkita User ob'ektini hosil qilamiz
```

```
User user1 = new User { Name = "Ahmad", Age = 33 }; User user2 = new User { Name = "Nodir", Age = 26 };

//              Ularni DBga qo'shamiz db.Users.Add(user1); db.Users.Add(user2); db.SaveChanges(); Console.WriteLine("Ob'ektlar DBga saqlanadi");

//              DBдан объектларни оламиз ва консолга чикарамиз var users = db.Users; Console.WriteLine("Ob'ektlar ro'yxati:");

foreach (User u in users)

{
Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Age);
}
}
Console.Read();
}
}

}
```

UserContext klassi **DbContext** klassi orqali **IDisposable** interfeysini tadbiq qilgani uchun, **UserContext** bilan ishlashda **using** konstruktsiyasidan foydalanishimiz mumkin. **using** konstruktsiyasida ikkita **User** ob'ekti hosil qilingan va ular **DB**ga saqlangan:

```
db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();
```

DBdan ma'lumotlar ro'yxatini olish uchun ma'lumotlar kontekstidagi **Users** xususiyatidan foydalanish yetarli:

```
var users = db.Users;
```

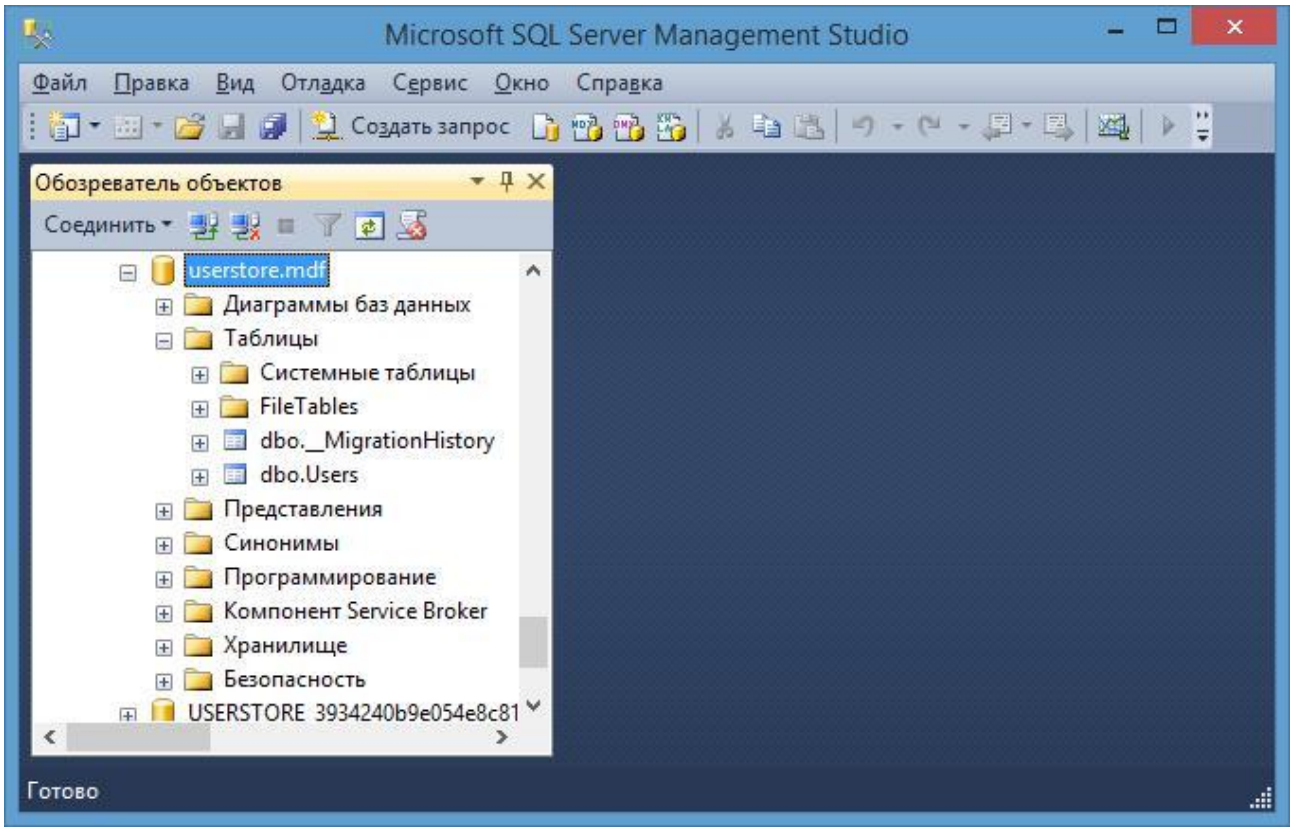
Dastur ishga tushirilgach, konsolga quyidagi natija hosil qilinadi:

Ob'ektlar DBga saqlanadi
Ob'ektlar ro'yxati:
1.Tom - 33
2.Sam - 26

Entity Framework vositalari orqali **DB** ob'ektlarini oddiy va qulay boshqarish ta'minlanadi. Bunda biz **DB** va unda mos jadvallarni hosil qilishimiz shart emas. Ushbu amallarni barchasini **Entity Framework** o'zi ma'lumotlar konteksti va modellar asosida bajaradi. Agar bizda **DB** mavjud bo'lsa, **Entity Framework** uni qayta yaratmaydi.

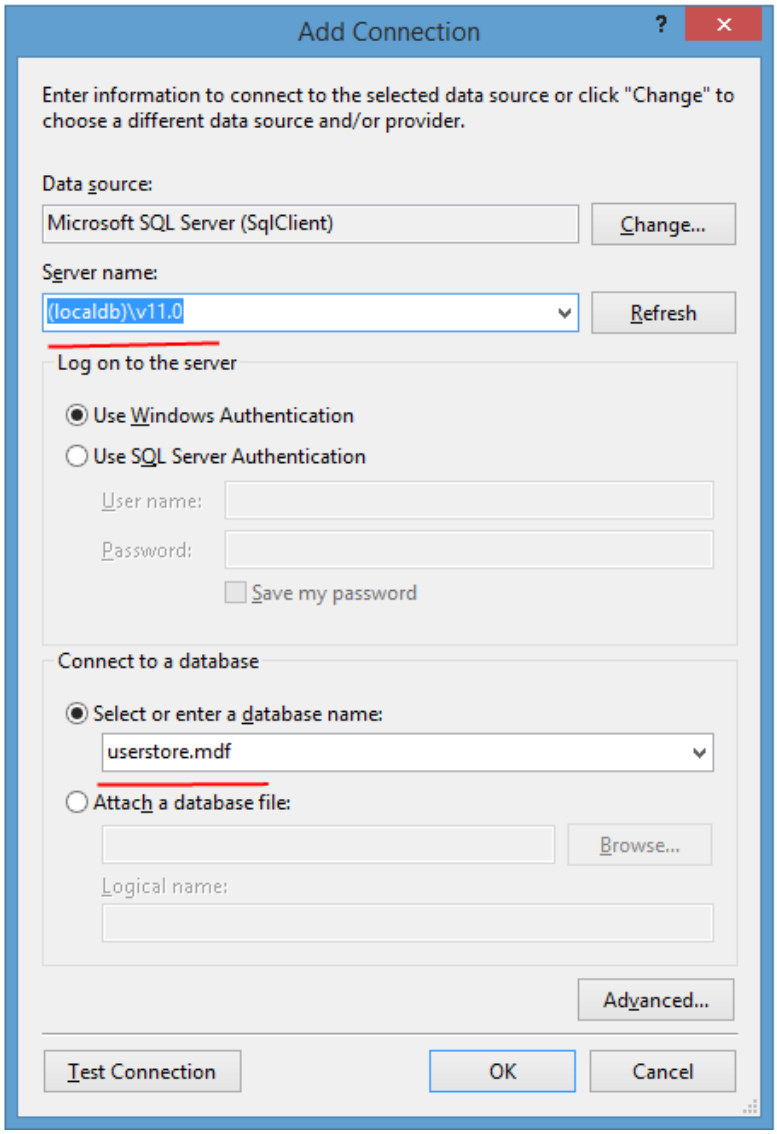
Bizning vazifamiz **DB**da saqlanishi lozim bo'lgan modelni va kontekstni aniqlash hisoblanadi. Shuning uchun ushbu yondashuv **Code First** deb nomlangan. Ya'ni avvalo kod yoziladi, so'ngra unga mos **DB** va jadvallar hosil qilinadi.

DB qayerda saqlanadi? Loyihadagi ma'lumotlar konteksti va modellar mos **DB**ni **Visual Studio** dagi **Database Explorer** oynasidan yoki **SQL Server Management Studio** maxsus boshqaruv vositasidan ko'rish mumkin.



5-Rasm

DBni **Visul Studio** dan ko'rish uchun menyuning **View->Other Windows->Database Explorer** qismini tanlashimiz lozim. So'ngra hosil qilingan **Database Explorer** oynasidan **Connect to Database** ni tanlab yangi bazaga ulanishni amalga oshiramiz.



6-Rasm

DBga ulanish muloqot oynasida server sifatida **(localdb)\v11.0** ni tanlaymiz yoki muayyan **DB** joylashgan manzilni ko'rsatamiz. **DB** fizik fayllari **C:\Program Files\Microsoft SQL Server\MSSQL11.SQLEXPRESS\MSSQL\DATA** katalogida joylashgan bo'lishi mumkin. Bizning misolda **DB – DbConnection.mdf** kabi faylda saqlanadi.

Nazorat savollari:

1. Entity Frameworkning maqsadi nima?
2. Entity Frameworkni o'rnatish bosqichlarini keltiring va tavsiflang.
3. 3.Entity Framework-ning paydo bo'lishiga nima sabab bo'ladi?
4. Entity Framework ma'lumotlar bazalari bilan qanday o'zaro ta'sir qiladi?
5. Entity Framework da sinflar qanday yaratilgan?

2. MA’LUMOTLAR BILAN ISHLASH. YONDASHUVLAR

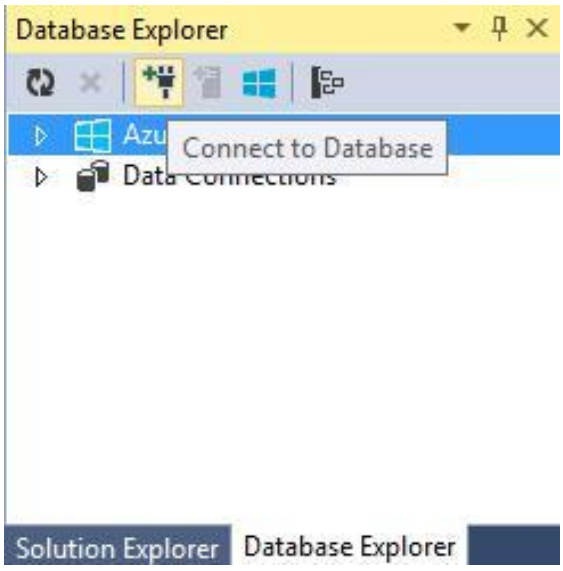
Reja:

1. **Mavjud ma'lumotlar bazasiga birinchi kodni qo'llash.**
2. **Birinchi koddagi ismlar.**
3. **Nazorat savollari.**

Mavjud ma’lumotlar bazasiga Code First

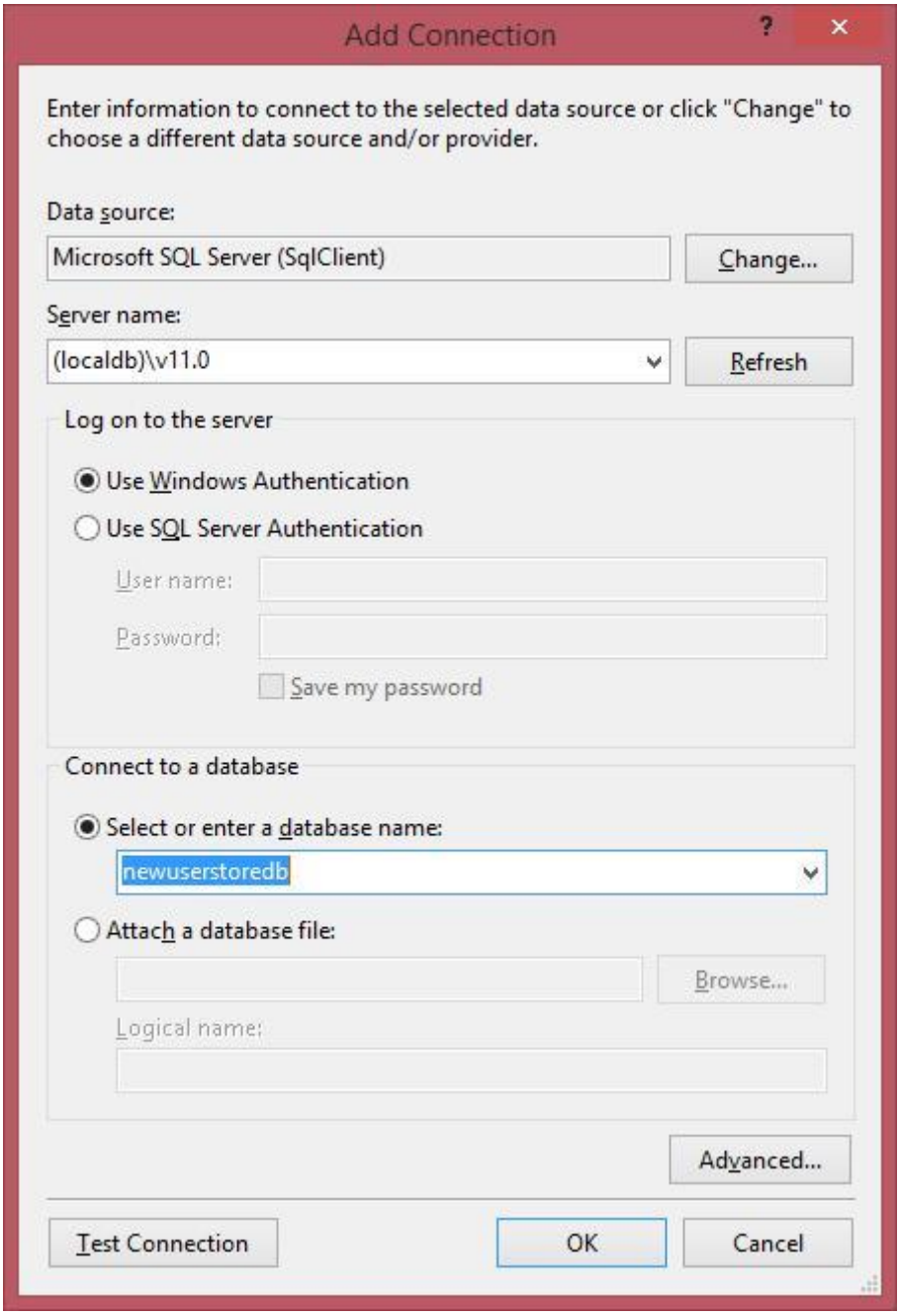
Birinchi bo’limda **Entity Framework** ga asoslangan birinchi dasturda biz **Code First** yondashuvdan foydalandik. Ushbu yondashuv juda oddiy, qulay va moslashuvchan hisoblanadi. Ko’p hollarda loyiha yaratilayotgan vaqtda **DB** mavjud bo’ladi. Ushbu holda ham **Code First** yondashuvdan foydalanish mumkin. Ba’zi dasturchilar ushbu yondashuvni **Code Second** deb yuritishadi. Yuqoridagi mulohazalarni misolda ko’rib chiqamiz.

Yangi loyihani yaratamiz. So’ngra loyiha uchun **DB**ni hosil qilamiz. **Visual Studio** da menyuning **View->Other Windows->Database Explorer** qismini tanlaymiz. Undagi **Database Explorer** oynasidan yangi bazaga **Connect to Database** qismini tanlash lozim:



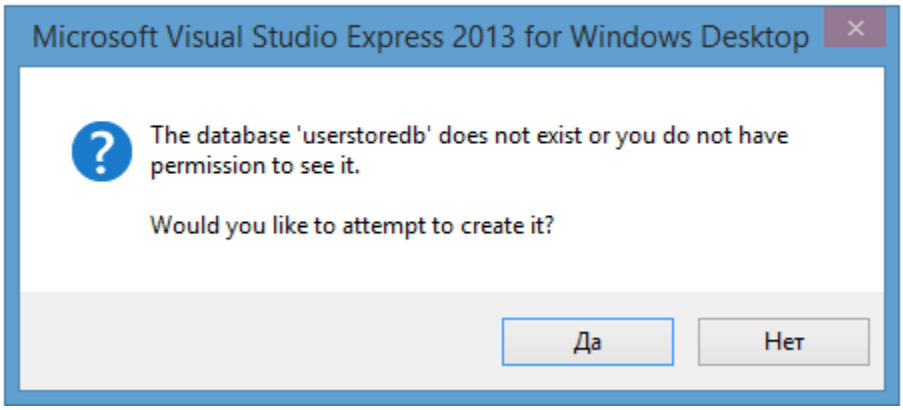
7-Rasm

DBga ulanish muloqot oynasida server sifatida **(localdb)\v11.0** ni tanlaymiz. Ushbu holda **MS SQL Server DB** bilan ishlash uchun **localdb** dvijogidan foydalanamiz **DB** sifatida **newuserstoredb** ni tanlaymiz.



8-Rasm

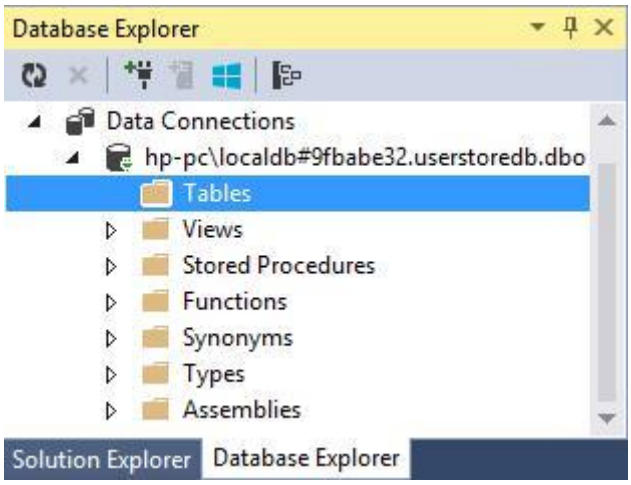
Agar ushbu **DB** mavjud bo’lmasa, uni yaratish muloqot oynasi taqdim etiladi:



9-Rasm

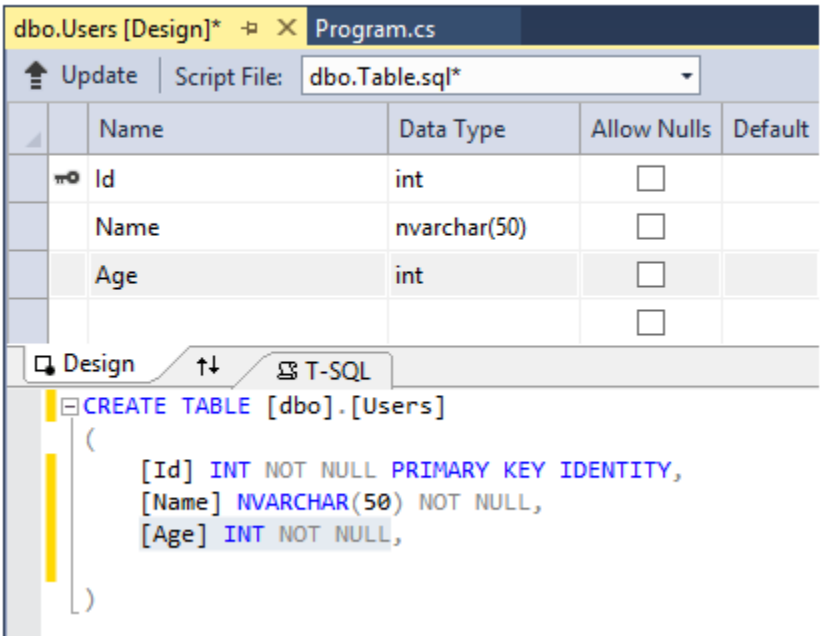
«**Да**» tugmasini bosamiz. Natijada **Database Explorer** oynasida hosil qilingan

DB keltiriladi:



10-Rasm

Hosil qilingan **DB** bo'sh bo'lib, unda yangi jadval hosil qilamiz. **Tables** qismiga sichqonchaning o'ng tugmasini bosib, menyudan **Add New Table** qismni tanlaymiz. So'ngra dizayner rejimida quyidagi maydonlarni hosil qilamiz:

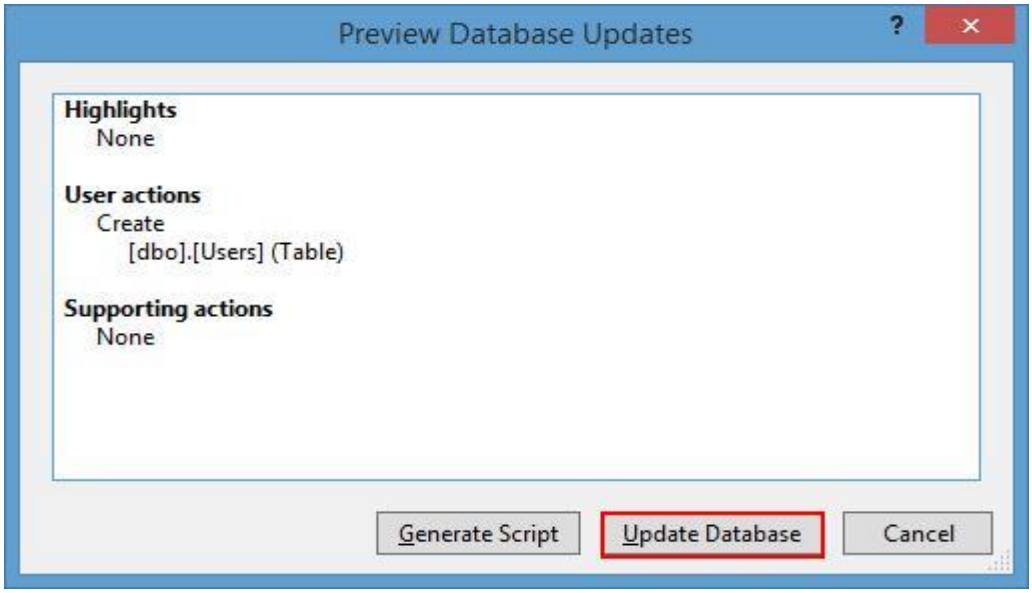


11-Rasm

```
CREATE TABLE [dbo].[Users]
(
[Id] INT NOT NULL PRIMARY KEY IDENTITY, [Name] NVARCHAR(50) NOT NULL, [Age] INT NOT NULL

)
```

T-SQL yoki grafik rejimida jadval tuzilmasi, nomini va ustun tiplarini aniqlaymiz va barcha amallar bajarilgach, **Update** tugmasini bosish lozim.

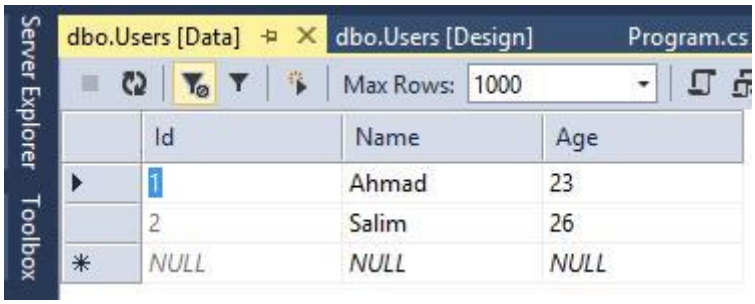
Yangi oynada **DB**da amalga oshirilishi lozim bo'lgan o'zgarishlar keltiriladi:



12-Rasm

Update Database tugmasini bosamiz. Shundan so'ng **DB**da **Users** jadvali hosil qilinadi. **Database Explorer** oynasini yangilab, **Tables** qismini tanlasak, hosil qilingan **Users** jadvalini ko'rish mumkin.

Keyingi qadamda ushbu jadvalda bir qancha ma'lumotlarni kiritishimiz mumkin. Buning uchun **Database Explorer** oynasidan jadvalni tanlab, sichqonchaning o'ng tugmasini bosib, muloqot oynasidan **Show Table Data** ni tanlashimiz lozim. Natijada bizda ma'lumotlar bilan ishlash uchun forma hosil qilinadi:



13-Rasm

Shu bilan **DB** tayyor. Endilikda loyihadagi konfiguratsiya faylida ushbu **DB** bilan ulanish satrini hosil qilishimiz lozim. **Solution Explorer** oynasidan **App.config** faylini topib, uni ochamiz. **</configuration>** yopiluvchi tegidan so'ng quyidagi sektsiyani qo'shib qo'yamiz:

```
<connectionStrings>

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

Endi ma'lumotlar modeli va kontekstiga mos klasslarni hosil qilamiz. Loyihamizda avval **User** klassini qo'shamiz:

```
public class User

{

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

}
```

Shuningdek, ma’lumotlar kontekstini qo’shamiz:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
namespace CodeFirstApp
{
class UserContext : DbContext
{

public UserContext()
:           base("DbConnection")
{ }

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

}
```

Ma’lumotlar konteksti konstruktorida biz **App.config** konfiguratsiya faylida keltirilgan ulanish satri nomini keltiramiz. Konfiguratsiya faylida

<add name="UserContext" connectionString="...

Kabi ulanish satridan foydalansak, ma’lumotlar konteksti klassi konstruktorida parameter ko’rsatilmasligi ham mumkin.

DBdagi ma’lumotlarni o’qib olish uchun quyidagi koddan foydalanamiz:

```
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);

}

}
```

Code First да номлар

DBda jadval va uning ustunlarini hosil qilish uchun **Entity Framework**da nomlarni aniqlashda bir qator kelishuvlar (qoidalar) mavjud. Ushbu qoidalar asosida jadval, ustun, tip nomlari ko’rsatiladi. Ushbu qoidalardan ba’zilarini ko’rib chiqamiz.

Tiplarni moslashtirish

SQL Server va **C#** dagi tiplar mosligi:

- **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 va NOT NULL

Jadvaldagi barcha birlamchi kalitlar **NOT NULL** kabi aniqlanadi. Uzatma tipiga (**string, array**) xususiyatlarga mos ustunalar **DB**da **NULL** kabi, qiymatli tiplar (**DateTime, bool, char, decimal, int, double, float**) - **NOT NULL** kabi aniqlanadi.

Agar xususiyat **Nullable<T>**tipiga mansub bo'lsa, u **NULL** ustuniga mos qo'yiladi.

Kalitlar

Entity Framework da jadvalda birlamchi kalit mavjudligi talab qilinadi. Ushbu kalit orqali ob'ektlar nazorat qilinadi. **Entity Framework** da kalitlar **Id** yoki **[Tip_nomi]Id** kabi formatga ega bo'ladi. Masalan, **Post** klassda **PostId** ustuni ushbu vazifani bajaradi.

Kalitlar asosan **int** yoki **GUID** tipiga mansub bo'ladi. Shuningdek, boshqa tipga mansub kalitlarni hosil qilish mumkin.

Jadval va ustun nomlari

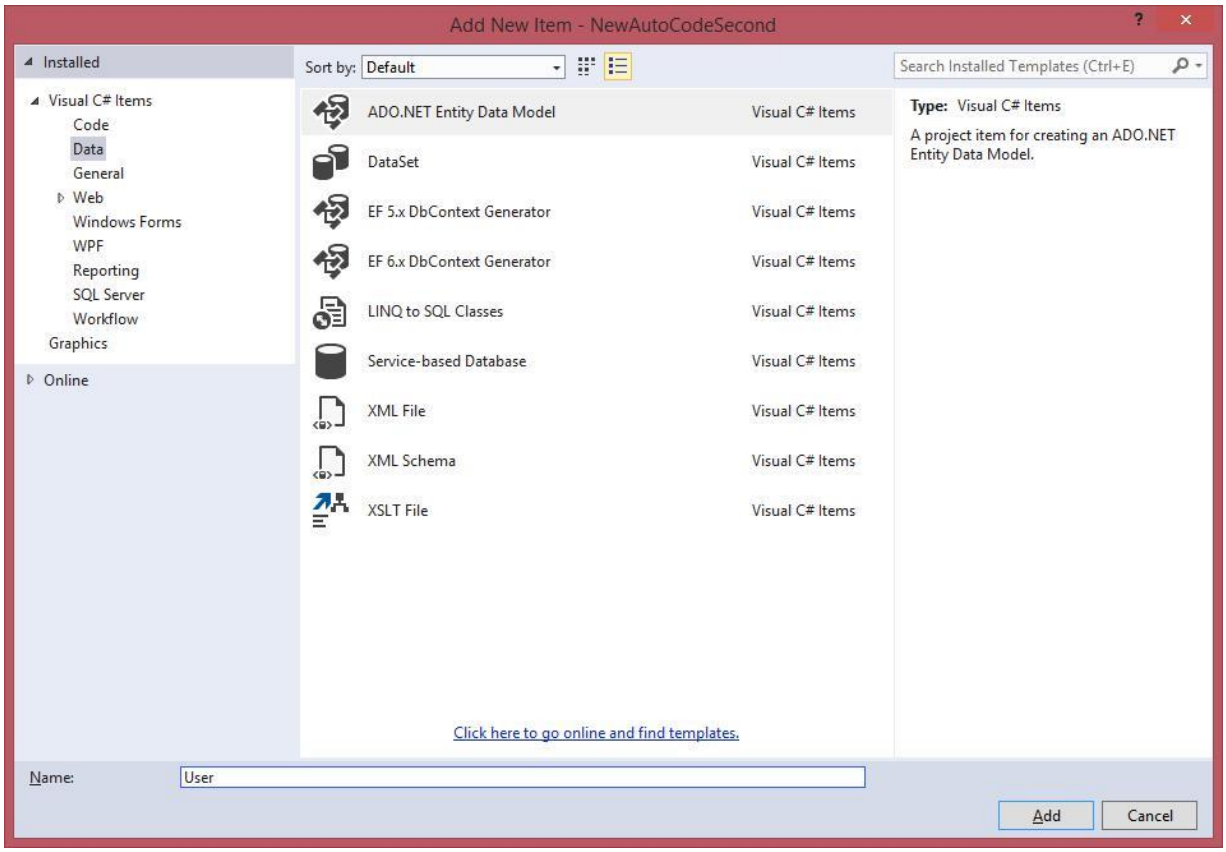
Entity Framework dagi **PluralizationService** klassi orqali modelga mos klass va jadval nomlari o'rtasida moslik o'rnatiladi. **User** klassiga mos jadval nomi – **Users, Person** klassiga mos jadval nomi – **People** kabi bo'lishi mumkin.

Jadval ustunlarini nomlari klass xuxsusiyatlari nomlariga mos bo'lishi lozim. Agar bizni jadval va ustun nomlari qanoatlantirmasa, **Fluent API** yoki **annotatsiyalar** orqali boshqa mexanizmdan foydalanishimiz mumkin.

Code First avtomatizatsiyasi

Agar **DB** da jadvallar aniqlanga bo'lsa, ularga mos klasslarni hosil qilish juda ko'p amallar va vaqtni talab qilishi mumkin. Ushbu amallar agar jadvallar soni ko'p bo'lmagan holda amalga oshirib bo'lmaydi.

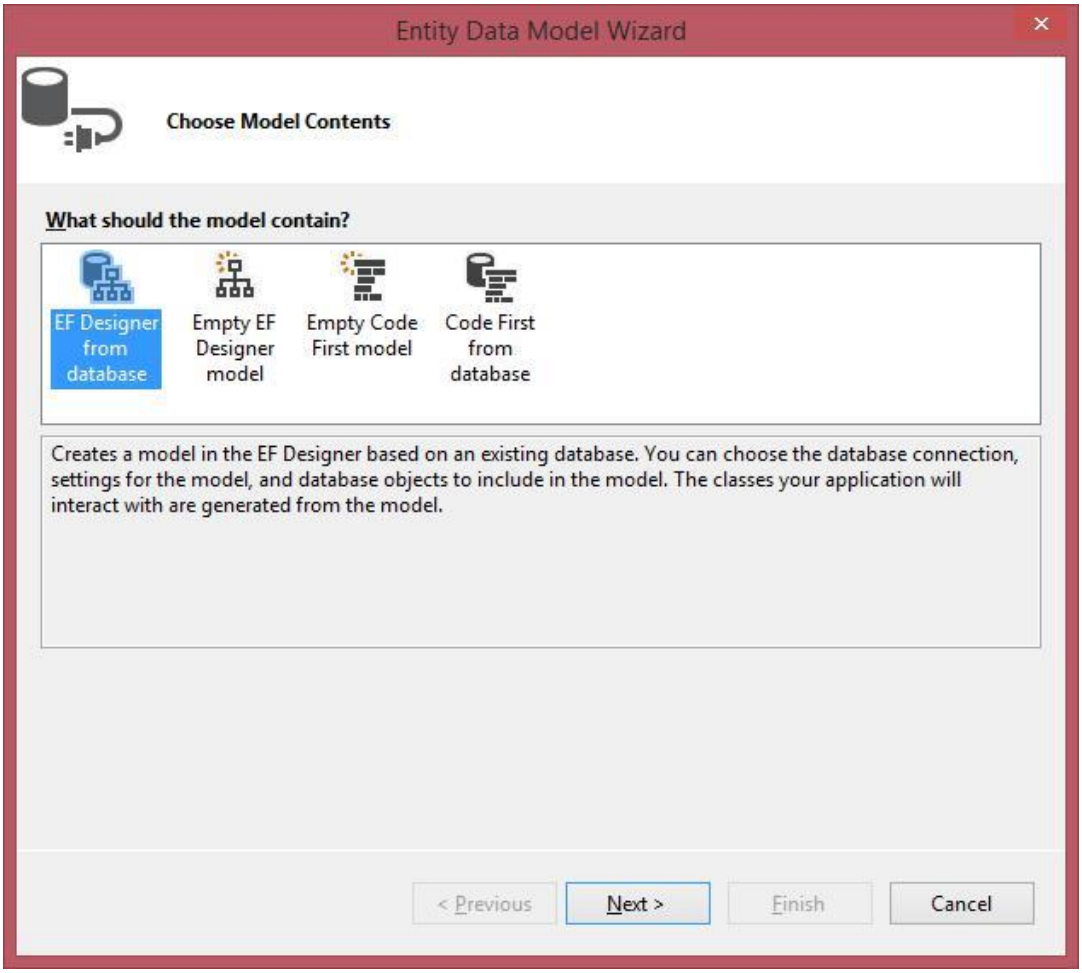
Visual Studio 2013 ning **SP3** yangilash paketi o'rnatilgan bo'lsa, ushbu jarayonni avtomatlashtirish mumkin. Buning uchun loyihada yangi **ADO.NET Entity Data Model** elementini qo'shamiz:



14-Rasm

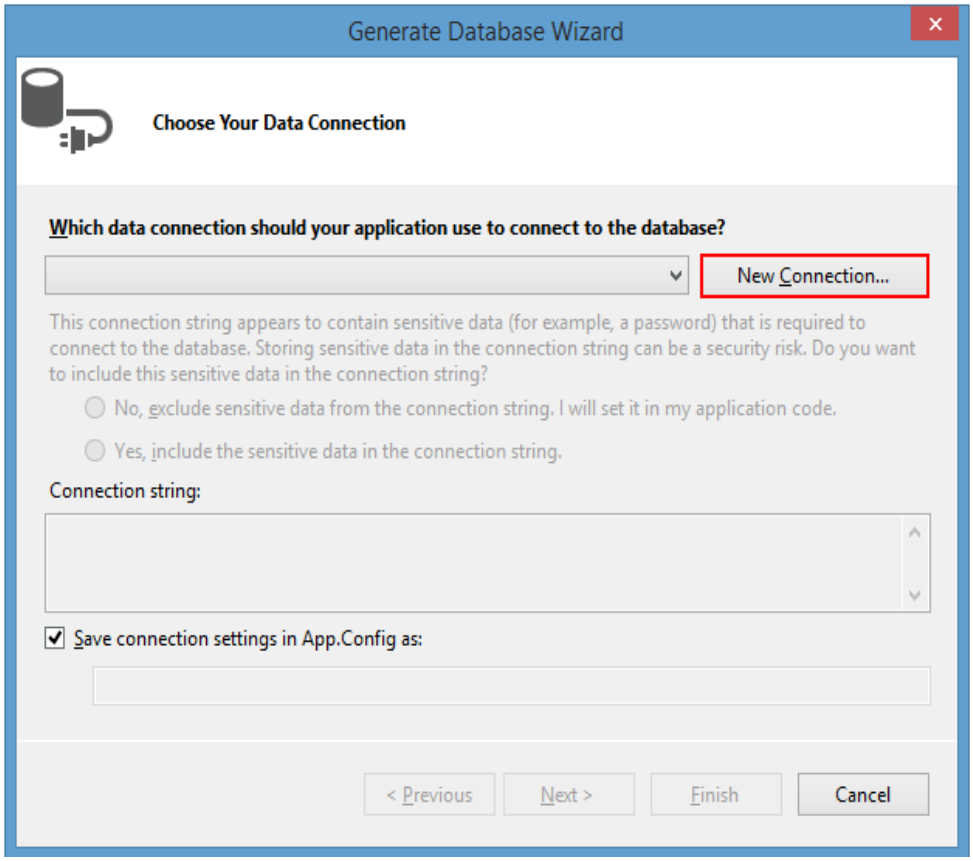
OK tugmasi bosilsa, bizda modelni hosil qilish muloqot oynasi hosil qilinadi.

So’ngra ushbu muloqot oynasidan **Code First from database** qismini tanlash lozim.



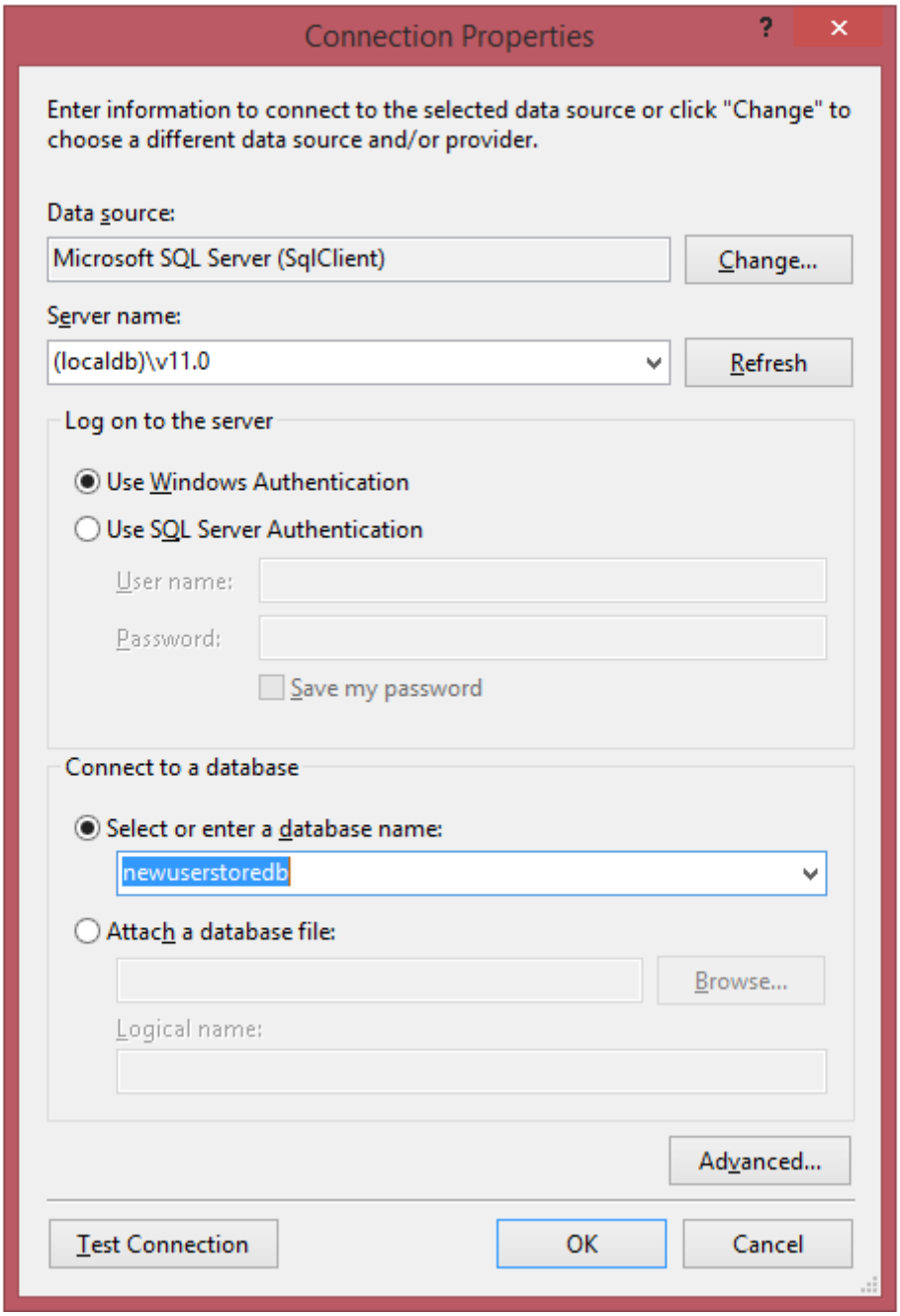
15-Rasm

Keyingi qadamda mavjud **DB**ga ulanishni hosil qilishimiz lozim.



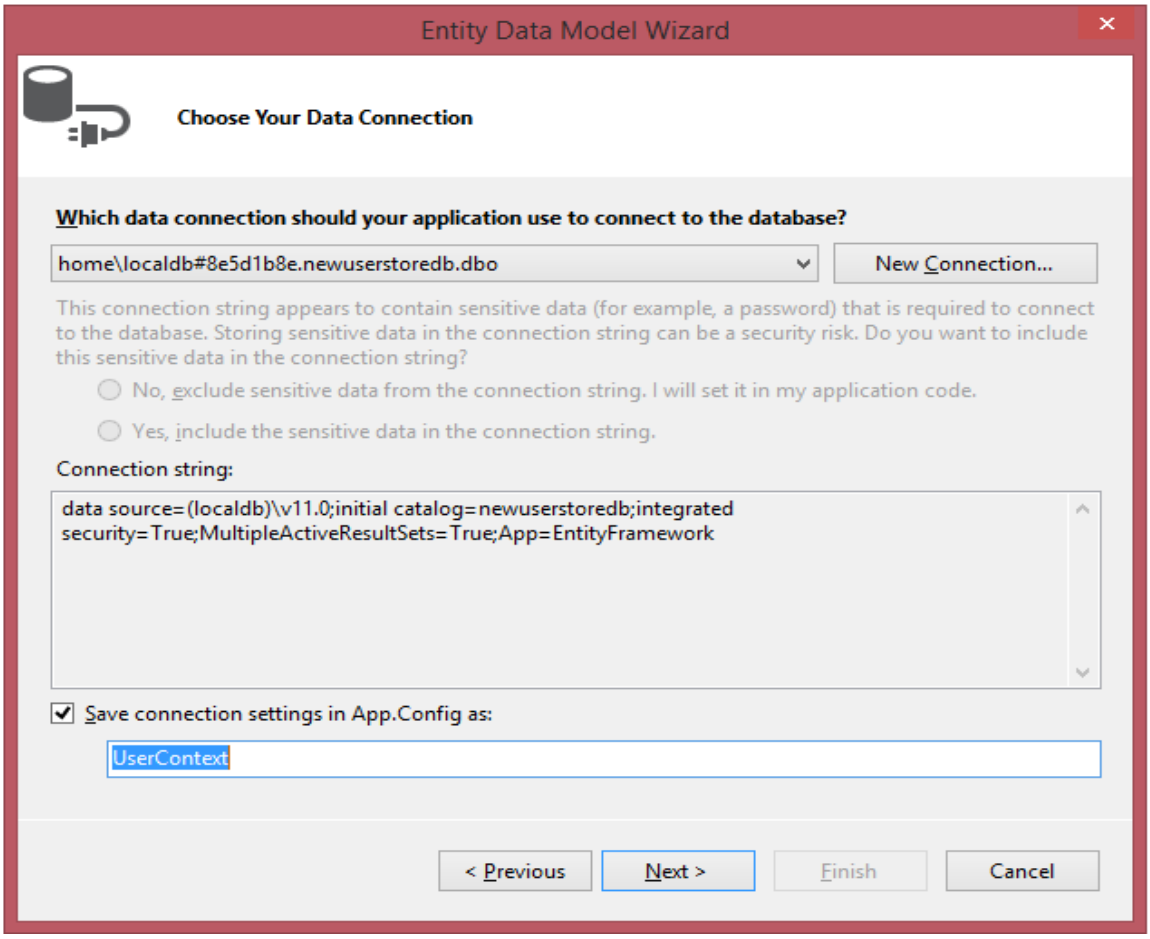
16-Rasm

Ushbu muloqot oynasidan **New Connection** tugmasini bosamiz va hosil qilingan oynadan **server** va **DB**ni tanlashimiz lozim:



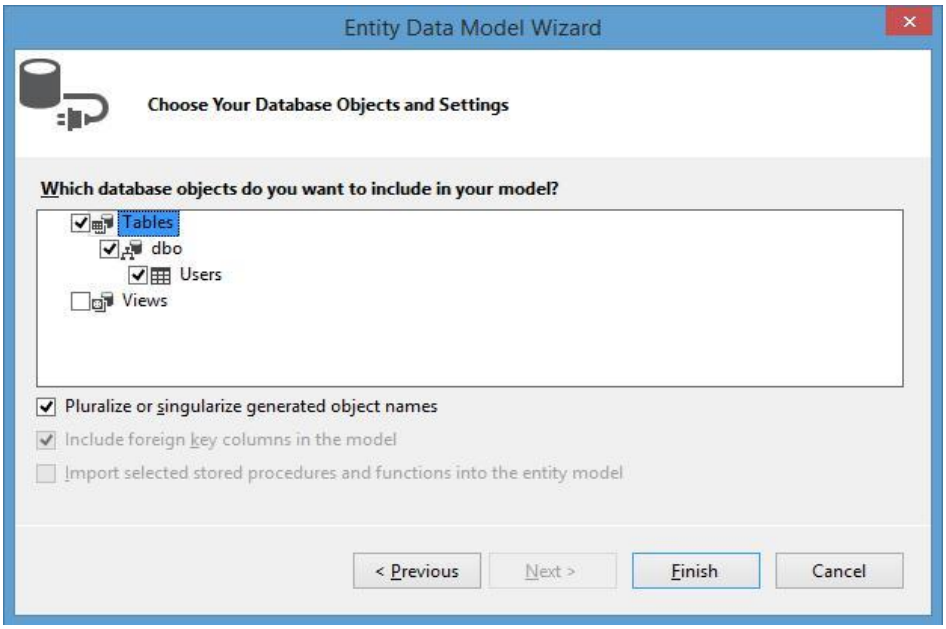
17-Rasm

Ushbu muloqot oynasida tanlangan serverga (**(localdb)\v11.0)** ulanish namoyish qilingan. Shuningdek, ushbu muloqot oynasida **App.config** konfiguratsiya faylida ishlatiladigan ulanish satri nomini ham keltirishimiz mumkin. Uni misol sifatida **UserContext** ga o’zgartiramiz:



18-Rasm

So’ngra **Next** tugmasini bosamiz va keying qadamda biz **DB**dagi ishlashimiz zarur bo’lgan jadvallarni tanlashimiz lozim.



19-Rasm

So’ngra **Finish** tugmasini bosing. Shundan so’ng **DB**dagi jadvallarga mos model klasslari generatsiya qilinadi. Bizning misoda quyidagi klass generatsiya qilinadi:

```
namespace NewAutoCodeSecond
{
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema; using
System.Data.Entity.Spatial;

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

[Required]
[StringLength(50)]
public string Name { get; set; }

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

Shuningdek, **App.config** faylida ulanish satri hosil qilingan:

```
<connectionStrings>
<add name="UserContext" connectionString="data source=(localdb)\v11.0;initial
catalog=newuserstore.mdf;integrated

security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Endi loyihamizga ma’lumotlar konteksti klassini shakllantirishimiz lozim.

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

namespace NewAutoCodeSecond

{
class UserContext : DbContext
{
```

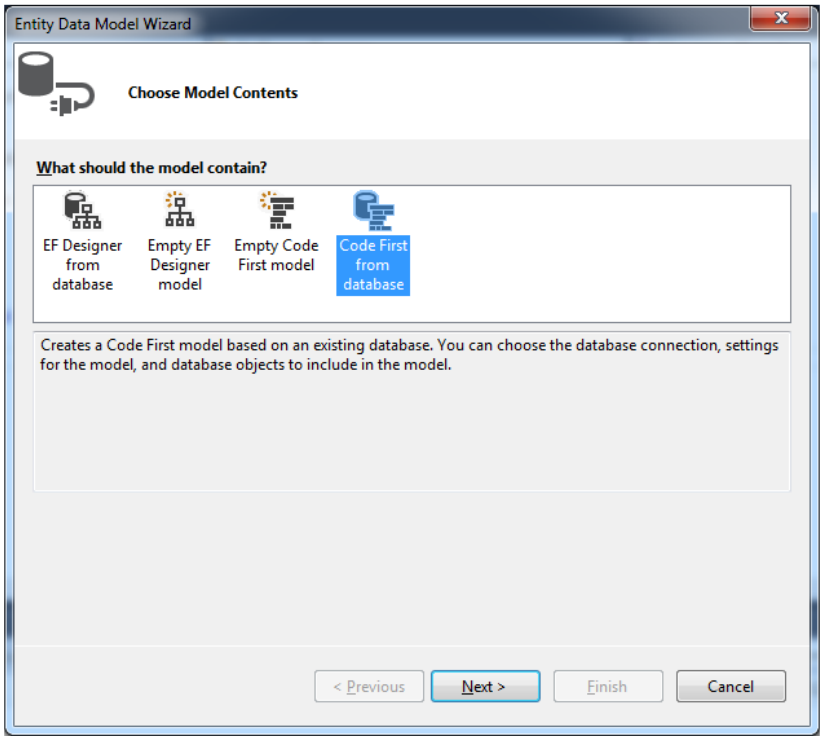
```
public UserContext()
:           base("UserContext")
{ }
public DbSet<User> Users { get; set; }
}
}
```

Endi biz **DB** bilan ishlashimiz mumkin:

```
using System;

namespace NewAutoCodeSecond
{
class Program
{

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

}
Console.ReadKey();
}
}
}
```

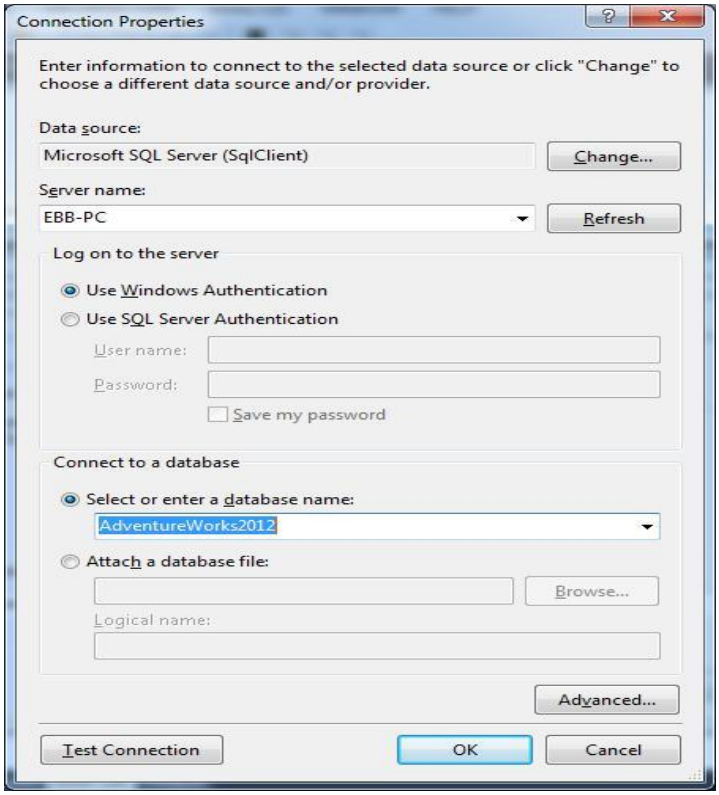
Yuqoridagi texnologiya asosida yana bir misolni ko’rib chiqamiz:

1. **VS 2013** da yangi (**GetPerson**) konsol loyihasini hosil qiling.
2. Loyihaga yangi **ADO.NET Entity Data Model** elementini qo’shing.
3. Muloqot oynasidan **Code First from database** qismini tanlang.



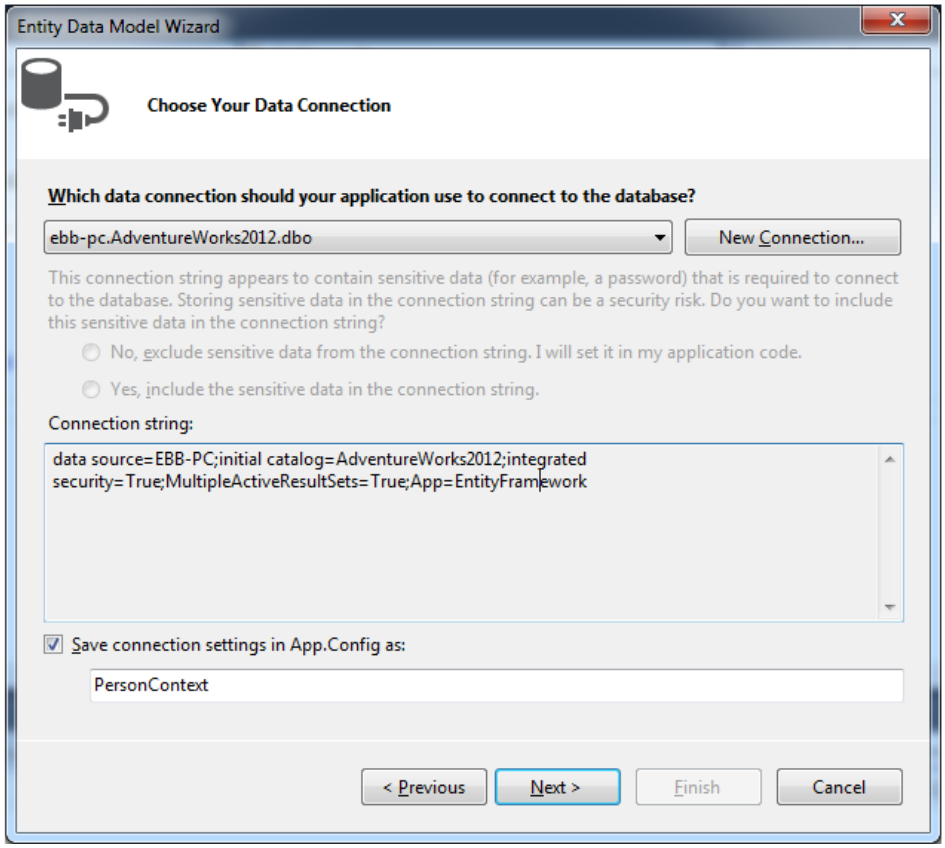
20-Rasm

4. **DB**ga yangi ulanishni (**New Connection**) hosil qiling.



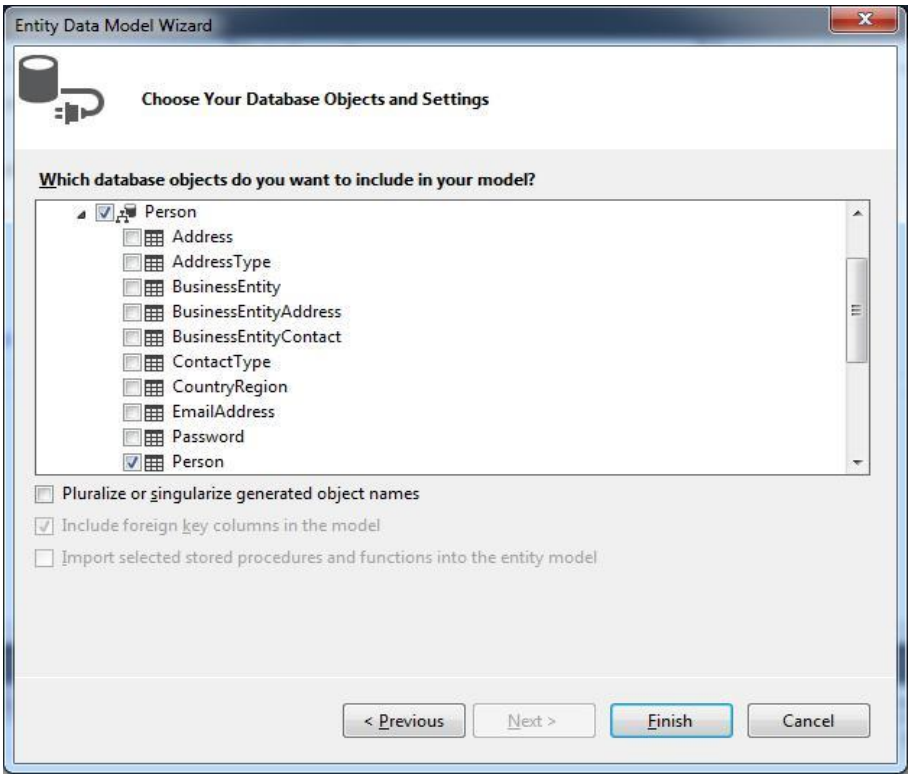
21-Rasm

5. Ushbu muloqot oynasida **App.config** konfiguratsiya faylida ishlatiladigan ulanish satri nomini **PersonContext** ga o’zgartiring.



22-Rasm

6. AdventureWorks2012 **DB**dagi Person.Person jadvalini tanlang.



23-Rasm

7. **Finish** tugmasini bosing.

8. Natijada quyidagi Person klassi generatsiya qilinadi**:**

```
namespace GetPerson
{
using System;

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema; using
System.Data.Entity.Spatial;

[Table("Person.Person")]
public partial class Person
{
[Key]

[DatabaseGenerated(DatabaseGeneratedOption.None)] public int BusinessEntityID {
get; set; }

[Required]

[StringLength(2)]
public string PersonType { get; set; }

public bool NameStyle { get; set; }

[StringLength(8)]
public string Title { get; set; }

[Required]
[StringLength(50)]
public string FirstName { get; set; }

[StringLength(50)]

public string MiddleName { get; set; }

[Required]
[StringLength(50)]
public string LastName { get; set; }

[StringLength(10)]
public string Suffix { get; set; }

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

```
[Column(TypeName = "xml")]
public string AdditionalContactInfo { get; set; }

[Column(TypeName = "xml")]
public string Demographics { get; set; }

public Guid rowguid { get; set; }

public DateTime ModifiedDate { get; set; }
}
}
```

9. **App.config** faylida ulanish satri hosil qilingan:

```
<connectionStrings>

<add name="PersonContext" connectionString="data source=EBB-PC;initial
catalog=AdventureWorks2012;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />

</connectionStrings>
```

10. Loyihada PersonContext ma’lumotlar konteksti klassini shakllantiring:

```
using System;

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

namespace GetPerson
{
class PersonContext : DbContext
{

public PersonContext()
:           base("UserContext")
{ }
public DbSet<Person> Persons { get; set; }
}
}
```

11. Endi AdventureWorks2012 **DB**dagi Person.Person jadvaliga mos **Persons** ob’ekti bilan ishlash mumkin. Barcha xodimlar ro’yxatini aniqlash (namuna):

```
using System;
```

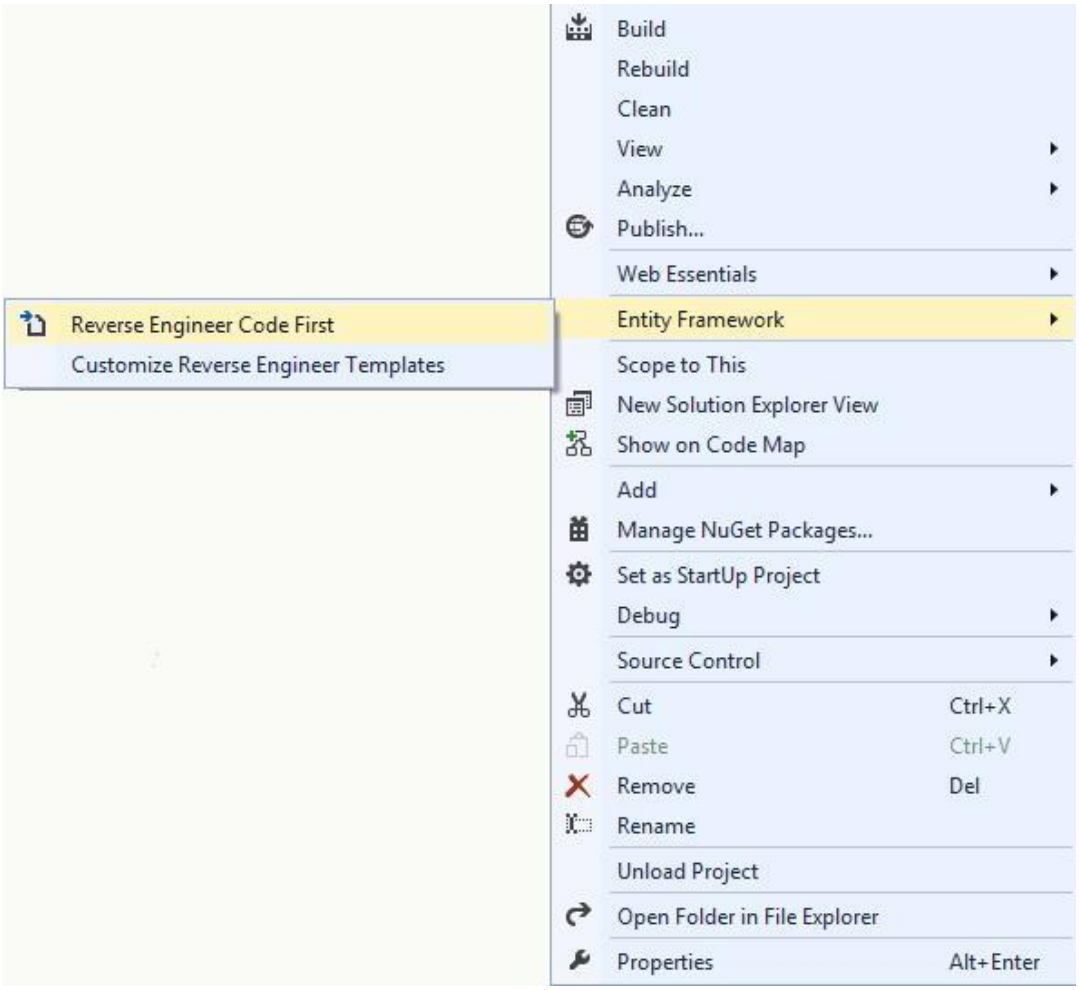
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GetPerson
{

class Program
{
static void Main(string[] args)
{
using (PersonContext db = new PersonContext())
{

foreach (Person u in db.Persons)

                Console.WriteLine("{0}. {1} {2}", u.BusinessEntityID, u.LastName,
u.FirstName);

}
Console.ReadKey();
}
}

}
```

Code First va EF Power Tools

Yuqorida keltirib o'tilgan texnologiya bilan bir qatorda **Microsoft** kompaniyasi bizga ushbu jarayonlarni avtomatlashtirish imkonini beradigan instrumentni taklif qiladi. Ushbu instrument **EF Power Tools** deb nomlanadi.

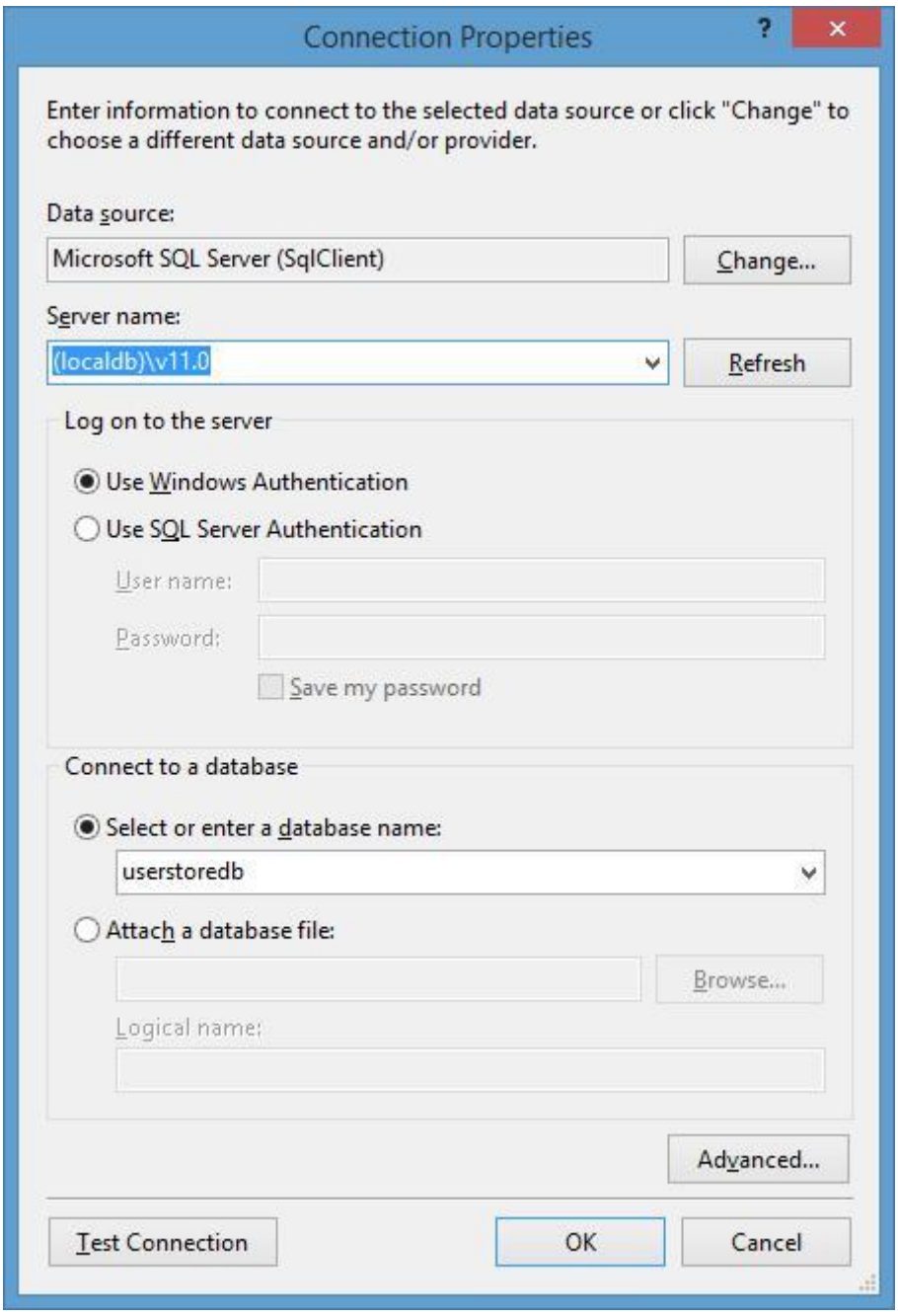
EF Power Tools orqali **Visual Studio** ga mos so'zlash amalga oshiriladi. Ushbu sozlashni **Entity Framework Power Tools** dan o'qib olishingiz mumkin. **Entity Framework Power Tools** faqatgina **Visual Studio** ning to'liq versiyalarida ishlaydi.

Agar sizda **Visual Studio** ning to'liq versiyasi o'rnatilgan bo'lsa, **Solution Explorer** (Обозреватель решений) oynasida loyiha nomida sichqonchani o'ng tugmasi bosilganda, kontekstli meyuda **Entity Framework** qismi hosil bo'lishi lozim:



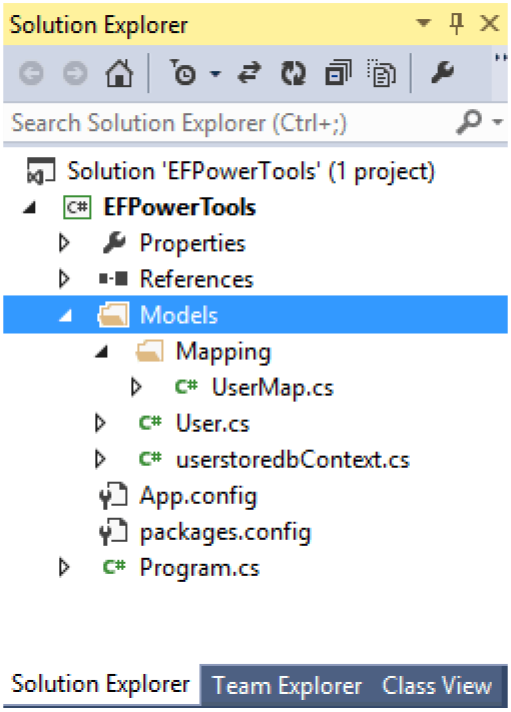
24-Rasm

Ushbu menyudan **Entity Framework –> Reverse Engineer Code First** ni tanlaymiz. Natijada bizda **DB**ga ulanish muloqot oynasi taqdim etiladi:



25-Rasm

Avvalgi bo’limda keltirilgan **DB**ni tanlash lozim. Shundan so’ng loyihada **Models** papkasi hosil qilinadi. Ushbu papkada barcha hosi qilingan klasslar mavjud. **DB**dagi har bir jadvalga mos klasslar va ularga mos ma’lumotlar konteksti klassi ushbu papkada joylashgan:



26-Rasm

Bizning **DB** da faqat bitta **Users** jadvali mavjud bo'lganligi sababli,avtomatik tarzda **User** klassi generatsiya qilingan bo'lib, u jadval tuzilamasini ifodalaydi:

```
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; }
}
}
```

Ma'lumotlar konteksti klassi **userstoredbContext** esa quyidagi mazmunga ega:

```
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());
}
}
}
```

Ushbu klassda ikkita konstruktor mavjud. Static konstruktor orqali ma'lumotlarni boshlangích inistiliazatsiya qilish amalga oshiriladi. Bizning xolda u hech qanday amal bajarmaydi. Standart konstruktor orqali bosh klassga (**DbContext**) murojaat qilinib, unga **DB** bilan ulanish satri **base("Name=userstoredbContext")** parametr sifatida ulanadi.

Users jadvali bilan ishlash uchun ma'lumotlar kontekstida **public DbSet<User> Users { get; set; }** xususiyati e'lan qilingan.

OnModelCreating metodida model hosil qilish uchun amallar bajariladi.Bizning holda **UserMap** klassi yordamida klasslar va **DB** o'rtasida aloqa shakllantiriladi.

UserMap klassi jadval ustunlari va klass xususiyatlari o'rtasida moslikni amalga oshiradi:

```
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");
}

}
}
```

Dasturdagi boshqa amallarni **Code First** yondashuviga o'xshash tarzda amalga oshirish mumkin. Biz jadvalga yangi **User** ob'ektini qo'shib qo'ymoqchi bo'lsak, quyidagi koddan foydalanamiz:

```
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());
}
}
}
```

Real hayotda **DB** murakkab tuzilmaga ega bo'ladi. Shuning uchun generatsiya qilinayotgan klasslar **User** klassidan murakkab bo'ladi. Ammo, ushbu misolda **DB**dagi ba'zi jadvallarga mos klasslarni avtomatik hosil qilishimiz mumkin.

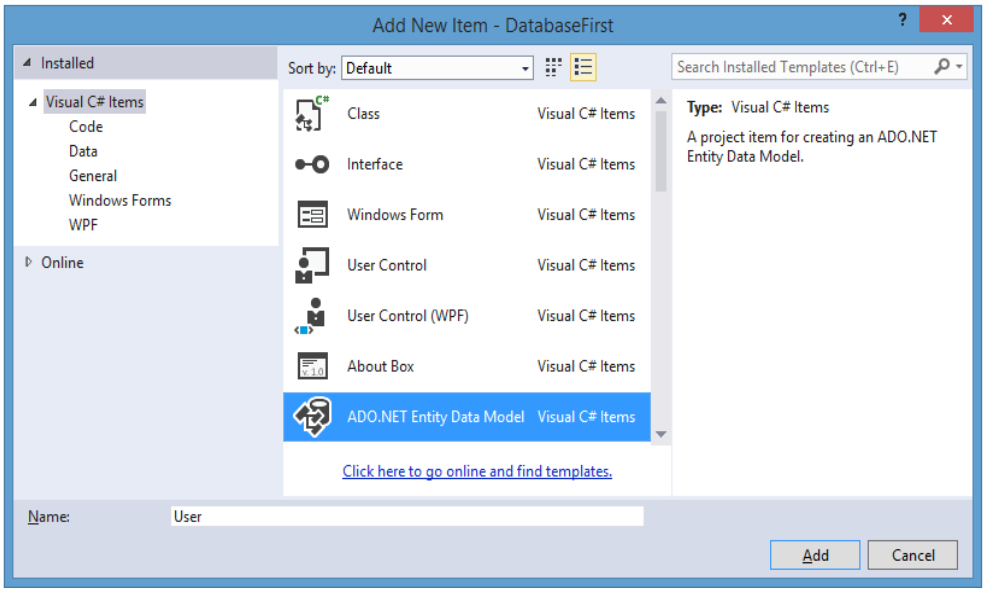
Database First

Database First yondashuvi **Entity Framework** birinchi bor yaratilgan vaqtda taqdim qilingan edi. Ushbu yondashuv ko'p jihatdan **Model First** ga o'xshash bo'lib, **DB** tayyor bo'lgan holda ishlatiladi.

Entity Framework muayyan **DB**dagi ma'lumotlar bilan ishlashi uchun tizimda mos provayder o'rnatilgan bo'lishi lozim. **Visual Studio** da **MS SQL Server** ob'ektlari bilan ishlash uchun barcha zaruriy infratuzilmalar o'rnatilgan. **MySQL**, **Oracle** va boshqa turdagi **DB**lar uchun mos provayderlarni o'rnatish lozim. Eng keng tarqalgan **DB** provayderlarini **ADO.NET** Data Providers dan olish mumkin.

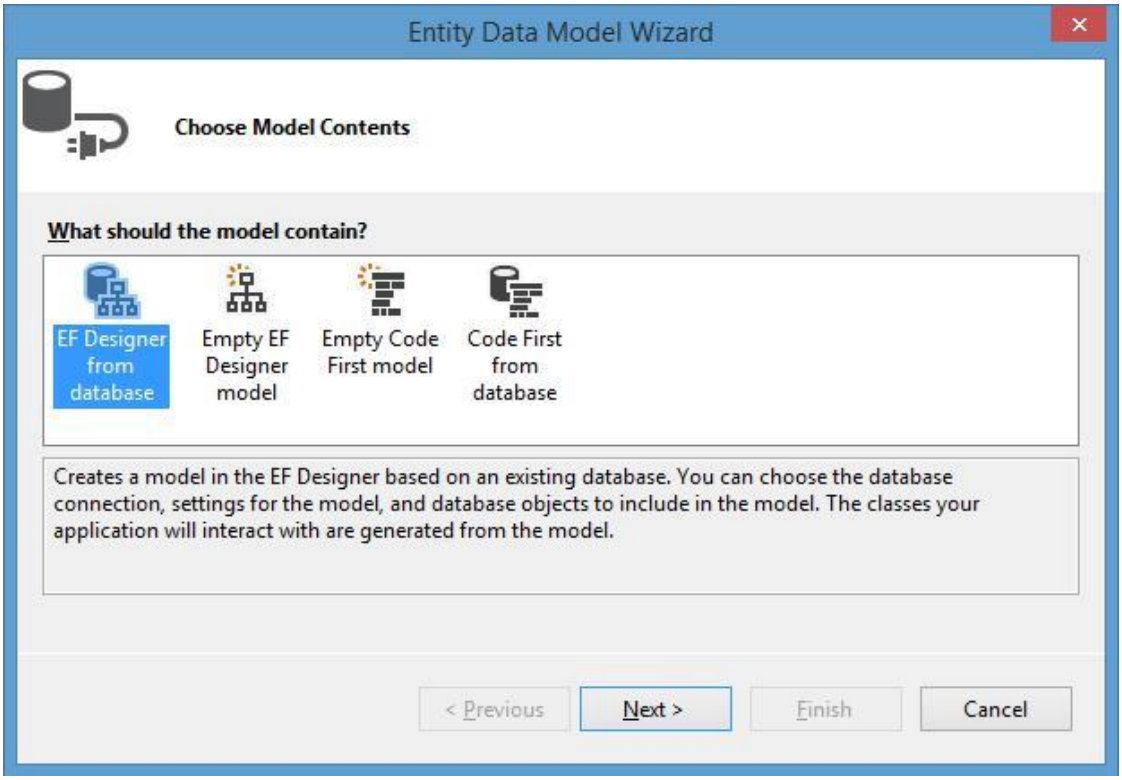
Yangi **Console Application** turidagi loyihani yaratamiz. Uning funksionali avvalgi loyihalarga o'xshash tarzda amalga oshiriladi. Faqat **Entity Framework** ga yondashuv boshqacha tarzda tashkil qilinadi. Ushbu yondashuv asosida dastur yaratilayotganda **DB** da zarur ob'ektlar mavjud bo'lishi shart.

Visual Studio dagi **Solution Explorer** oynasidan loyiha nomini tanlab, sichqonchani o'ng tugmasini bosamiz. Hosil qilingan menyudan **Add --> New Item** qismni tanlaymiz. So'ngra hosil qilingan muloqot oynasidan **ADO.NET Entity Data Model** ni tanlaymiz. Yangi komponentga **User** nomini beramiz:



27-Rasn

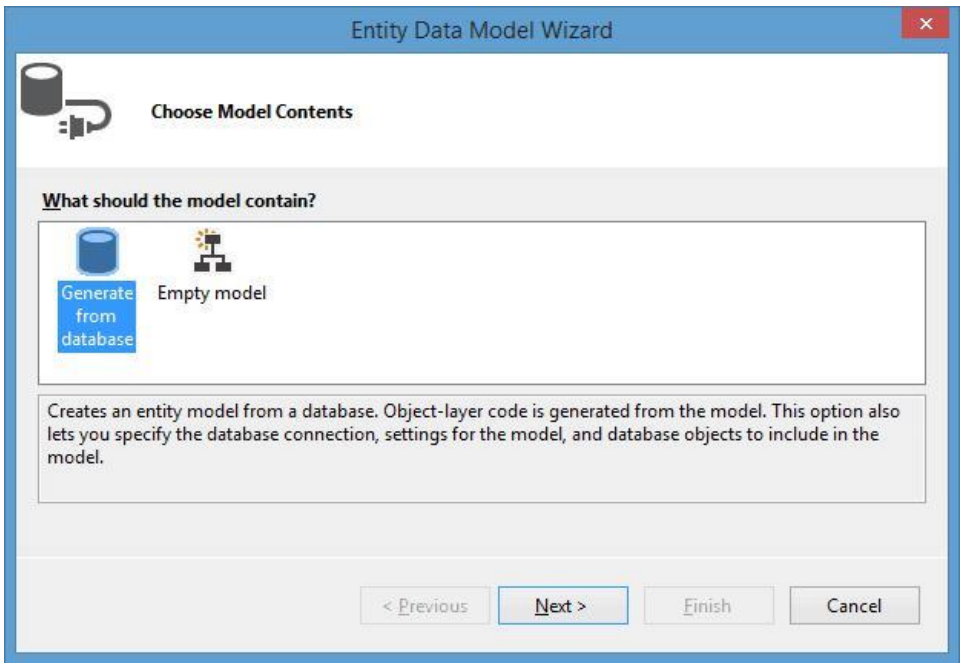
So’ngra bizga model hosil qilish uchun oyna taqdim etiladi. Agar siz **Visual Studio 2013ning SP2, SP3** o’rnatilgan varianti bilan ishlayotgan bo’lsangiz, quyidagi muloqot oynasi taqdim etiladi:



28-Rasm

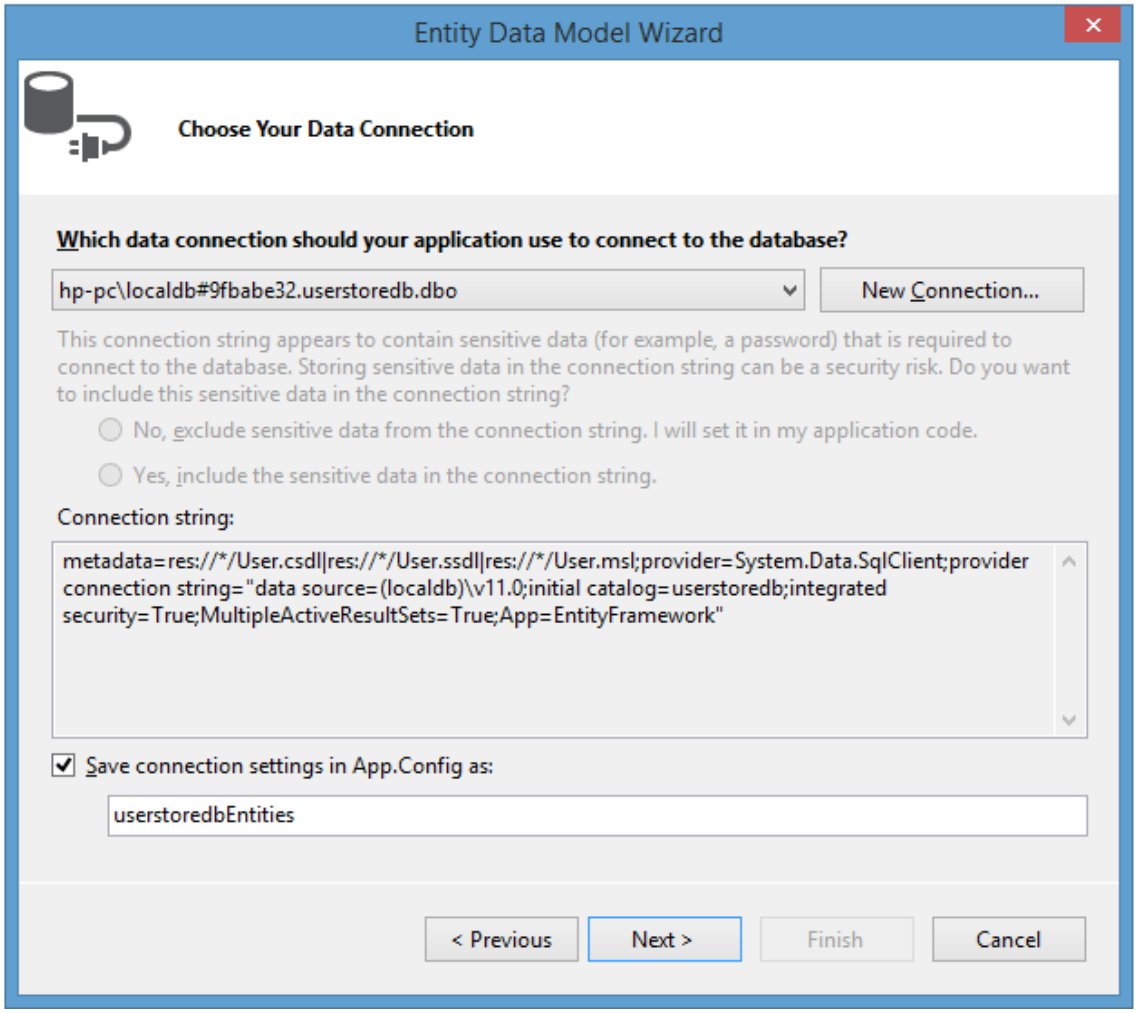
Ushbu muloqot oynasidan **EF Designer from database** qismini tanlaymiz.

Agar sizda **Visual Studio 2013** zaruriy paketlari o’rnatilmagan bo’lsa, u quyidagi ko’rinishga ega bo’ladi:



29-Rasm

Ushbu holda **Generate from database** (Создание модели по имеющейся базе данных)qismni tanlash lozim. Natijada modelni hosil qilishning keyingi qadami uchun oyna taqdim etiladi. Ushbu muloqot oynasida **DB**ga ulanishni ko’rsatish lozim.

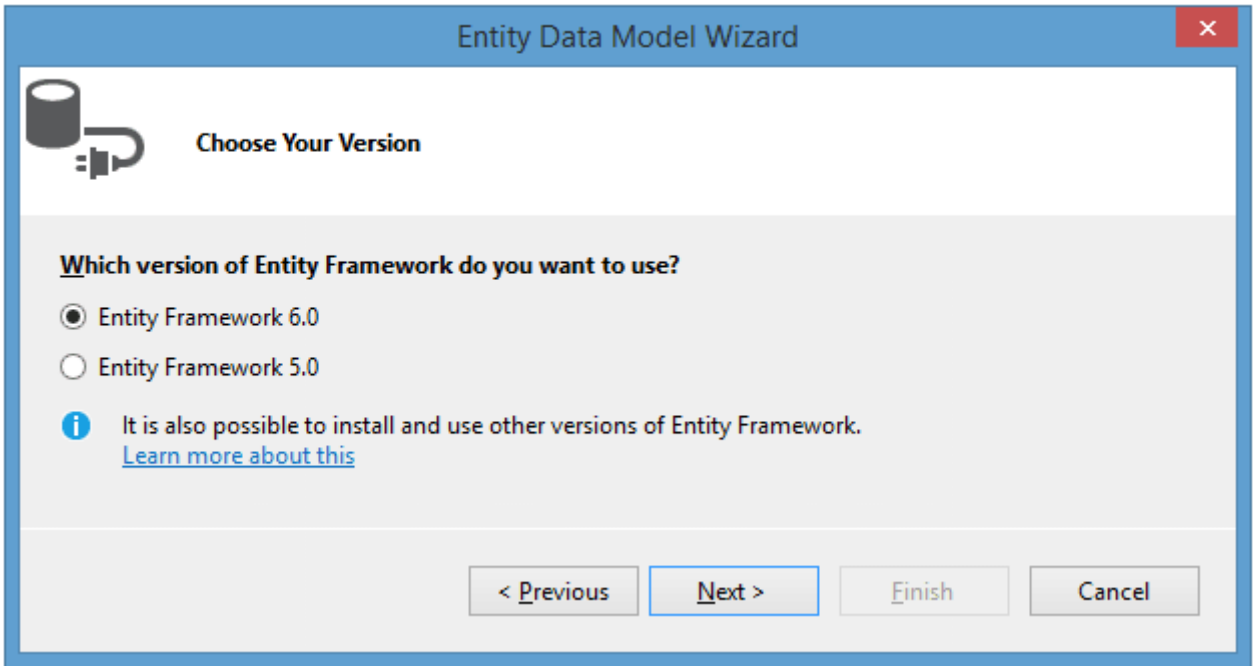


30-Rasm

Ro'yxatdan zarur bo’lgan ulanishni tanlaymiz. Agar ro'yxatda bizga kerakli ulanish mavjud bo’lmasa, **New Connection** tugmasini bosib, yangi ulanishni shakllantiramiz.

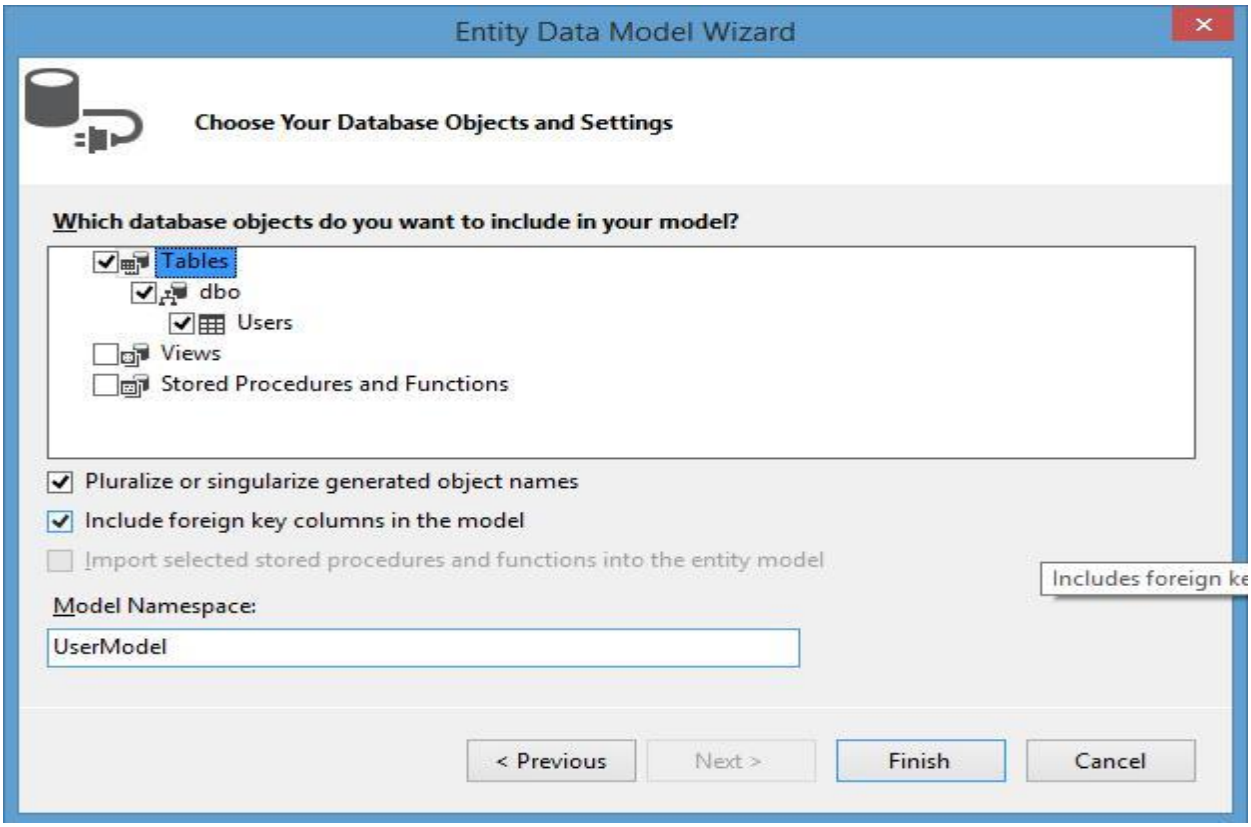
Shuningdek, loyihamiz uchun zarur bo’lgan ma’lumotlar kontekstini tanlash lozim. Bizning misolda ma’lumotlar konteksti sifatida **userstoredbEntities** kelgan. Bu qiymatni qoldirish yoki o’zgartirish mumkin.

Ulanish tanlanganidan so’ng, keyingi qadamda o'tish mumkin. Bizda **Visual Studio 2013** da zarur paketlar o’rnatilmagan bo’lsa, **Entity Framework** versiyani tanlash taklif qilinadi. Oltinchi versiyasini tanlaymiz:



31-Rasm

Visual Studio 2013 SP2, SP3 versiyalarida **EF** 6 ishlatiladi. Shuning uchun ushbu qadam tashlab o'tiladi. So'ngra **Visual Studio DB** haqidagi barcha ma'lumotlarni oladi:

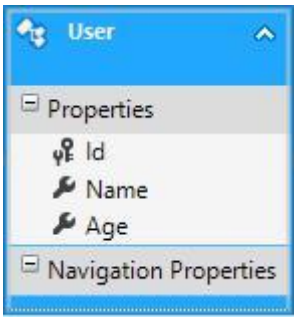


32-Rasm

So’ngra **Tables** qismini tanlaymiz. Natijada **DB**da mavjud jadvallar ro'yxati keltiriladi. Bizning misolda u faqat bitta **Users** jadvalidan iborat. **Tables** tugunidagi barcha qismlarni tanlaymiz.

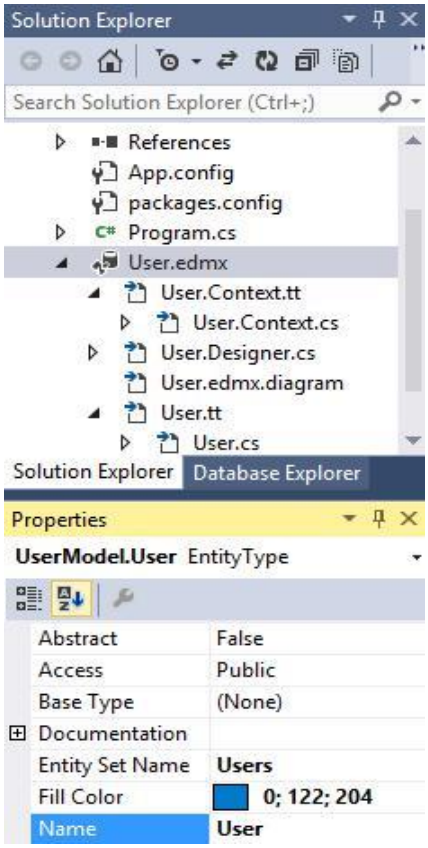
Model Namespace maydonida modelga mos nomni yozamiz va **Finish** tugmasini bosamiz. Shundan so’ng **Entity Framework DB** asosida modelni generatsiya qiladi va uni loyihaga qo’shib qo’yadi.

Visiual Studio bizga model sxemasini ko’rsatadi. Bizning misolda **DB**da faqat bitta **Users** jadvali mavjudligi sababli, sxemada faqat bitta **User** ob’ekti namoyish qilinadi.



33-Rasm

Ob’ekt tanlanganidan so’ng, **Visual Studio** ning yuqori o’ng qismida ushbu ob’ektga mos xususiyatlarni ko’rishimiz mumkin:



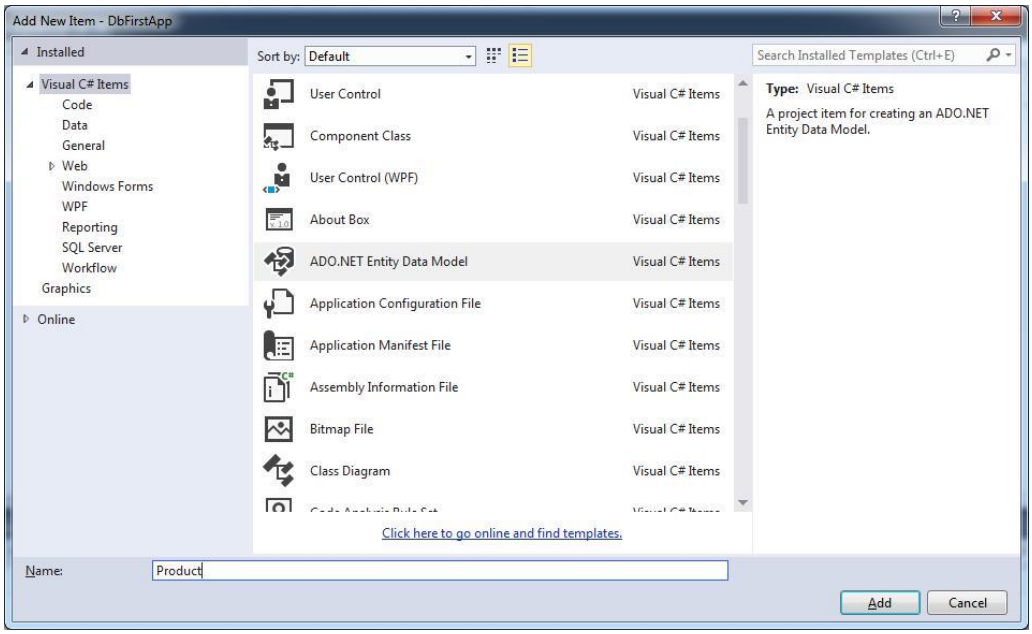
34-Rasm

Xususiyatlar oynasidagi **Name** xususiyati ob'ekt namoyish qilinadigan klassni ko'rsatadi. **Entity Set Name** xususiyati orqali ob'ektlar ro'yxati nomi (ma'lumotlar kontekstidagi **DbSet** xususiyati) keltirilgan. Bizning misolda ushbu qiymat **Users** ga teng.

Endi oddiy dastur yaratib, ma'lumotlar ustida ba'zi amallarni bajaramiz:

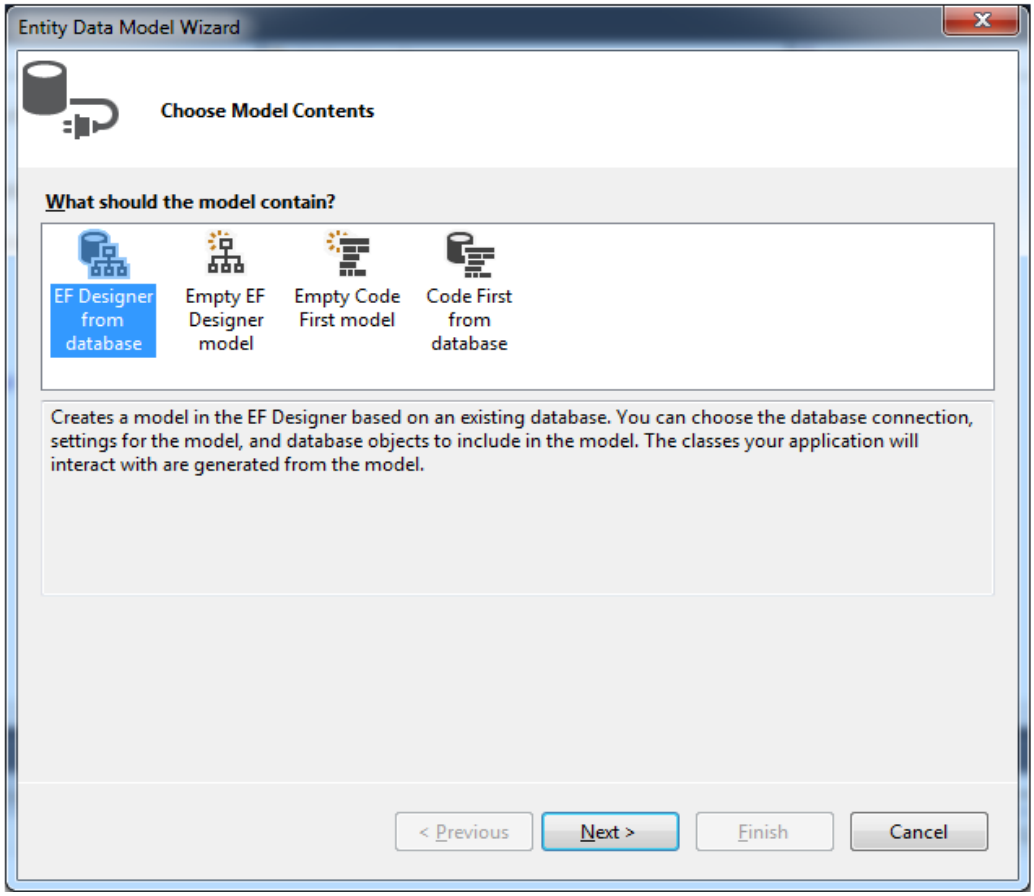
```
using (userstoredbEntities db = new userstoredbEntities())
{
var users = db.Users;

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

Yuqoridagi texnologiya asosida yana bir misolni ko'rib chiqamiz:

1. **VS 2013** da yangi (**DbFirstApp**) konsol loyihasini hosil qiling.
2. Loyihaga yaqin **ADO.NET Entity Data Model** elementini qo'shing.
3. 

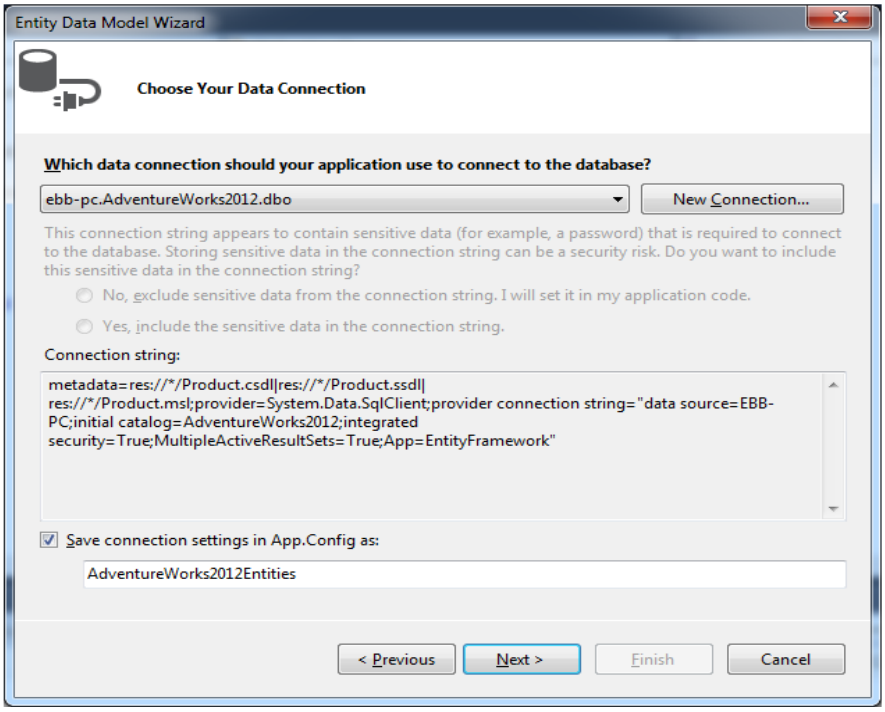
35-Rasm

4. Muloqot oynasidan **EF Designer from database** qismini tanlang.



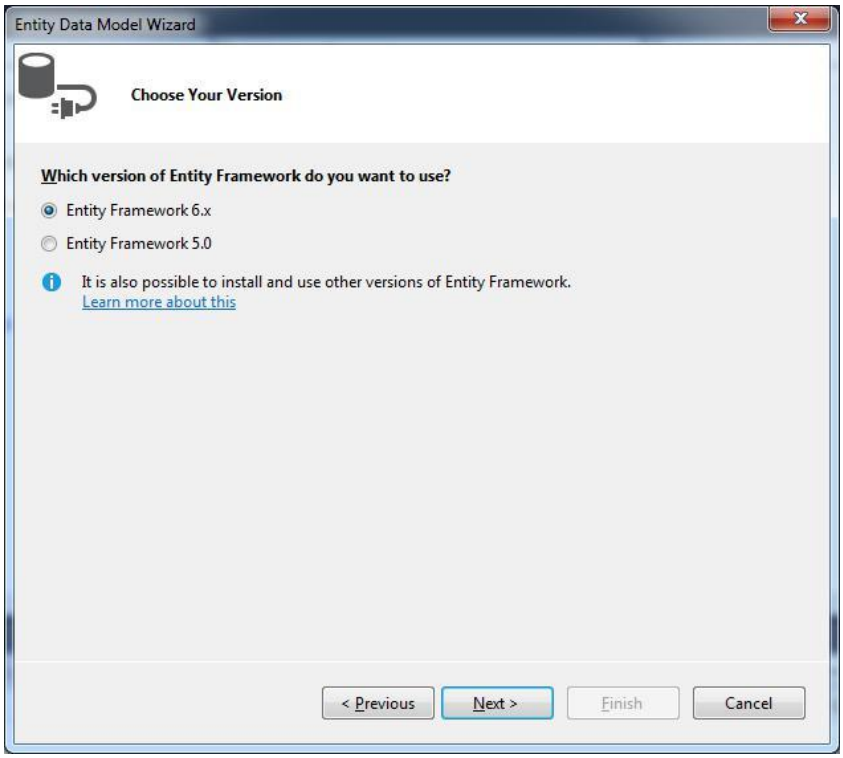
36-Rasm

5. Ro’yxatdan zarur bo’lgan ulanishni tanlang. Agar ro’yxatda bizga kerakli ulanish mavjud bo’lmasa, **New Connection** tugmasini bosib, yangi ulanishni shakllantiring . Bizning misolda AdventureWorks2012 bazasi bilan aloqa o’rnatilishi zarur.



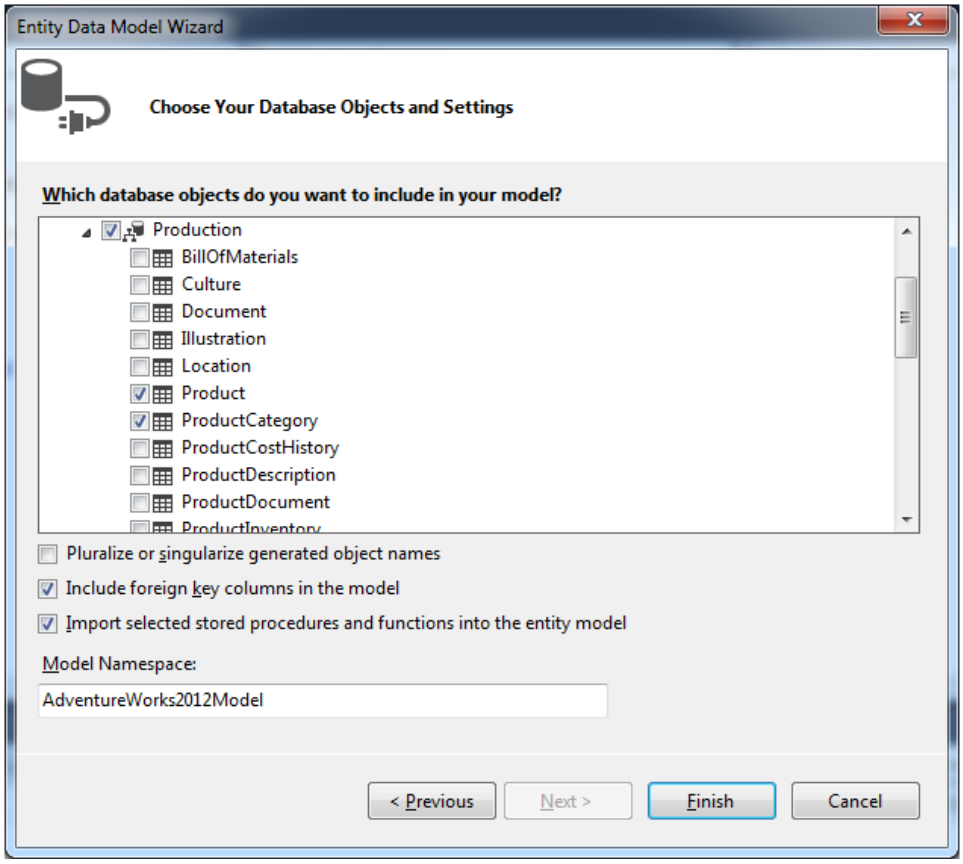
37-Rasm

6. Zarur bo'lgan **EF 6.x** versiyasini tanlang.



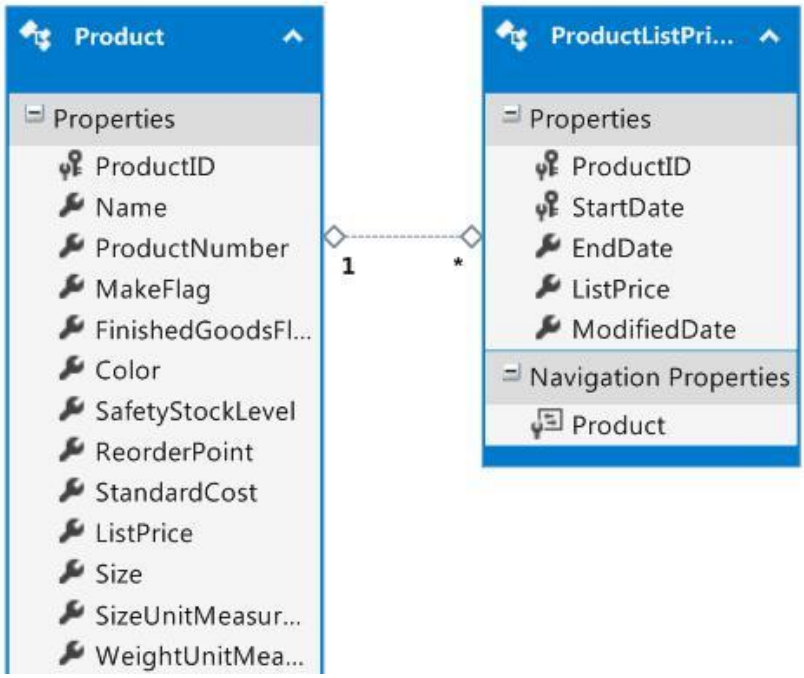
38-Rasm

7. Jadvallar ro'yxatini Production.Product va Production.ProductListPriceHistory jadvallarini tanlang.



39-Rasm

8. Natijada loyihada quyidagi diagramma shakllantiriladi:



40-Rasm

1. Endi ushbu modellar asosida **DB**dan olingan ma’lumotlar ustida quyidagi amallarni bajaruvchi dastur tuzing:

a. Eng qimmat 50ovar narxi va rangi qizil bo’lgan tovarlar summasi:

```
using System;

using System.Linq;

namespace DbFirstApp
{
class Program
{
static void Main(string[] args)
{
using (ProductContext db = new ProductContext())
{
decimal minPrice = db.Products.Max(p => p.ListPrice); Console.WriteLine(“Eng qimmat 50ovar narxi: {0}”, minPrice);

            decimal sum = db.Products.Where(p => p.Color.Contains(“Red”)).Sum(p => p.ListPrice);

Console.WriteLine(“Rangi qizil bulgan tovarlar summasi: {0}”, sum);
}
Console.ReadLine();
}
}
}
```

b. Tovarlarning ranglari bo’yicha nechtadan mavjudligini aniqlash:

```
using System;
using System.Linq;

namespace DbFirstApp
{
class Program
{
static void Main(string[] args)
{
using (ProductContext db = new ProductContext())
{
var groups = from p in db.Products
group p by p.Color into g

            select new { Color = g.Key, Count = g.Count() }; foreach (var p in groups)
```

```
Console.WriteLine("{0} : {1}", p.Color, p.Count);
}
Console.ReadLine();
}
}
}
```

c. Rangi qora va narxi 15000 dan qimmat bo'lgan tovarlar ro'yxati aniqlansin:

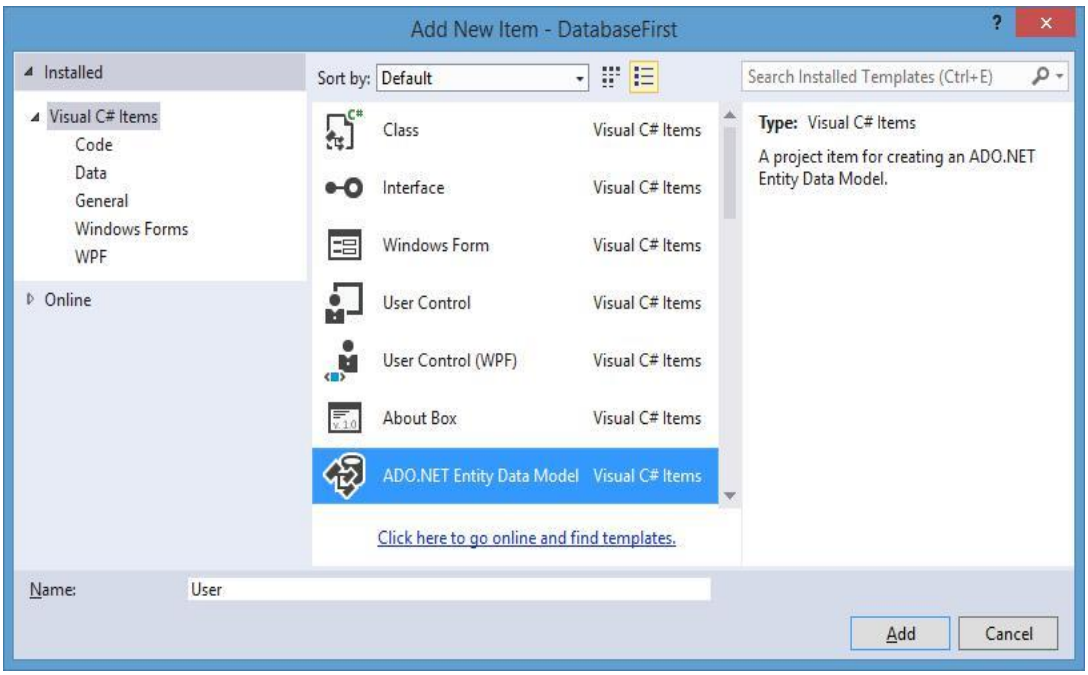
```
using System;
using System.Linq;

namespace DbFirstApp
{
class Program
{
static void Main(string[] args)

{
using (ProductContext db = new ProductContext())
{
var products = db.Products.Where(p => p.ListPrice> 15000)
.Union(db.Products.Where(p => p.Color.Contains("Black")));

foreach (Product p in products)

                    Console.WriteLine("{0} : {1}", p.Name, p.ListPrice);
}
}
Console.ReadLine();
}
}

}
```

Model First

Model First yondashuvi orqali **Entity Framework** bilan imkoniyati mavjud. Ushbu yondashuv asosida avvalo model hosil qilinadi. So'ngra u asosida **DB** yaratiladi.

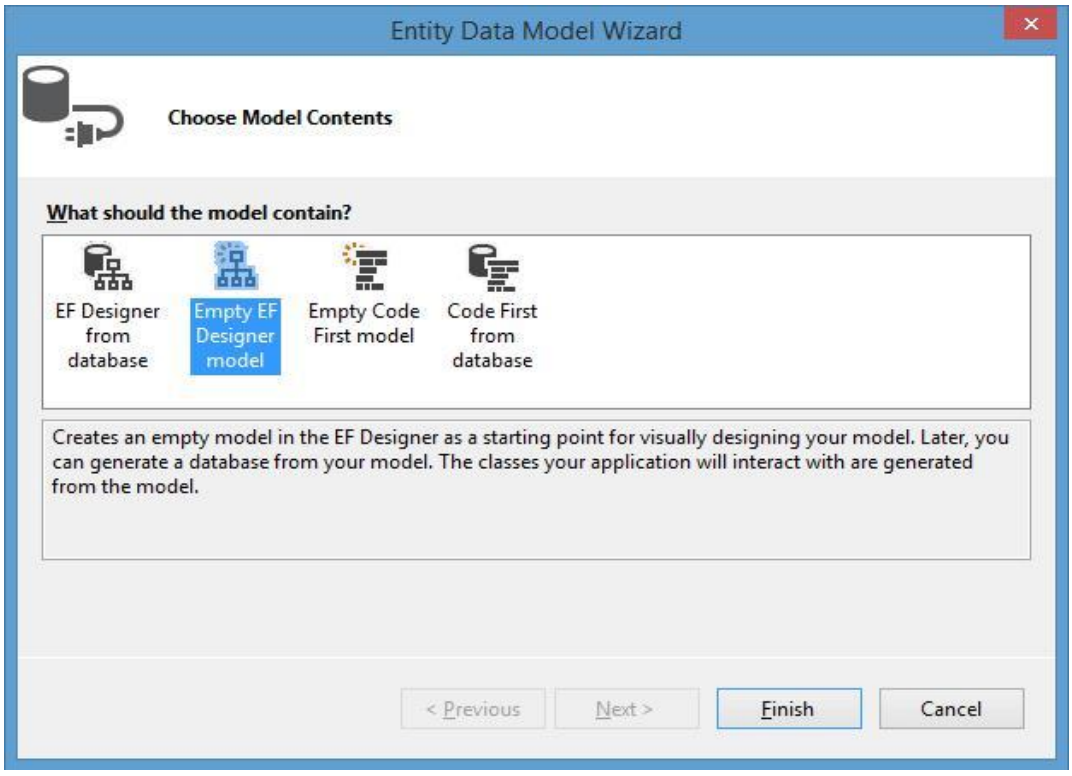
Buning uchun avvalo **Console Application** yangi loyihasini hosil yaratamiz . So'ngra ushbu loyihaga yangi elementni qo'shib qo'yamiz. **Solution Explorer** dagi loyiha nomini ustiga sichqonchaning o'ng tugmasini bosib, **Add -> New Item**

qismini tanlaymiz.Keyingi qadamda hosil qilingan ro'yxatdan **ADO.NET Entity Data Model** ni tanlaymiz.



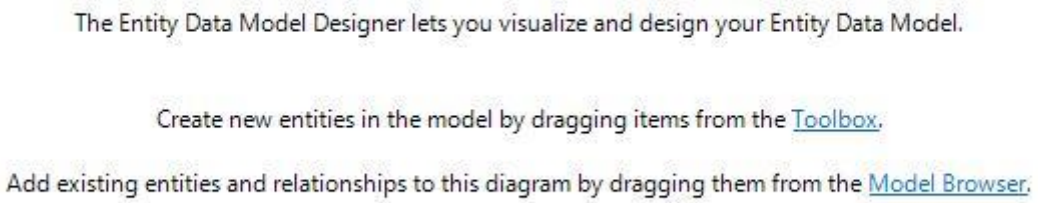
41-Rasm

Ushbu modelimiz insonni tavsiflashgani uchun unga **User** nomini beramiz va **OK** tumasini bosamiz. Agar bizda **Visual Studio** ning **SP2, SP3** paketlari o'rnatilgan bo'lsa, u quyidagi ko'rinishga ega:

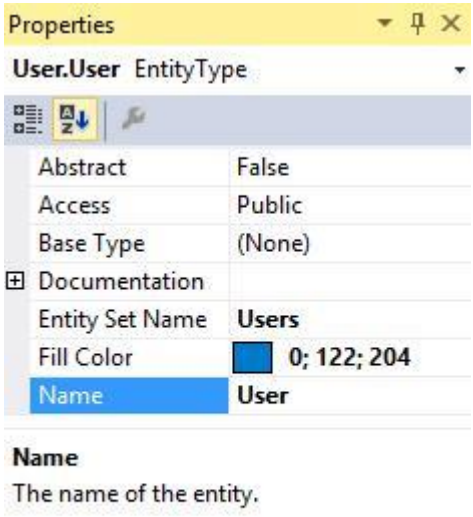


42-Rasm

Ushbu oyna orqali to’rtta variantdagi modelni hosil qilish mumkin. Ushbu roýhatdan **Empty EF Designer Model** ni tanlashimiz va **Finish** tugmasini bosishimiz lozim. Natijada bizda modelni bo’sh hosil qilish oynasi taqdim etiladi:



Ushbu oynaga **Toolbox** (**Панель Инструментов**) oynasidan **Entity** elementini olib joylashtiramiz. Endi bizda hosil qilishi lozim bo’lgan modelning sxemasi mavjud. Unda boshlang‘içh xolda faqat bitta **Id** maydoni mavjud. Birinchi navbatda ob’ektga yangi nom beramiz. Boshlang‘ich holda u **Entity1** ga teng. Sxemani tanlab ekranning o’ng qismida joylashgan loyihaning xususiyatlar qismiga o'tamiz.



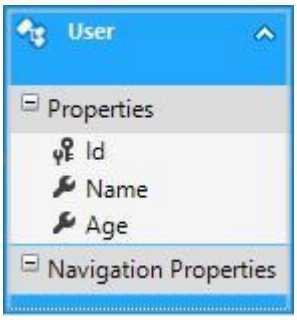
43-Rasm

Ushbu oynada **Name** xususiyati qiymatini **User** ga o’zgartiramiz. Bu qiymat biz yaratayotgan model (ob’ekt) nomi hisoblanadi. Keyingi qadamlarda ushbu modelga mos bir nechta xususiyatlarni shakllantiramiz. Bizning misoldagi **(User)** modelimizda inson FIO va yoshi mavjud. Sxemani tanlab, sichqonchaning o’ng

tugmasini bosamiz. Hosil qilingan menyudan **Add New -> Scalar Property** ni tanlaymiz. Shundan so'ng modelda yangi xususiyat hosil qilinadi.

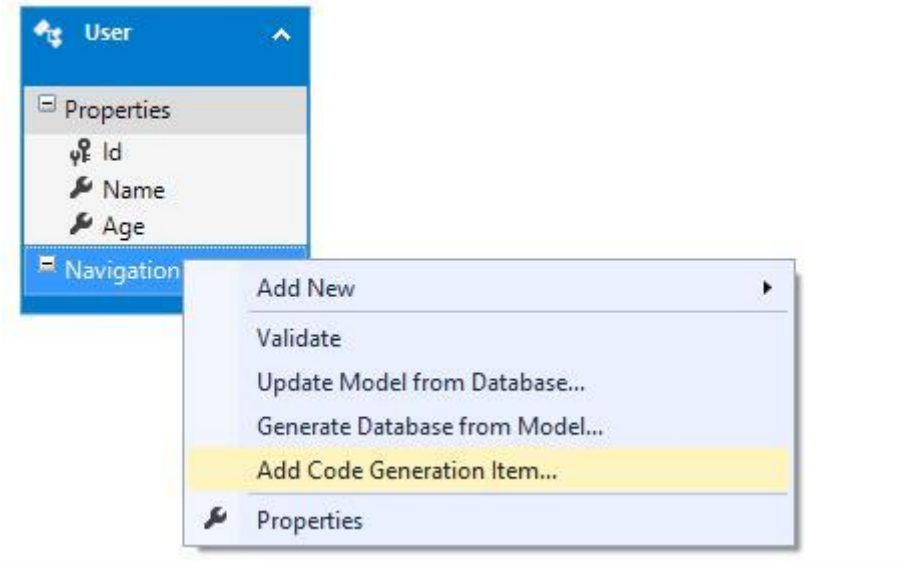
Scalar Property orqali modelda oddiy **(int, float, string)** tipga mansub xususiyatlarni hosil qilish mumkin. Ikkita **Name** va **Age** xususiyatlarini shakllantiramiz. Ularning ikkalasi ham boshlang'ich holda **string** tipiga mansub bo'ladi. Ushbu xususiyatlarga mos tiplarni xususiyatlar oynasidan o'zgartirishimiz mumkin.

Barcha zaruriy amallarni bajarganimizdan so'ng, quyidagi rasmda keltirilgan ob'ekt hosil qilinishi lozim:



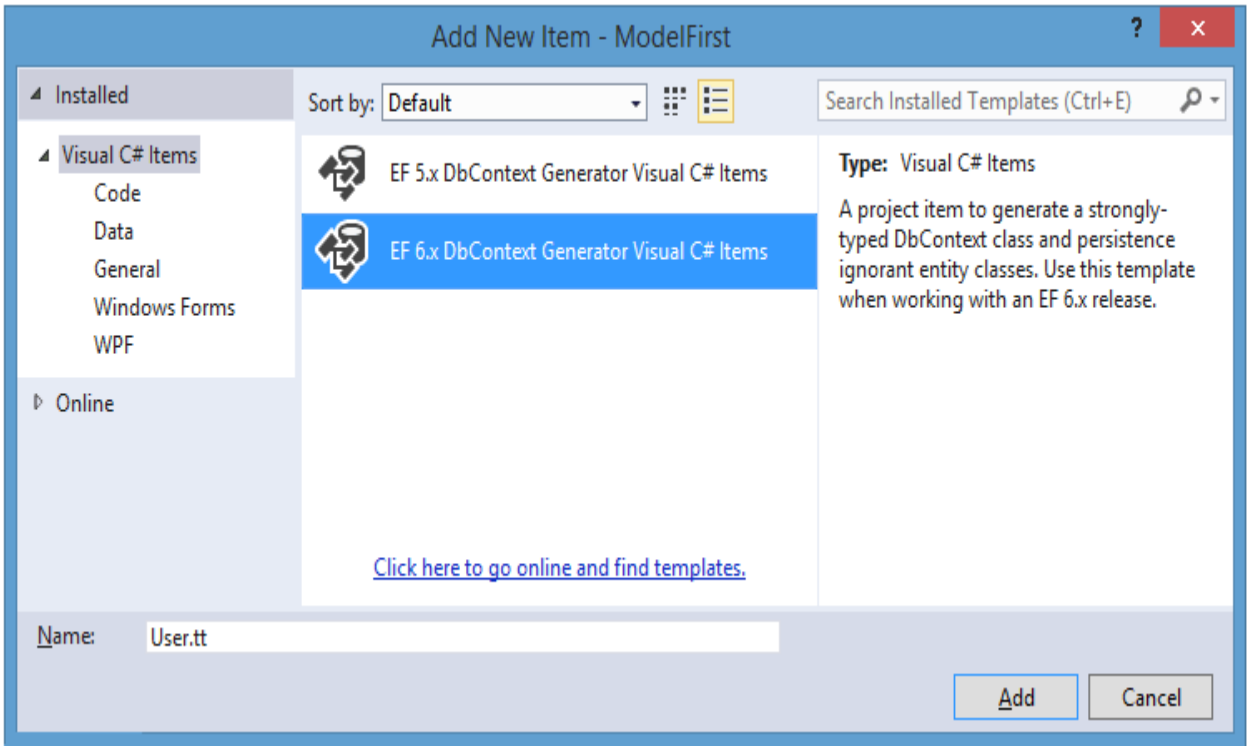
44-Rasm

Modelni shakllantirganimizdan so'ng uni loyihaga **Rebuild** opsiyasi yangilab qo'yamiz. Endi ushbu modelga mos tuzilmani **DB**da hosil qilish mumkin. Avvalo model kodini generatsiya qilamiz. Buning uchun model diagrammasida sichqonchaning o'ng tugmasini bosib, **Add Code Generation Item qismni tanlaymiz**.



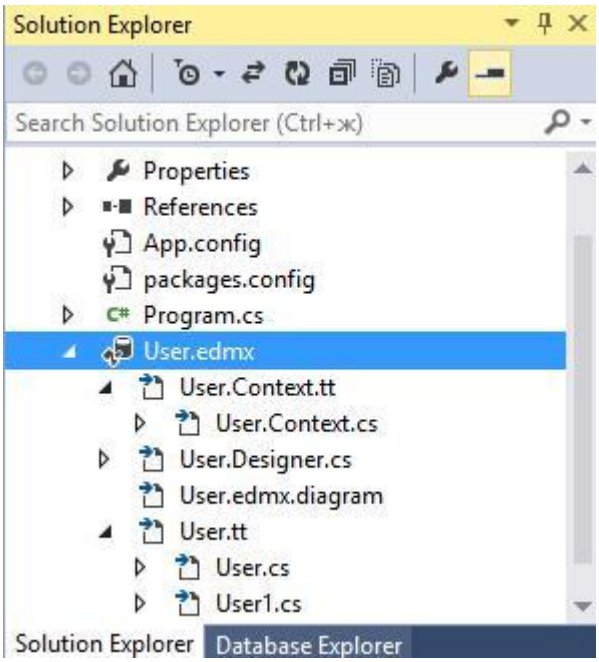
45-Rasm

Natijada **EF** versiyasini tanlash taklif qilinadi:



46-Rasm

Shundan so'ng loyiha tuzilmasida **User.tt** tuguni hosil bo'ladi. Ushbu tugunda model klassi **User.cs** joylashgan.

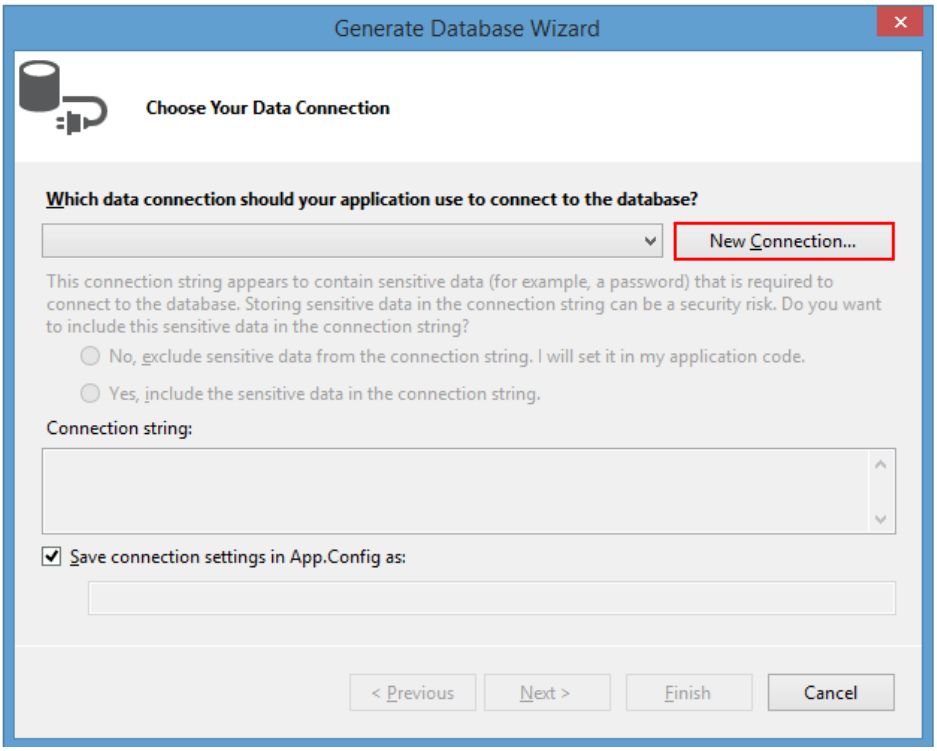


47-Rasm

Shuningdek, ushbu tugunda ma’lumotlar konteksti fayli **User.Context.cs** ham mavjud. U quyidagi kodga ega:

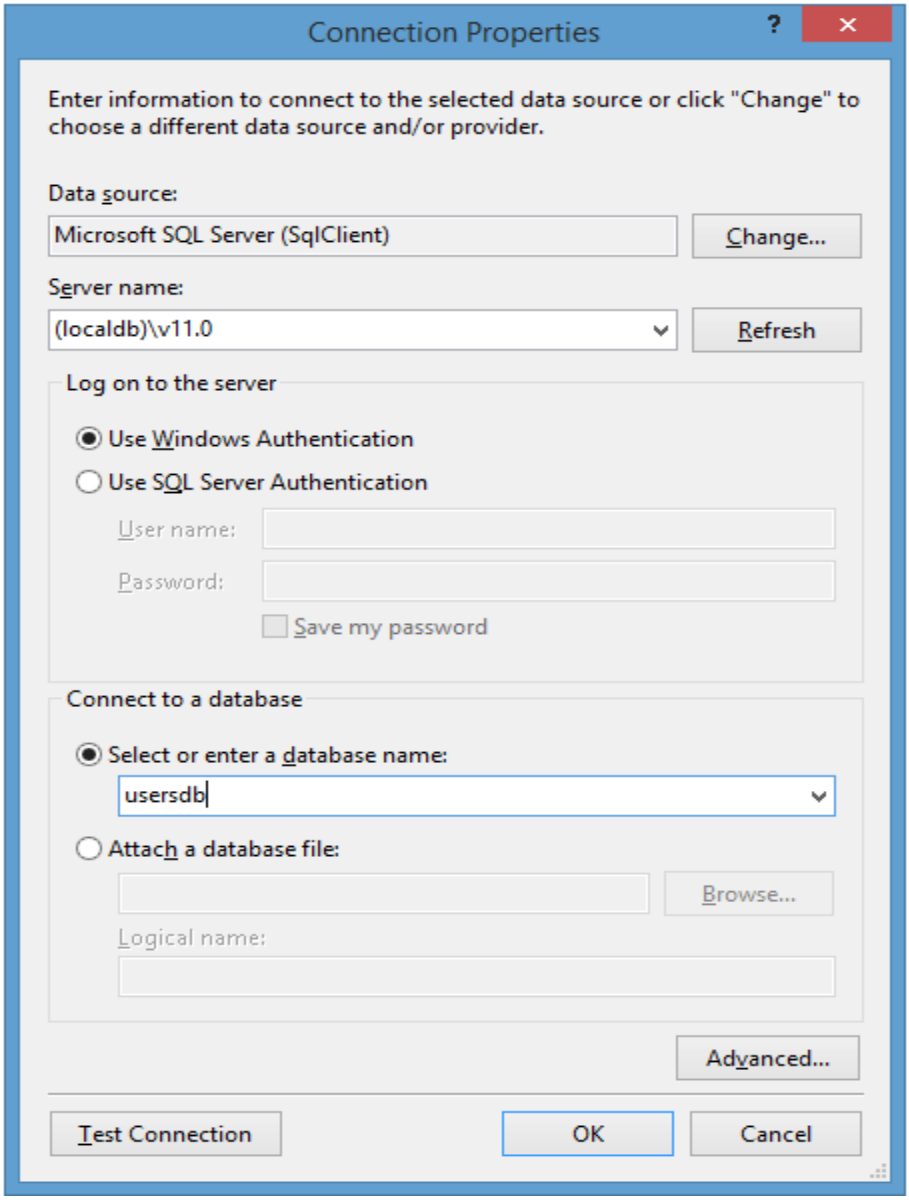
```
namespace ModelFirstApp
{
using System;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;

public partial class UserContainer : DbContext
{
public UserContainer()
:           base("name=UserContainer")
{
}
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
throw new UnintentionalCodeFirstException();
}

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

}
}
```

Endi ushbu modelga mos jadvallarni **DB**da hosil qilamiz. Buning uchun model diagrammasida sichqonchaning o’ng tugmasini bosib, **Generate Database from Model**(**Сгенерировать базу данных по модели**) ni tanlaymiz. Natijada bizga **DB**ga ulanishni hosil qilish interfeysi taqdim etiladi:



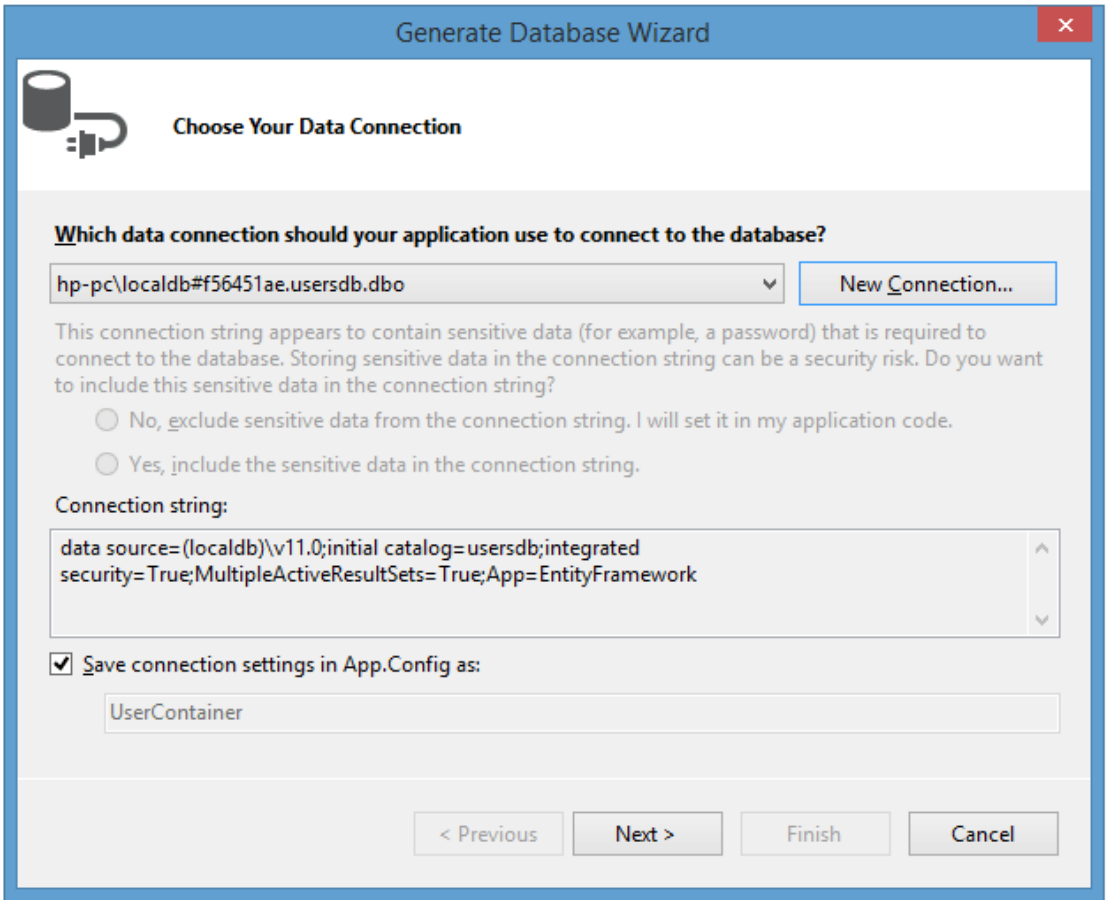
48-Rasm

Ushbu muloqot oynasidan **New Connection(Новое подключение)** qismni tanlaymiz. So’ngra bizga ulanishni hosil qilish va **DB**ni yaratish amalgi o'tiladi:



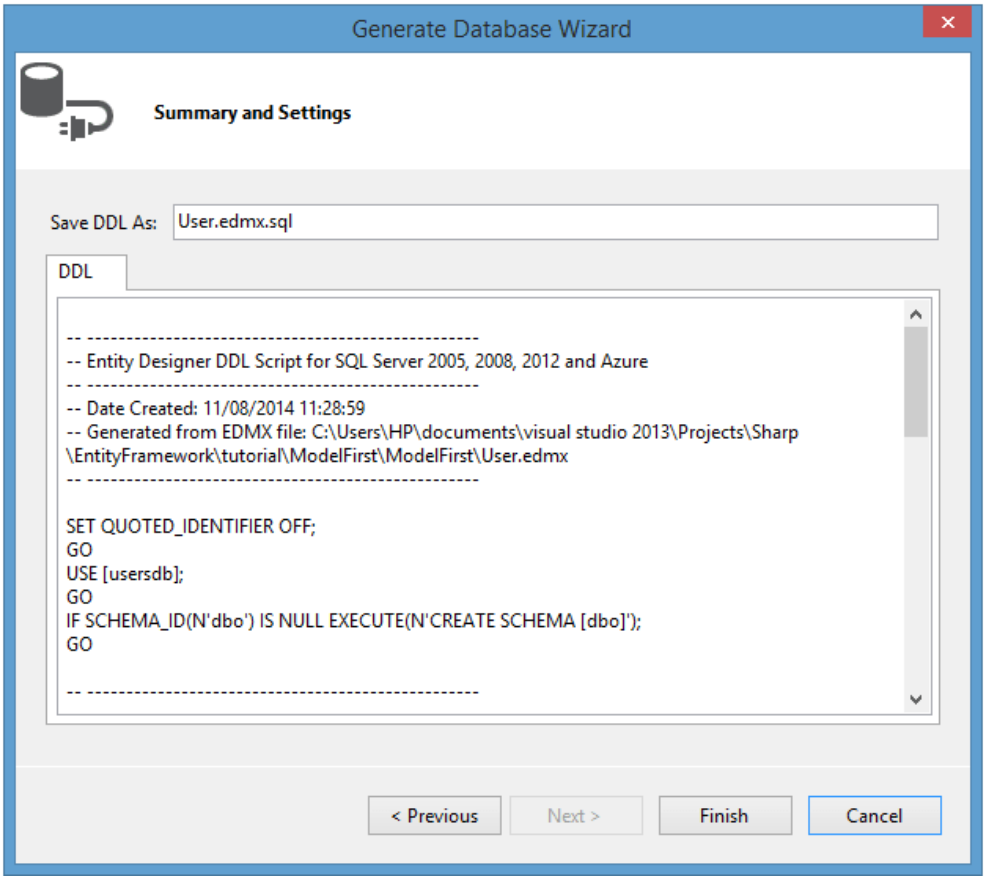
49-Rasm

Ushbu muloqot oynasida server nomini va hosil qilinayotgan **DB** kiritish lozim. **DB** nomi sifatida **usersdb** kiritamiz. Server nomi sifatida **usersdb** ni kiritamiz. So'ngra OK tugmasini bosamiz. Natijada **Visual Studio** modelga ulanish satri va **DB** nomini quyidagicha shakllantiradi:



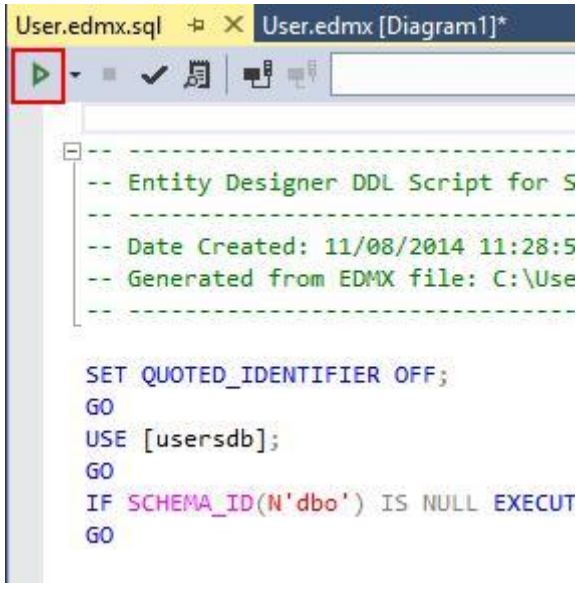
50-Rasm

Shundan so’ng **DB** skripti hosil qilinadi:



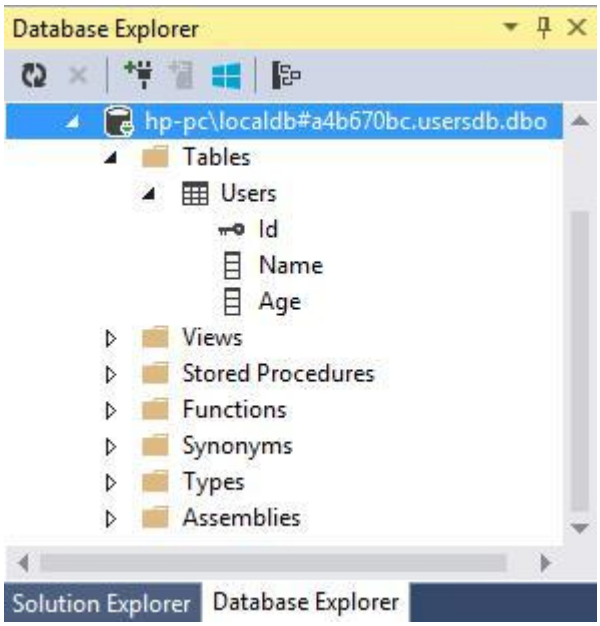
51-Rasm

So’ngra **Finish** tugmasini bosamiz. **Visual Studio**da avtomatik tarzda **User.edmx.sql** skript fayli ochiladi. Oxirgi qadamda biz ushbu skriptni ishga tushirishimiz lozim. Buning uchun yuqori chap qismdagi **Execute (Выполнить)** tugmasini bosish lozim:



52-Rasm

Natijada **Visual Studio** oynasida yuqoridagi amallarning omadli yoki omadsiz bajarilganligi haqidagi xabar chiqariladi. **View->Other Windows** dagi **Database Explorer** oynasi ochib, hozir hosil qilingan **DB**ni ko'rishimiz mumkin:



53-Rasm

Yuqorida keltirilgan amallar modellar va **DB**ni hosil qilish uchun zarur. Oxirgi qadamda ushbu hosil qilingan **DB** bilan ba'zi amallarni bajaramiz.

```
static void Main(string[] args)
{
using (UserContainer db = new UserContainer())
{

//ma'lumotlar qo'shish

db.Users.Add(new User { Name = "Tom", Age = 45 }); db.Users.Add(new User { Name = "John", Age = 22 }); db.SaveChanges();

//              ma'lumotlarni olish var users = db.Users; foreach (User u in users)
Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Age);
}
Console.Read();}
```

Nazorat savollari:

1. Entity Framework ma'lumotlar bazasi komponentlarini o'rnatish bilan bog'liq qadamlarni tasvirlab bering.

2. Entity Framework dasturida ma’lumotlar bazasini yaratish va ular bilan ishlash uchun qaysi dasturlash tilidan foydalanish mumkin?

3. Code First obyektlariga nomlar qanday beriladi?

4. Ma'lumotlar bazasiga kalitlarni o'rnatish jarayonini aytib bering.

5. Code First ma'lumotlar bazasida avtomatik sinflar qanday yaratiladi?

3. ENTITY FRAMEWORK ASOSLARI
Reja

1. **Entity Framework Power Tools bilan ishlash.**
2. **Entity Framework Database First asbobi bilan ishlash.**
3. **Birinchi model.**
4. **Nazorat savollari.**

Ma’lumotlar ustida asosiy amallar

Ko’pgina ma’lumotlar ustida **CRUD-amallari (Create, Read, Update, Delete)** bajariladi. Ya’ni ma’lumotlarni olish, yozish, o’zgshartirish va o’chirish amallari. **Entity Framework** orqali ushbu amallarni tezda amalga oshirish mumkin.

Yuqoridagi amallarni bajaruvchi loyihani shakllantiramiz. **Windows Forms** tipiga mos loyihani yaratamiz. Yangi loyihada futbolchilar bazasi ustida ish bajarilsin. Yangi loyihada **DB** bilan ishlash uchun **Code First** yondashuvidan foydalanamiz.

Avvalo loyihaga futbolchilarni ifodalovchi klassni qo’shib qo’yamiz:

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

Ushbu klassda futbolchilar to’rtta xususiyat orqali aniqlangan: **Id, FIO,** maydondagi pozitsiyasi va yoshi. Shuningdek, **NuGet** orqali **Entity Framework** paketini ham loyihada o’rnatish va ma’lumotlar kontekstini hosil qilish lozim.

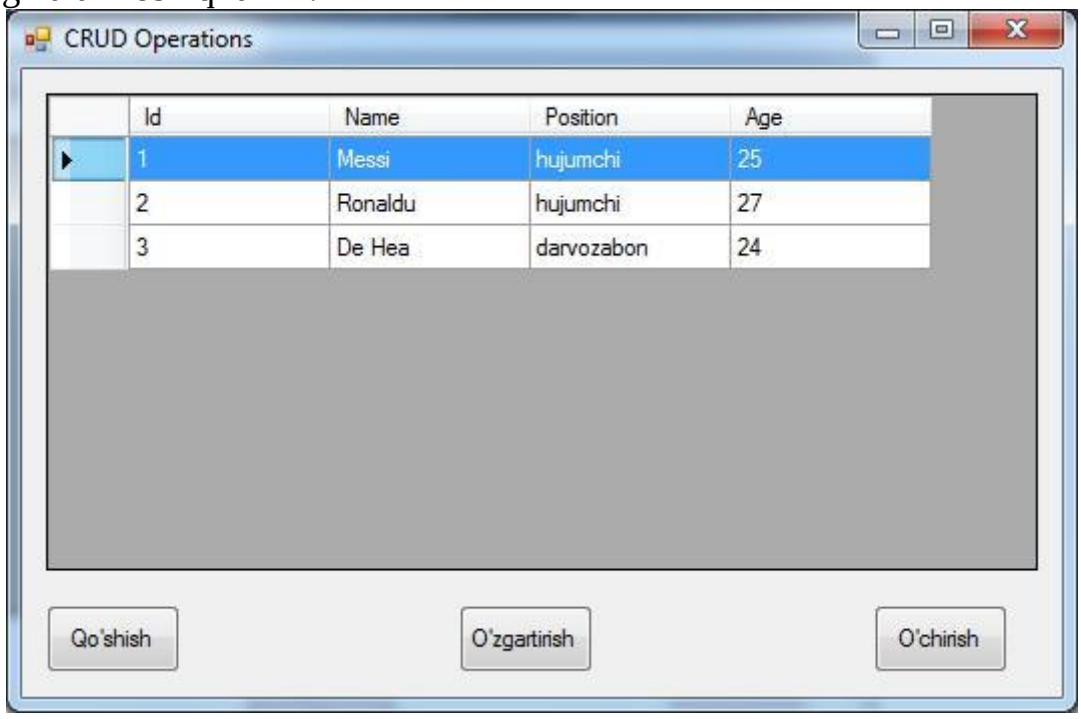
```
using System.Data.Entity;
namespace NewModelFirstApp
{

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

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

Shuningdek, loyihadagi **App.config** faylida ulanish satri hosil qilish lozim:

```
<connectionStrings>

<add name="DefaultConnection" connectionString="data source=.;initial
catalog=Soccer;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

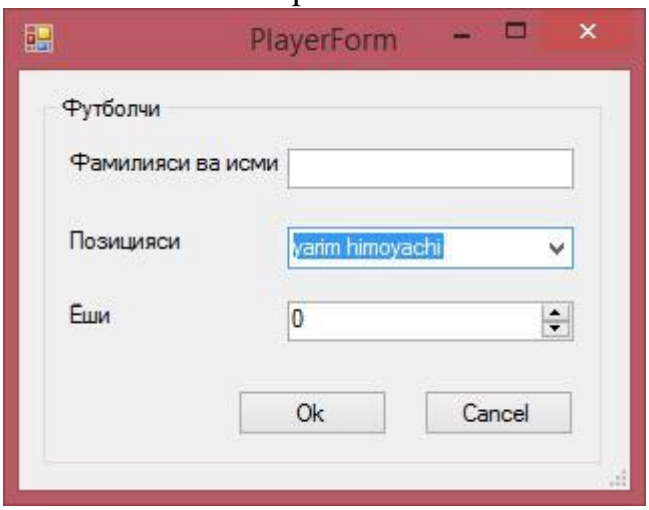
Endi loyihaning vizual qismini shakllantiramiz. Bizning loyihada **Form1** formasi mavjud. Ushbu formaga **DataGridView** elementini qo'shib qo'yamiz. Ushbu element orqali **DB**dagi zarur ma'lumotlar namoyish qilinadi. Shuningdek, ma'lumotlar ustida amal bajarish uchun qo'shish, o'zgartirish va o'chirish amallariga mos tugmalar hosil qilamiz:



54-Rasm

DataGridView elementning **AllowUserToAddRows** xususiyatiga **False** qiymatini, **SelectionMode** xususiyatiga **FullRowSelect** qiymatini beramiz.

Ushbu formamiz asosiy forma bo'lganligi sababli, **DB**ga yangi yozuv qo'shish, o'zgartirish va o'chirish amallari uchun alohida formalarni shakllantirish lozim. Loyihaga yangi **PlayerForm** formasini qo'shamiz:



55-Rasm

Ushbu formada futbolchi **FIO**ni kiritish uchun matnli maydon, so'ngra futbolchining maydondagi pozitsiyasini tanlash uchun **ComboBox** elementini qo'shib qo'yamiz. Ushbu elementda to'rtta pozitsiya mavjud. So'ngra, futbolchi yoshini tanlash uchun **NumericUpDown** ni shakllantiramiz.

Shuningdek, loyihaga ikkita tugmani qo'shib qo'yamiz. **"OK"** tugmasining **DialogResult** xususiyati uchun **OK** qiymatini, **"Отмена"** tugmasi **DialogResult** xususiyati uchun **Cancel** qiymatini beramiz.

Ushbu forma hech qanday kodni o'zida saqlamaydi. Endi **Form1** formamizga qaytamiz. Ushbu formada barcha mantiq joylashadi:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;

using System.Linq;
using System.Data.Entity;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace NewModelFirstApp
{
```

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

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

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

private void Form1_Load(object sender, EventArgs e)
{

}

// yangi yozuv qo’shish
private void button1_Click(object sender, EventArgs e)

{

PlayerForm plForm = new PlayerForm(); DialogResult result =
plForm.ShowDialog(this);

if (result == DialogResult.Cancel)
return;

Player player = new Player();

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

db.Players.Add(player);

db.SaveChanges();

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

// o’chirish

private void button3_Click(object sender, EventArgs e)
{
if (dataGridView1.SelectedRows.Count > 0)
```

```
{

int index = dataGridView1.SelectedRows[0].Index; int id = 0;

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

return;

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

MessageBox.Show("Yozuv o’chirildi");

}
}

// o’zgartirish
private void button2_Click(object sender, EventArgs e)
{
if (dataGridView1.SelectedRows.Count > 0)

{

int index = dataGridView1.SelectedRows[0].Index; int id = 0;

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

return;

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

PlayerForm plForm = new PlayerForm();

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

DialogResult result = plForm.ShowDialog(this);

if (result == DialogResult.Cancel)

return;

player.Age = (int)plForm.numericUpDown1.Value;
```

```
player.Name = plForm.textBox1.Text;
player.Position = plForm.comboBox1.SelectedItem.ToString();

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

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

}
}
}
}
```

Ushbu kodda ma'lumotlarni **DB**dan olish uchun **db.Players** ifodasidan foydalanilgan. Ammo **DataGridView** elementiga ushbu ob'ektni bog'lash va jadval ustida amalga oshirilgan o'zgartirishlarni namoyish qilish lozim. Shuning uchun avvalo **db.Players.Load()** metodi orqali ma'lumotlarni **DbContext** ga yuklaydi. So'ngra **dataGridView1.DataSource = db.Players.Local.ToBindingList()** jamoa orqali **db.Players** va **dataGridView1** ob'ektlari bog'lanadi.

Qo'shish

DBga yangi ob'ekt qo'shish uchun quyidagi koddan foydalaniladi:

```
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();
```

Muayyan klassga tegishli ob'ektni **DB**ga qo'shish uchun **DbSet** klassidagi **Add** metodidan foydalaniladi. Ushbu ob'ekt orqali formada mavjud maydonlarda kiritilgan qiymatlarga mos yozuv shakllantiraladi. **Add** metodi orqali yangi ob'ekning holati **Added** qiymati o'rnatiladi. Shuning uchun **db.SaveChanges()** metodi jadvalga yangi yozuv qo'shish uchun **INSERT** ifodasini generatsiya qiladi.

O’zgartirish

O’zgartirish ham yuqorida keltirilgan qo’shish amaliga o’xshaydi. Avvalo biz tanlangan ob’ektning xususiyatlari qiymatlarini forma maydonlariga uzatamiz. Ob’ektning o’zgartirilganligini belgilash uchun quyidagi ifodadan foydalanamiz:

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

Ushbu ifoda **db.SaveChanges()** metodiga joriy ob’ekt uchun **UPDATE SQL**-ifodasini shakllantirish lozimligini anglatadi.

O’chirish

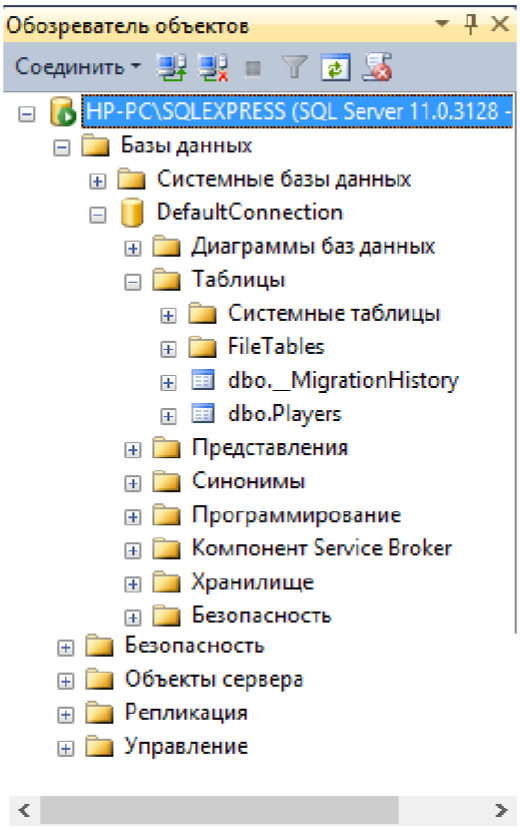
O’chirish amali juda sodda bo’lib, tanlangan **Id** qiymat orqali **DB**da zarur ob’ektni izlab topib, uni **db.Players.Remove(player)**ga uzatamiz. Ushbu metod ob’ekt statusini **Deleted** ga o’zgartiradi. Natijada **Entity Framework** da **db.SaveChanges()** metodi chaqirilishi natijasida **DELETE SQL**-ifodasi generatsiya qilinadi.

Ulanish satri

Yuqoridagi misolda ishlatilgan ma’lumotlar konteksti quyidagicha:

```
class SoccerContext : DbContext
{

public SoccerContext()
:           base("DefaultConnection")
{ }

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

Ushbu klass vorislangan klass konstruktoriga ulanish satri nomi parametr sifatida uzatilgan **(DefaultConnection)**. Ushbu holda **DB** bilan ishlashda **Code First** yondashuvidan foydalanilganligi sababli, agar bunday **DB** mavjud bo’lmama, u **MS SQL Server** da qayta **DefaultConnection** ga keltirilgan qiymatga mos hosil qilingan. **DB** hosil qilingandan so’ng barcha so’rovlar ushbu **DB**da bajariladi. Biz **MS SQL Server** ga **Visual Studio** yoki **SQL Server Management Studio** muhitidan foydalangan holda ulanishimiz mumkin.



56-Rasm

Ammo real dasturda ulanishga mos **DB** nomi yoki manzilini dinamik tarzda o'zgartirishga to'g'ri keladi. Ushbu holda ulanish satridan foydalanish lozim. **Visual Studio** da loyihaga konfiguratsiya faylini biriktirib qo'yadi. Oddiy loyihalarda (**Windows Forms**, **konsolli**) konfiguratsiya fayli sifatida **App.config**, veb-dasturlar loyihalarida (**ASP.NET**) ushbu fayl sifatida **Web.config** ishlatiladi. Loyiha turidan qat'iy nazar konfiguratsiya fayli muayyan tuzilma va elementlardan iborat. Ular orasida **connectionStrings** qismi mavjud bo'lib, u ulanish satrini ifodalaydi.

Biz **MS SQL Server** da joylashishi lozim bo'lgan tasodifiy nomli **DB**ni aniqlamoqchi bo'lsak, **App.config** faylidagi **configuration** yopiluvchi tegdan so'ng quyidagi kodni qo'shib qo'yamiz:

```
<connectionStrings>

  <add name="DefaultConnection" connectionString="Data
Source=.\SQLEXPRESS;Initial Catalog=Players;Integrated Security=True"

providerName="System.Data.SqlClient"/> </connectionStrings>
```

add elementi orqali **connectionStrings** seksiyasiga **DB**ga ulanish satri keltiriladi.

Bitta loyihada bir qancha **add** elementlaridan foydalanish mumkin.

name="DefaultConnection" atributi ulanish satri nomini ifodalaydi. Ulanish satri nomi ma'lumotlar kontekstiga mos holda shakllantirilishi lozim. Yuqoridagi

misolda bosh klass konstruktoriga **base("DefaultConnection")** qiymatni parametr qilib uzatdik. Agar konstruktor parametrsiz holda chaqirilsa,

```
class SoccerContext : DbContext
{
public SoccerContext()
{ }
public DbSet<Player> Players { get; set; }
}
```

Ushbu holda, ulanish satri kontekst nomi bilan ustma-ust tushishi lozim: **name="SoccerContext".**

Ulanish satridagi keyingi element ulanishning parametrni aniqlaydi. Ushbu element bir qancha qismlardan iborat:

- **Data Source: server** nomi. **MS SQL Express** uchun ushbu parametr. **\SQLEXPRESS** qiymatga teng;
- **Initial Catalog: DB** katalogi nomi. Bizning misolda bu qiymat **Players** ga teng, shuning uchun **Code First** yondashuvchidan foydalanilganda serverda **Players.mdf DB** si hosil qilinadi;
- **Integrated Security**: foydalanuvchi parametrlari.

Oxirgi element sifatida provayder o'rnatiladi:
providerName="System.Data.SqlClient"

Natijada, dastur ishlashi jarayonida **Players DB** hosil qilinadi (agar mavjud bo'lmasa) va ishlatiladi.

Model First va Database First yondashuvlarida ulanish satri

Model First va **Database First** yondashuvlaridan foydalanilganda ulanish satri biroz boshqacha ko'rinish oladi:

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

Bu yerda ulanish satrida quyidagi parametrlar muhim:

- **metadata** parametri. Modeldagi metama'lumotlarni o'zida saqlaydi (ushbu holda model **Person** kabi nomlanganligi sababli, metama'lumotlar: **Person.csdl, Person.ssdl, Person.msl**)
- **data source** parametri. **MS SQL** serverni moslashtiradi.
- **initial catalog** parametri. **DB** katalogini o'rnatadi.

Ushbu holda ma'lumotlar bazasi nomini, uning joylashgan manzilini o'zgartirish yoki boshqa modelni ishlatish uchun yuqorida keltirilgan parametrlarga muayyan o'zgartirishlarni amalga oshirish lozim.

Navigatsion xususiyatlar va lazy loading

Yuqoridagi misolda model sifatida **Player** klassidan foydalanildi. Ushbu model futbolchini ifodalab, to'rtta xususiyatdan iborat edi. Ammo ushbu model juda sodda hisoblanib, real hayotda bizning **DB** bir qancha jadvallar mavjud bo'lishi mumkin va ushbu jadvallar o'zaro turli usullarda bog'langan.

Har bir futbolchini muayyan jamoaga biriktirish lozim bo'lsin. Yoki bitta futbol jamoasida bir qancha futbolchilar mavjud bo'lsin. Ushbu holda yuqoridagi **Player** va **Team** ob'ektlari birga ko'p (**one-to-many**) kabi aloqaga kirishadi.

Bizda **Team** klassi quyidagicha aniqlangan bo'lsin:

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

public string Name { get; set; } // jamoa nomi public string Coach { get; set; } // trener
}
```

Player klassi esa futbolchini tavsiflab, quyidagicha ko'rinishga ega bo'lsin:

```
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; }
}
```

Ushbu Player klassida oddiy **Name, Position** va **Age** tiplaridan farqli ravishda tashqi kalit aniqlangan. Tashqi kalit oddiy va navigatsion xususiyatlardan iborat.

Player klassidagi public Team Team { get; set; } xususiyati - navigatsion xususiyat deb nomlanadi. Futbolchi haqidagi ma'lumot shakllantirilayotgan vaqtda u **DB**dan ma'lumotlarni avtomatik oladi.

Tashqi kalitning ikkinchi qismi – **TeamId** xususiyati hisoblanadi. Navigatsion xususiyat bilan bog'lanishda tashqi kalit nomi quyidagilardan biriga mos bo'lishi shart:

- *Navigatsion_xususiyat_nomi+Bog'langan_jadvaldagi_kalit_nomi* – bizning misolda navigatsion xususiyat nomi **Team, Team** modelidagi kalit - **Id**, shuning uchun ushbu hol uchun xususiyat **TeamId** kabi aniqlangan.

- *Bog'langan_jadval_klass_nomi+Bog'langan_jadvaldagi_kalit_nomi* – bizning holda klass - **Team, Team** modeldagi kalit - **Id**, shuning uchun ushbu hol uchun xususiyat **TeamId** kabi aniqlangan.

Tashqi kalit bog'langan ma'lumotlarni olishga xizmat qiladi. **Code First** yondashuvi orqali **DB** generatsiya qilinganidan so'ng, **Players** jadvali quyidagicha tashkil qilingan edi:

```
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])

);
```

Tashqi kalitni aniqlashda quyidagilarga e'tibor berish lozim. Agar tashqi kalitdagi oddiy xususiyat tipi **int?** kabi aniqlangan bo'lsa (ya'ni **null** qiymatga teng bo'lishi mumkinligi), **DB**da hosil qilinadigan maydon, **NULL** qiymatni qabul qilish mumkin: **[TeamId] INT NULL**

Ammo biz **Player** klassida **TeamId** tipini **int**: **public int TeamId** { **get; set;** } kabi aniqlasak, ushbu xususiyatga mos maydon **NOT NULL** cheklagichga ega bo'ladi. Tashqi kalit esa kaskadli o'chirishni aniqlaydi:

```
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 );
```

Bog‘langan ma’lumotlarni olish

Entity Framework da bog‘langan ma’lumotlarni turli usullarda olish mumkin. Ularning biri **«ochko’z yuklash»** yoki «**eager loading»** deb nomlanadi. Ushbu usul orqali bog‘langan ma’lumotlarni tashki kalit bo’yicha **Include** metodi orqali olish mumkin. Yuqoridagi misolda barcha futbolchilarni ularning jamoalari bilan birga olamiz:

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

{
MessageBox.Show(p.Team.Name);
}
}
```

Include metodini ishlatmasdan, bog‘langan ma’lumotlarga ega bo’lmasdik va **p.Team.Name** xususiyatga murojaat qilib bo’lmas edi.

Bog‘langan ma’lumotlarni olishning ikkinchi usuli «ishyoqmas yuklash» yoki

lazy loading deb yuritiladi. Ushbu usulda agar ob’ektga murojaat qilinganda bog‘langan ma’lumotlar zarur bo’lmasa, ular yuklanmaydi. Ammo navigatsion xususiyatga birinchi bor murojaat qilinganda ushbu ma’lumotlar avtomatik tarzda **DB**dan yuklanadi.

«Ishyoqmas yuklash» usuli ishlatilganda klasslarni e’lon qilishda ba’zi qoidalarga rioya qilish lozim. Avvalo «ishyoqmas yuklash»ni amalga oshiruvchi klasslar **public** statusiga, ularning xususiyatlari esa **public** va **virtual** kabi modifikatorlarga ega bo’lishi lozim. Misol, **Player** va **Team** klasslari quyidagicha tarzda aniqlanishi lozim:

```
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; } // jamoa nomi public string Coach { get; set; } //trener

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

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

Program.cs fayli quyidagicha:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace OneToOneApp

{
class Program
{
static void Main(string[] args)
{
using (SoccerContext db = new SoccerContext())

{
//              Barcha o'yinchilar tegishli Jamoалар ro'yxati

IEnumerable<Player> players = db.Players; foreach (Player p in players)

{
Console.WriteLine("{0} - {1}", p.Team.Name, p.Name);
}

Console.WriteLine();
//Barcha Jamoалар va undagi o'yinchilar
IEnumerable<Team> teams = db.Teams;
foreach (Team t in teams)
```

```
{

Console.WriteLine("{0}. {1}",t.Id,t.Name); IEnumerable<Player> ps = t.Players;
foreach (Player p in ps) {

Console.WriteLine("\t{0}. {1}", p.Id, p.Name);
}
}
}
Console.ReadLine();
}
}
}
```

Endi **Entity Framework** orqali loyihada modellarni bog‘lash usullarini ko’rib chiqamiz.

Birga-bir bog‘lanish

Entity Framework da birga-bir bog‘lanish birga-ko’p bog‘lanishga o’xshash tarzda amalga oshiriladi. Bizda foydalanuvchilarni ifodalovchi **User** klassi mavjud bo’lib, unga mos ob’ektlarda login va parol saqlanadi. Foydalanuvchi profillari haqidagi ma’lumotlar esa **UserProfile** klassiga mos tuzilmada saqlanadi. **User** va **UserProfile** modellari o’zaro birga-bir aloqaga ega.

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

using System.Text;
using System.Data.Entity;
using System.Data;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace OneToOneApp
{

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

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

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

public int Id { get; set; }

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

}
```

Yuqoridagi tarzda amalga oshirilgan aloqada **UserProfile** klassi **User** klassiga nisbatan bo'ysunuvchi hisoblanadi. Modellar o'rtasida birga-bir aloqani o'rnatish uchun bo'ysunuvchi klassda bosh klassdagi kabi identifikator xususiyati o'rnatiladi. Yuqoridagi misolda ushbu vazifani **User** va **UserProfile** klasslaridagi **Id** xususiyati bajaradi.

UserProfile klassida **Id** xususiyatida ikkita atribut o'rnatilgan: **[Key]** va **[ForeignKey]**. **[Key]** atributi **Id** xususiyatining birlamchi kalit ekanligini,

[ForeignKey] atributi esa ushbu xususiyat tashqi kalit ekanligini anglatadi. Ushbu ikkilamchi kalit **User** jadvalidagi **Id** xususiyatiga mos qo'yilgan.

Shuning uchun **User** va **UserProfile** klasslari o'zaro bir-biriga uzatmaga ega.

Ma'lumotlar kontekstida ushbu klasslar **DB**dagi mos jadvallarni ifodalaydi:

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

public DbSet<UserProfile> UserProfiles { get; set; }
}
```

Ushbu klasslar uchun ma'lumotlar konteksti asosida **DB**da quyidagi jadvallar hosil qilinadi:

```
CREATE TABLE [dbo].[Users](
[Id] [int] IDENTITY(1,1) NOT NULL,
[Login] [nvarchar](max) NULL,
[Password] [nvarchar](max) NULL,

CONSTRAINT [PK_dbo.Users] PRIMARY KEY CLUSTERED
)
```

```
CREATE TABLE [dbo].[UserProfiles] (
[Id]      INT           NOT NULL,
[Name] NVARCHAR (MAX) NULL,
[Age] INT               NOT NULL,

   CONSTRAINT [PK_dbo.UserProfiles] PRIMARY KEY CLUSTERED ([Id] ASC),

CONSTRAINT [FK_dbo.UserProfiles_dbo.Users_Id] FOREIGN KEY ([Id]) REFERENCES [dbo].[Users] ([Id])
);
```

Yuqoridagi klasslarga mos modellar ustida ba’zi amallarni ko’rib chiqamiz.

Yangi yozuv qo’shish va **o’qib olish**:

```
static void Main(string[] args)
{
using (UserContext db = new UserContext())
{

User user1 = new User { Login = "login1", Password = "pass1234" }; User user2 = new User { Login = "login2", Password = "5678word2" }; db.Users.AddRange(new List<User> { user1, user2 }); db.SaveChanges();

            UserProfile profile1 = new UserProfile { Id = user1.Id, Age = 22, Name = "Ahmad" };

            UserProfile profile2 = new UserProfile { Id = user2.Id, Age = 27, Name = "Salim" };

db.UserProfiles.AddRange(new List<UserProfile> { profile1, profile2 });
db.SaveChanges();

foreach (User user in db.Users.Include("Profile").ToList())

Console.WriteLine("Name: {0} Age: {1} Login: {2} Password: {3}", user.Profile.Name, user.Profile.Age, user.Login, user.Password);
}
Console.ReadLine();
```

Tahrirlash

```
static void Main(string[] args)
{
using (UserContext db = new UserContext())
{
User user1 = db.Users.FirstOrDefault();
if (user1 != null)
```

```
{

user1.Password = "dsfvbggg"; db.Entry(user1).State = EntityState.Modified;
db.SaveChanges();
}

UserProfile profile2 = db.UserProfiles.FirstOrDefault(p => p.User.Login
==     "login2");
if (profile2 != null)
{
profile2.Name = "Alice II";

db.Entry(profile2).State = EntityState.Modified; db.SaveChanges();
}
}
Console.ReadLine();
}
```

Modelda o'chirish amalini bajarishda **UserProfile** ob'ekti **User** ob'ektining mavjud bo'lishini talab qiladi. Shu sababli, **User** ob'ekti o'chirilishi natijasida **UserProfile** ob'ekti ham o'chiriladi. **UserProfile** ob'ekti o'chirilishi natijasida **User** ob'ekti o'chirilmaydi.

```
static void Main(string[] args)
{
using (UserContext db = new UserContext())
{

User user1 = db.Users.Include("Profile").FirstOrDefault();
if (user1 != null)
{
db.UserProfiles.Remove(user1.Profile);
db.Users.Remove(user1);
db.SaveChanges();
}

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

db.SaveChanges();
```

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

Birga-ko’p munosabat

Birga ko’p munosabat orqali tashkil qilingan modelda, bitta modelda boshqa model ob’ektiga uzatma mavjud bo’lsa, ikkinchi model esa birinchi model ob’ektlari kolleksiyasiga uzatmani saqlashi lozim. Misol, bitta jamoada bir qancha futbolchilar mavjud bo’lgan holda:

```
class Player
{
public int Id { get; set; }
public string Name { get; set; }
public string Position { get; set; }
public int Age { get; set; }
public int? TeamId { get; set; }
public Team Team { get; set; }
}
class Team
{
public int Id { get; set; }
public string Name { get; set; } // jamoa nomi
public ICollection<Player> Players { get; set; }
public Team()
{
Players = new List<Player>();
}
}
class SoccerContext : DbContext
{
public SoccerContext()

:           base("SoccerContext")
{ }

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

```
}
```

Ushbu klasslarga mos holda quyidagi amallarni ko’rib chiqamiz**.**

Yangi yozuv qo’shish va **olish:**

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

namespace OneToMoreApp
{
class Program
{
static void Main(string[] args)
{
using (SoccerContext db = new SoccerContext())
{

// modellarni yaratish va qo’shish
Team t1 = new Team { Name = "Barselona" };
Team t2 = new Team { Name = "Real Madrid" };
db.Teams.Add(t1);
db.Teams.Add(t2);
db.SaveChanges();

        Player pl1 = new Player { Name = "Ronaldu", Age = 31, Position =
"Hujumchi", Team = t2 };

        Player pl2 = new Player { Name = "Messi", Age = 28, Position =
"Hujumchi", Team = t1 };

        Player pl3 = new Player { Name = "Xavi", Age = 34, Position = "Yarim
himoyachi", Team = t1 };

db.Players.AddRange(new List<Player> { pl1, pl2, pl3 }); db.SaveChanges();

// konsolga chiqarish

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("Komanda: {0}", t.Name); foreach (Player pl in t.Players) {
```

```
                Console.WriteLine("{0} - {1}", pl.Name, pl.Position);
}
Console.WriteLine();
}
}

}
}
}
```

O’zgartirish:

```
// o’zgartirish

t2.Name = "Реал М."; // jamoa nomini o’zgartiramiz db.Entry(t2).State =
EntityState.Modified;

//bir futbolchini boshqa jamoga o'tkazamiz pl3.Team = t2;

db.Entry(pl3).State = EntityState.Modified; db.SaveChanges();
```

O’chirish:

```
// O’chirish

Player pl_toDelete = db.Players.First(p => p.Name == "Ronaldu");
db.Players.Remove(pl_toDelete);

// Komandani o’chirish
Team t_toDelete = db.Teams.First();
db.Teams.Remove(t_toDelete);
db.SaveChanges();
```

DB dan muayyan jamoa o’chirilganda **Player** ob’ektining **Team** xususiyati **null** ga teng bo’ladi.

Birga ko’p aloqa. Misol

Avvalgi bo’limlarda keltirilgan materiallar o’zlashtirilganidan so’ng, yangi loyiha yaratib, birga ko’p aloqani amalga oshirish mumkin. Buning uchun **lazy loading** texnologiyasidan foydalanamiz.

Yangi **Windows Forms** loyihasini yaratamiz. Birinchi navbatda loyihamizga

NuGet Entity Framework ni va barcha zaruriy modellarni qo’shamiz:

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

```
namespace OneToMore
{
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; } // jamoa nomi public string Coach { get; set; } // trener

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; }
}
}
```

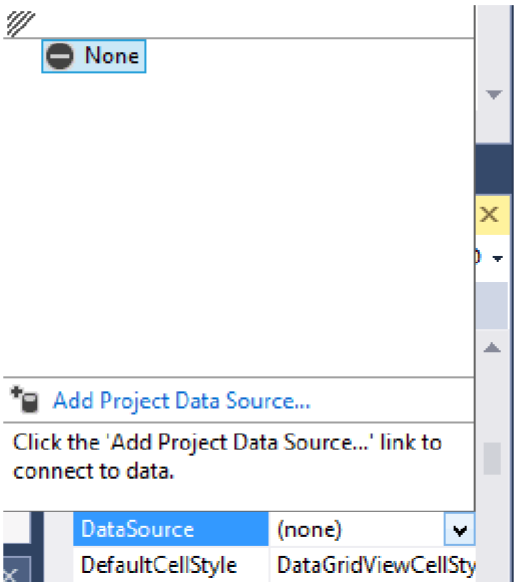
Bizning grafikli loyihamiz to’rtta formadan iborat bo’ladi: **futbolchilar ro'yxati, jamoalar ro'yxati, futbolchini qo’shish/o’zgartirish va jamoani qo’shish/o’zgartirish**.

Futbolchilarni ifodalovchi forma quyidagi ko’rinishga ega bo’lsin:

57-Rasm

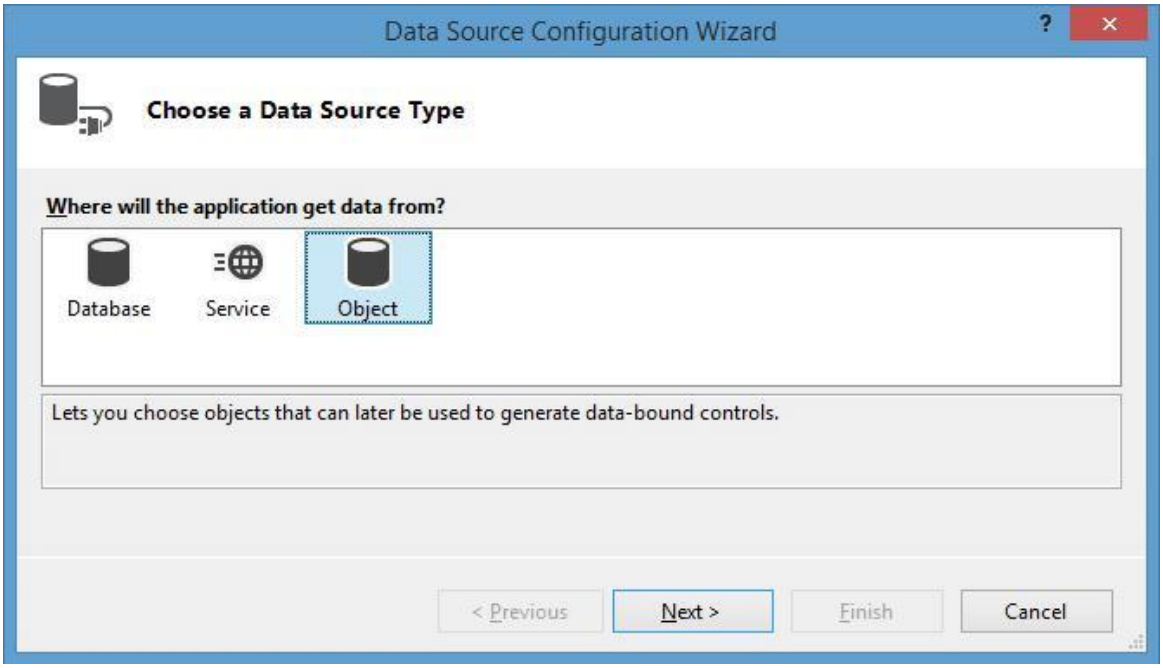
Ushbu formadagi **DatatGridView** elementi ma’lumotlarni ifodalash uchun, uchta tugma (**qo’shish/o’chirish/o’zgartirish**) va futbol jamoalarini chaqiriluvchi bitta tugmadan iborat.

Yuqoridagi mavzularda ma’lumotlarni **DataGridView** ga bog‘lashni ko’rib chiqqan edik. Ma’lumotlarni bog‘lash jarayonida har bir xususiyatga mos ustun hosil qilinadi. Ammo bizga **TeamId** xususiyati zarur emas. Shuningdek, **DataGridView** dagi ustun sarlavhalarida xususiyatlarning nomlari yozilishi maqsadga muvofiq. Shuning uchun **DatatGridView** ni tanlab, xususiyatlar oynasidan ni **DataSource** tanlaymiz. Ushbu xususiyatga qiymat tanlash vaqtida quyidagi interfeys shakllantiriladi:



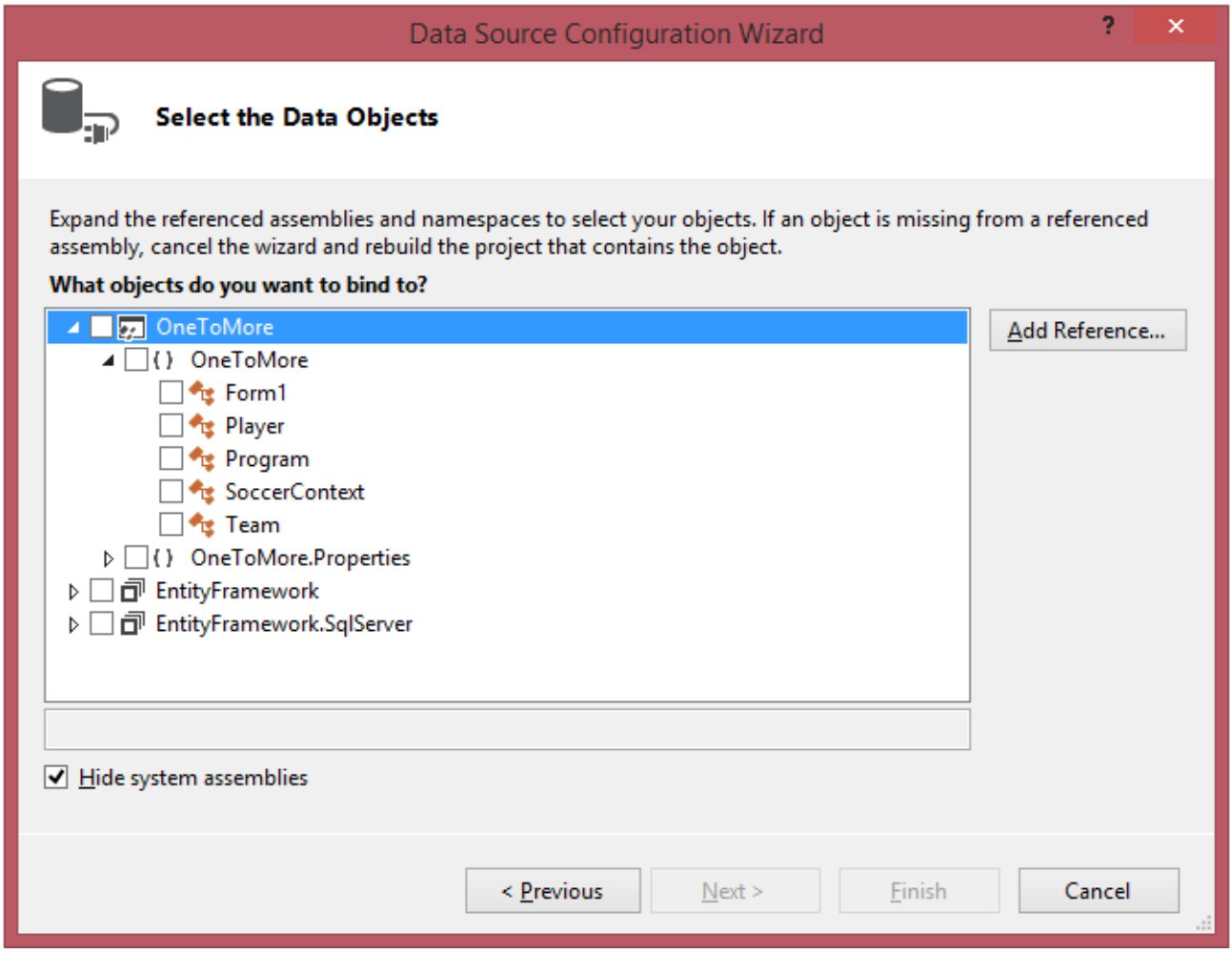
58-Rasm

Ushbu muloqot oynasidan **Add Project Data Source...**ni tanlaymiz. Shundan so'ng bizga ma'lumotlar manbaasi oynasi shakllantirilib, undan **Object** ni tanlashimiz lozim.



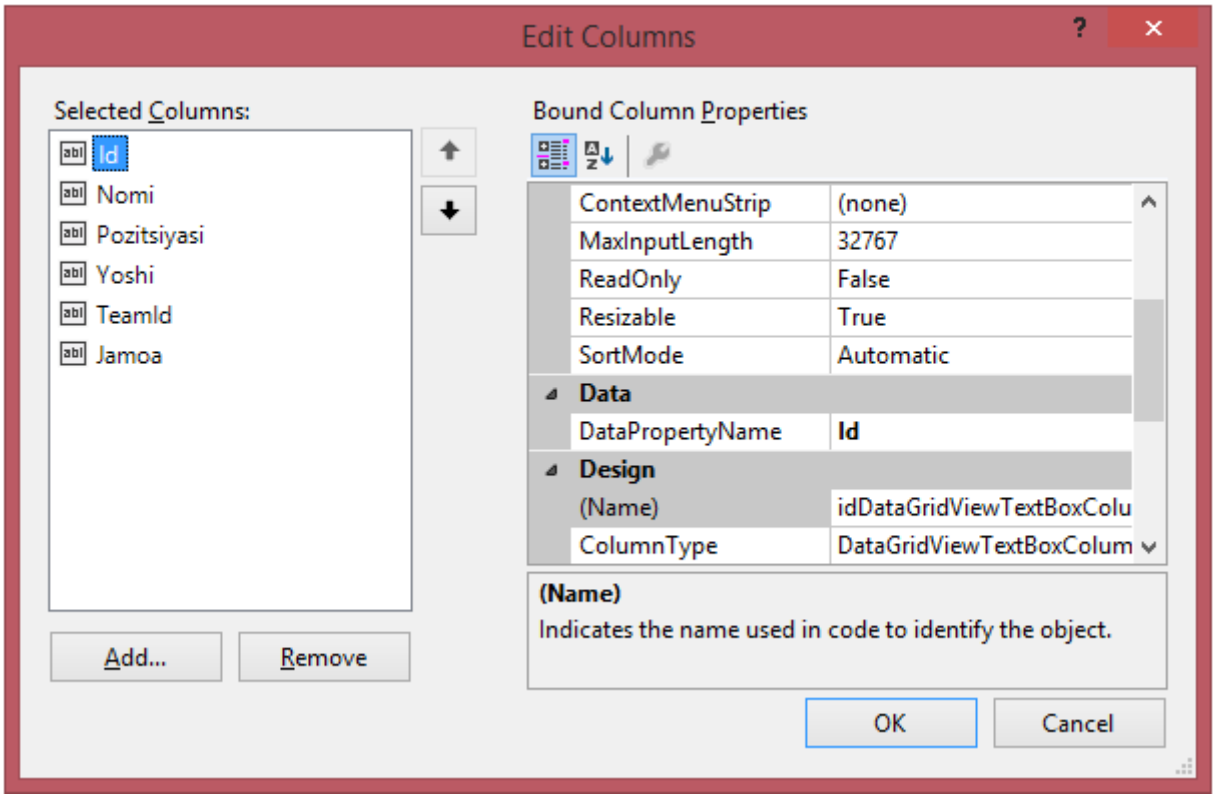
59-Rasm

Keyingi qadamda bizda loyiha tuzilmasi shakllantiriladi va ushbu tuzilmadan **Player** klassini topib olamiz:



60-Rasm

Shundan so’ng **DataGridView** da klass xususiyatga mos ustunlar shakllantiriladi. **DataGridView** dagi **Columns** xususiyatiga o'tamiz. **HeaderText** xususitiga kerakli ustunlaga mos qiymatlarni beramiz va ushbu qiymatlar **DataGridView** dagi mos ustunlarga sarlavha sifatida namoyish qilinadi. **TeamId** ustunini esa o’chiramiz:



61-Rasm

Yuqorida keltirilgan misoldagi kabi **Team** klassi uchun mos formani shakllantiramiz:

62-Rasm

Ushbu formadagi **DataGridView** ob’ektida jamoalar ro'yxati chiqariladi. Shuningdek formada bitta **ListBox** va **'Tarkib'** tugmasi ham mavjud. Ushbu ro'yxatda tanlangan jamoada mavjud futbolchilar ro'yxati chiqariladi.

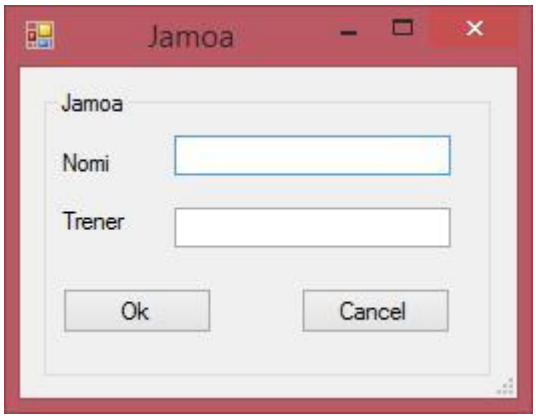
Loyihaga futbolchi va jamoa qo’shish/o’zgartirish uchun formani ham qo’shish lozim. Futbolchi uchun formani **PlayerForm** kabi nomlaymiz:

63-Rasm

Ushbu formada futbolchi **FIO**si uchun matnli maydon, yosh uchun **NumericUpDown** va ikkita **ComboBox** elementi mavjud.

“OK” tugmasidagi **DialogResult** xususiyatiga va **“Отмена”** tugmasidagi **DialogResult** xususiyatiga **Cancel** qiymatini beramiz. Barcha maydonlarda **Modifiers** ni **Private** dan **Protected Internal** ga o’zgartiramiz.

Yangi jamoa hosil qilish **TeamForm** formasi quyidagicha:



64-Rasm

Tugmalar va maydonlar uchun **DialogResult** va **Modifiers** xususiyatlarini avvalgi forma kabi o’zgartiramiz.

Futbolchilar ro'yxatidan tarkib topgan bosh forma kodi quyidagicha:

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

using System.Windows.Forms;
namespace OneToMore
{
public partial class Form1 : Form
{
SoccerContext db;
public Form1()
{
InitializeComponent();
db = new SoccerContext();
db.Players.Load();
dataGridView1.DataSource = db.Players.Local.ToBindingList();

}
private void Form1_Load(object sender, EventArgs e)
{
}
private void button1_Click(object sender, EventArgs e)
{
PlayerForm plForm = new PlayerForm();
//              db dan jamoa ro'yxatini shakllantirish

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("Yangi futbolchi qo'shildi!");
}

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("Obyekt yangilandi!");
}
}

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("Obyekt o’chirildi");
}

}
private void button4_Click(object sender, EventArgs e)

{
TeamForm teams = new TeamForm();
teams.Show();
}
}
}
```

Avvalgi misollarda ham xuddi yuqorida keltirilgan kodda o’xshash amallar bajarilgan. Faqat qo’shimcha tarzda jamoani tanlash maydoni qo’shib qo’yilgan:

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

Tahrirlash jarayonida esa ushbu maydon qiymatini quyidagi kod asosida amalga oshiramiz
plForm.comboBox2.SelectedValue = player.Team.Id;

lazy loading ga ko'ra jamoa va unga mos o'yinchilarga ega bo'lishimiz va ularning xususiyatlariga murojaat qilishimiz mumkin.

Jamoa formasi kodi

:

```
using System;

using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Drawing;
using System.Linq;
using System.Text;

using System.Windows.Forms;
namespace OneToMore
{
public partial class TeamForm : Form
{

SoccerContext db;
public TeamForm()
{
InitializeComponent();
db = new SoccerContext();
db.Teams.Load();
dataGridView1.DataSource = db.Teams.Local.ToBindingList();

}

private void button1_Click(object sender, EventArgs e)
{
PTeam tmForm = new PTeam();
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("Yangi obyekt qo’shildi");
}

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("Obyekt o’chirildi");
}
}

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);

PTeam tmForm = new PTeam();
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("Obyekt yangilandi");
}

}
}
}
```

Ushbu kodga nimaga ahamiyat berish lozim? Birinchidan, **lazy loading** orqali jamoa va undagi futbolchilar kabi bog‘langan ma’lumotlarga ega bo’lamiz. Ularni

ListBox: listBox1.DataSource = team.Players.ToList(); orqali yuklab olish mumkin.

Ikkinchidan, o’chirish jarayonida ushbu ro'yxatni **team.Players.Clear();** kabi tozalashimiz mumkin.

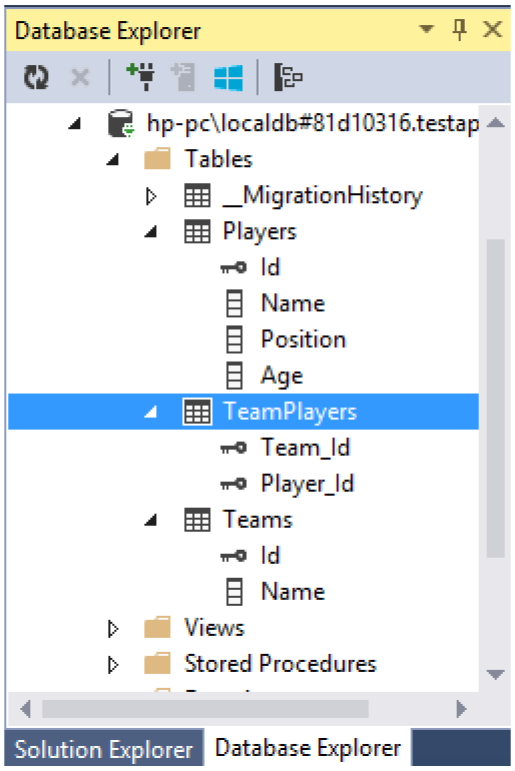
Agar biz «qo'lda» **DB**ni hosil qilgan bo'lsak**, entity framework** orqali **database first** yoki **code first** yondashuvi orqali loyihani uni biriktirishimiz mumkin. Tashqi kalitni hosil qilish jarayonida **DB**da kaskadli o'chirishni amalga oshirishimiz mumkin. Yoki jamoa o'chirilganda u bilan bog'liq bo'lgan o'yinchilardagi maydon qiymatini **null** ga o'zgartirishimiz mumkin.

Ko'pga-ko'p aloqa

Ob'ektlar o'zaro ko'pga-ko'p aloqada bo'lishi mumkin. Bizda futbolchi va jamoa modeli mavjud bo'lsin. Futbolchi yillar davomida bir qancha jamoalarda to'p surishi mumkin. Bitta jamoada esa bir qancha futbolchilar to'p surishlari mumkin. Ushbu modellar quyidagicha tashkil qilinishi mumkin:

```
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>();
}

}
```

Ikkala model ham kolleksiya xususiyatidan iborat bo'lib, ushbu xususiyatlar orqali ko'pga-ko'p aloqa shakllantirilgan. Agar biz **CodeFirst** yondashuvidan foydalansak, **DB**da avtomatik tarzda quyidagi sxema shakllantiriladi:



65-Rasm

Ya’ni ikkita ob’ektni o’zaro bog‘lovchi **Player-Team** jadval shakllantiriladi. Agar **Player-Team** yondashuvidan foydalansak, biz mustaqil tarzda ushbu jadvalni shakllantirishimiz lozim.

Yangi yozuv qo’shish va chiqarish

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.Entity;
namespace ManyToMany

{
class Program
{
static void Main(string[] args)
{
using (SoccerContext db = new SoccerContext())
{

// Yangi model yaratish va qo’shish

            Player pl1 = new Player { Name = "Ronaldu", Age = 31, Position =
"Hujumchi" };
```

```
            Player pl2 = new Player { Name = "Messi", Age = 28, Position =
"Hujumchi" };

            Player pl3 = new Player { Name = "Xavi", Age = 34, Position = "Yarim
himoyachi" };

db.Players.AddRange(new List<Player> { pl1, pl2, pl3 }); db.SaveChanges();

Team t1 = new Team { Name = "Barselona" };
t1.Players.Add(pl2);
t1.Players.Add(pl3);
Team t2 = new Team { Name = "Real Madrid" };
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("Jamoa: {0}", t.Name); foreach (Player pl in t.Players) {

                Console.WriteLine("{0} - {1}", pl.Name, pl.Position);
}
Console.WriteLine();
}
}
}

}
}
```

Ro’yxatda bitta modelni boshqasiga qo’shishda ushbu ro’yxat shakllangan

bo’lishi shart. Aks holda xatolik qaytariladi. Ushbu holda har ikkala modellar konstruktorida ro'yxat shakllantirilgan.

O’zgartirish

```
// Bitta obyekt bilan aloqani o’chiramiz

Player pl_edit = db.Players.First(p => p.Name == "Messi"); Team t_edit =
pl_edit.Teams.First(p => p.Name == "Barselona"); t_edit.Players.Remove(pl_edit);
```

Komandadan biror futbolchini o’chirish natijasida **TeamPlayers** jadvalidagi futbolchi **Id** si va komanda **Id** siga mos yozuvning o’chirilishiga olib keladi.

Futbolchini **DB**dan o'chirish esa futbolchi **Id** siga mos bo'lgan **TeamPlayers** dagi barcha yozuvlarning o'chirilishiga olib keladi.

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

Ko'pga-ko'p aloqa.Misol

Windows Forms tipiga mansub yangi loyihani hosil qilamiz va unga ushbu modellarni qo'shib qo'yamiz. So'ngra, **NuGet** orqali **Entity Framework** paketini va mos ma'lumotlar kontekstini hosil qilamiz:

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

{ }

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

Modellar:

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

public string Coach { get; set; }

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

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

public virtual ICollection<Team> Teams { get; set; }
```

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

Endi asosiy dasturni shakllantiramiz. Asosiy formada jalval shaklida futbolchilar ro'yxati shakllantiriladi:

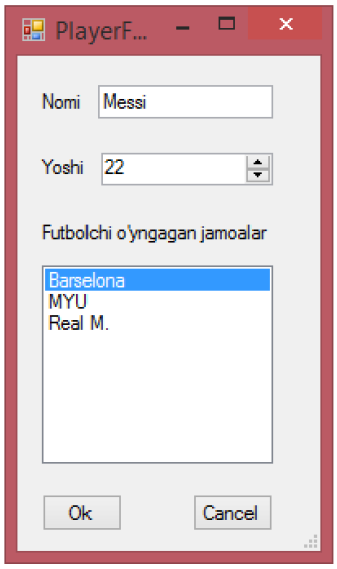
66-Rasm

Ushbu formadagi funksional avvalgi misoldagi kabi amalga oshiriladi:bitta

DataGridView va to’rtta tugma. Shuningdek, ikkita qo’shimcha formalar zarur:

bittasi yangi jamoani hosil qilish uchun, ikkinchisi – futbolchini qo’shish/o’zgartirish uchun.

Loyihamizga yangi formani qo’shamiz. Unga **PlayerForm** nomini beramiz:

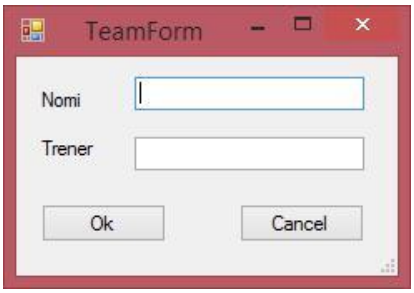


67-Rasm

Bu yerda matnli maydonda futbolchining FIOsi, maydoni **NumericUpDown** futbolchi yoshi uchun, **ListBox** elementida esa jamoalar ro'yxati namoyish qilinadi. Formadagi barcha elementlarning **Modifier** xususiyatini **Protected Internal** ga tenglashtiramiz.

Shuningdek ushbu formada ikkita tugma mavjud. '**OK**' tugmasining **DialogResult** xususiyati qiymatini **OK** ga, 'Otmena' tugmasining **DialogResult** xususiyati qiymatini **Cancel** ga tenglashtiramiz.

Yangi jamoani qo'shish formasini **TeamForm** deb nomlaymiz va u quyidagi ko'rinishga ega:



68-Rasm

Ushbu formadagi elementlarning xususiyatlari qiymatlarini yuqoridagi futbolchiga mos forma elementlariga o'xshash tarzda o'zgartiramiz. So'ngra bosh formadagi o'yinchilar ro'yxatini chiqaruvchi kodni shakllantiramiz:

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

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

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

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

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

6.          plForm formaga buyruqlar ro'yxatini ochish List<Team> teams =
db.Teams.ToList(); plForm.listBox1.DataSource = teams;
plForm.listBox1.ValueMember = "Id"; plForm.listBox1.DisplayMember = "Name";

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

Player player = new Player();

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

        teams.Clear(); // ro'yxat tozalanadi va uni qaytadan ma'lumotlar bilan
//to'ldirib chiqiladi

foreach (var item in plForm.listBox1.SelectedItems)
{
```

```
teams.Add((Team)item);
}
player.Teams = teams;
db.Players.Add(player);
db.SaveChanges();

MessageBox.Show("Yangi futbolchi qo’shildi");

}

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

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

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

team.Players = new List<Player>();

db.Teams.Add(team);
db.SaveChanges();
MessageBox.Show("Yangi jamoa qo’shildi");
}

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

int index = dataGridView1.SelectedRows[0].Index; int id = 0;

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

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

PlayerForm plForm = new PlayerForm();
```

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

7. komanda ro’yxatini olish

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

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

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

9. Futbolchining komandasi mavjudligini aniqlaymiz foreach (var team in teams)

```
{
if (plForm.listBox1.SelectedItems.Contains(team))
{

if (!player.Teams.Contains(team)) player.Teams.Add(team);
}

else
{

if (player.Teams.Contains(team)) player.Teams.Remove(team);
}
}

db.Entry(player).State = EntityState.Modified; db.SaveChanges();
MessageBox.Show("Ma’lumot yangilandi");
}

private void button3_Click(object sender, EventArgs e)
{

if (dataGridView1.SelectedRows.Count < 1)
return;
```

```
int index = dataGridView1.SelectedRows[0].Index; int id = 0;

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

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

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

MessageBox.Show("Futbolchi o’chirildi");

}
}
}
```

Ma’lumotlar bazasini initsializatsiya qilish

Agar **DB**ga birinchi marotaba murojaat qilinganda, **DB**ni boshlang‘ich ma’lumotlar bilan to’ldirish lozim bo’lsa, initsializatsiyani amalga oshirishimiz mumkin.

Initsializatsiya ma’lumotlar kontekstiga birinchi marotaba murojaat qilinganda amalga oshiriladi. Initsializatsiya uchun **.NET** bibliotekasida mavjud quyidagi klasslardan foydalanishimiz mumkin:

\{ **CreateDatabaseIfNotExists:** boshlang‘ich holda ishlatiladigan initsializator. U **DB** dagi ma’lumotlarni avtomatik o’chirmaydi. Agar model va ma’lumotlar kontekstini tuzilmasi o’zgartirilgan vaqtda xatolikni qaytaradi.

\{ **DropCreateDatabaseWhenModelChanges:** ushbu initsializator model va **DB** jadvallari mosligini aniqlaydi. Agar modelda jadvalga moslik mavjud bo’lmasa, **DB** qayta yaratiladi.

: **DropCreateDatabaseAlways:** Ushbu initsializator orqali **DB** qayta shakllantiriladi

Ushbu initsializatorlarning biridan foydalanish lozim. Buning uchun bizga **Seed** metodi xizmat qiladi:

1) Phones.cs

```
using System.Data.Entity;
namespace Init

{
class Phone
```

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

: **MobileContext.cs**

```
using System.Data.Entity;

namespace Init
{

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

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

: **MyContextInitializer.cs**

```
using System.Data.Entity;

namespace Init
{
class MyContextInitializer : DropCreateDatabaseAlways<MobileContext>
{
protected override void Seed(MobileContext db)
{

Phone p1 = new Phone { Name = "Samsung Galaxy S5", Price = 14000 }; Phone p2
= new Phone { Name = "Nokia Lumia 630", Price = 8000 };

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

```
}
}
```

4. **Programm.cs** using System;

```
namespace Init
{
class Program
{
static void Main(string[] args)

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

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

Initsializator klassi yuqorida keltirilgan biror klassdan vorislanadi. Bizning misolda **DropCreateDatabaseAlways<MobileContext>** klassidan vorislangan.

Initsializatsiyaning barcha amallari Seed metodida amalga oshiriladi. Initsializatsiya jarayoni **DB**ga ma'lumotlar konteksti orqali saqlanadi.

Initsializator ishga tushishi uchun uni chaqirish lozim. Initsializatorni chaqirish uchun kontekst klassidagi statik konstruktordan foydalanishimiz mumkin:

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

Entity Framework da parallel dasturlash

Bir nechta foydalanuvchilar bir vaqtda ma'lumotlar ustida **Entity Framework** texnologiyasi orqali amal bajarayotganda parallellash muammosi yuzaga keladi. Masalan, ikkita foydalanuvchi bir-biridan mustaqil tarzda bir ob'ektni tahrirlayotgan bo'lsa, birinchi foydalanuvchi ma'lumotlarni **DB**ga saqlagandan so'ng, ikkinchi foydalanuvchi aktual bo'lmagan ma'lumotlar ustida amal bajaradi.

Misol orqali ushbu amalni ko'rib chiqamiz. **ASP.NET MVC** dasturida tahrirlash uchun standart amal quyidagicha aniqlangan bo'lsin:

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

return View(p);
}

[HttpPost]
public ActionResult Edit(Person p)
{
db.Entry(p).State = EntityState.Modified;

db.SaveChanges();
return RedirectToAction("Index");
}
```

Yuqorida keltirilgan dastur kodi to'g'ri ishlasada, ikkita foydalanuvchi bitta ob'ekt ustida tahrirlash amali bajarayotganda, birinchi foydalanuvchi o'zgartirilgan ob'ekt ma'lumotlarini xotiraga saqlagach, ikkinchi foydalanuvchi eski ma'lumotlar ustida amal bajaradi.

Bunday holatda **Entity Framework** bizga nimani taklif qiladi?

Ushbu hol uchun parallellash texnologiyasini qo'llash mumkin. Ikki turdagi parallellash texnologiyasi mavjud: optimistik va pessimistik.

Pessimistik parallellashtirishda **(pessimistic concurrency) DB**da amal bajarilayotgan ob'ektdan foydalana olish (dostup) cheklanadi. Ya'ni satrlar faqat o'qish uchun statusiga keltiriladi. Pessimistik parallellashtirishda **DB** darajasidagi murakkab dasturiy mantiq amallarga ishonch bildiriladi va barcha amallarni **DB** o'zi bajaradi. **Entity Framework**da pessimistik parallellashtirish qo'llab quvvatlanmaydi.

Optimistik parallellashda **(optimistic concurrency)** ma'lumotlar ustida bir nechta foydalanuvchilar parallel amal bajarishlari uchun **Entity Framework** ning bir nechta metodlari mavjud.

Ushbu metodlardan biri model xususiyatiga **[Timestamp]** atributini joriy etish hisoblanadi. Ushbu atribut orqali xususiyatga mos jadval yozuvi o'zgartirilishi nazorat qilinadi. Masalan:

```
public class Person
{

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

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

Shuningdek, **Person** modeliga mos ob'ektni o'zgartirishga mo'ljallangan ko'rinishda ushbu xususiyatni yashiringan tarzda e'lon qilish lozim: @Html.HiddenFor(model => model.RowVersion)

Endilikda ikkita foydalanuvchi bir vaqtda biror ob'ektni tahrirlamoqchi bo'lishsa, birinchi foydalanuvchi o'zgartirilgan ma'lumotlarni xotiraga saqlashi natijasida ikkinchi foydalanuvchiga

System.Data.Entity.Infrastructure nomlar fazosida joylashgan **DbUpdateConcurrencyException** xatoligi qaytariladi:

```
[HttpPost]
public ActionResult Edit(Person p)
{
try
{
db.Entry(p).State = EntityState.Modified;
db.SaveChanges();

}
catch (DbUpdateConcurrencyException ex)
{

        ViewBag.Error = "Ob'ekt avval boshqa foydalanuvchi tomonidan
o'zgartirilgan";

return View(p);
}

return RedirectToAction("Index");
}
```

Yuzaga kelgan xatolikni foydalanuvchiga ma'lum qilish uchun **ViewBag.Error** dan foydalanamiz va ikkinchi foydalanuvchining modelda amalga oshirgan o'zgarishlari **DB**da saqlanmaydi.

Tranzaksiyalarni boshqarish

Model ustida bajarilgan qo'shish, o'zgartirish va o'chirish amallarini **DB**da qayd etish uchun **SaveChanges()** metodi chaqirilganda **Entity Framework** tranzaksiyani amalga oshiradi.

C# tilida tranzaksiyalarni boshqarish usullari mavjud. Tranzaksiyalarni boshqarish asosan **DB**da bir nechta o'zaro bog'liq amallarni bajarishda qo'llaniladi. Agar tranzaksiyadagi amallar ketma-ketiligida biror amal noto'g'ri bajarilsa, avval bajarilgan amallar barchasi bekor qilinadi. Misolda ushbu amallarni ko'rib chiqamiz. Bizning DBda **Nodir** ismli inson mavjud bo'lsin. U o'g'il farzand ko'rib, uning ismini ham **Nodir** qo'ydi. Ushbu ikkita **Nodir**larni farqlash uchun **katta Nodir** va **kichkina Nodir** deb o'zgartiramiz:

```
static void Main(string[] args)

{
using (UserContext db = new UserContext())
{
using (var transaction = db.Database.BeginTransaction())
{
try

{
Person p1 = db.Persons.FirstOrDefault(p => p.Name == "Bob"); p1.Name = "Bob
Senior";

db.Entry(p1).State = EntityState.Modified;
Person p2 = new Person { Name = "Bob Junior", Age = 1 };
db.Persons.Add(p2);
db.SaveChanges();

transaction.Commit();
}
catch (Exception ex)
{
transaction.Rollback();
}
}
foreach (Person p in db.Persons.ToList())
Console.WriteLine("Name: {0} Age: {1}", p.Name, p.Age);
}
}
```

Yangi tranzaksiyani hosil qilish uchun quyidagi koddan foydalaniladi:

```
var transaction = db.Database.BeginTransaction()
```

Model ustida zarur amallar bajarilgach, o'zgartirishlarni **transaction.Commit()** metodi orqali **DB**ga fiksirlash mumkin.

Ammo model ustda amallarni bajarish jarayonida xatolik yuzaga kelganda **transaction.Rollback()** metodi orqali barcha amallar bekor qilinadi.

Nazorat savollari:

1.Entity Framework-da turli modellardan foydalanishni tavsiflang.
2.DataGridView dan foydalanishning qanday xususiyatlari bor?
3.Ma'lumotlar bazasini ishga tushirish bosqichlarini keltiring va tavsiflang.
4.Klasslarni yaratishda qanday inisializerlardan foydalanish mumkin?
5.Bitimni amalga oshirish tartibini tavsiflang.

4. LINQ TO ENTITIES. LINQ TO ENTITIES GA KIRISH

Reja:

1. **LINQ to Entities ga kirish.**
2. **Ma’lumotlar bazasidan tanlash va proyeksiyalash.**
3. **Nazorat savollari**

Yuqoridagi bo’limlarda **DB**dan ma’lumotlarni olishda bir qator amallarni ko’rib o’tgan edik. Ushbu amallarning asosida **LINQ (Language Integrated Query)** yoki **LINQ to Entities** texnologiyasi yotadi. **LINQ to Entities** oddiy va tushunarli yondashuv hisoblanib, ma’lumotlarni **SQL** tiliga yaqin ifodalar yordamida olishga xizmat qiladi.

DBdagi ma’lumotlari ustida **LINQ** so’rovlari orqali amallar bajarsakda, **DB**da faqatgina **SQL** tilida yozilgan so’rovlar tushaniladi. Shuning uchun **LINQ to Entities** va **DB** o'rtasida vositachi mavjud bo’lib, u **EntityClient** provayderi hisoblanadi. U orqali **SQL** Server bilan ishlash uchun **ADO.NET** provayderi vositasida interfeys shakllantiriladi.

DB bilan ishlash uchun avvalo **EntityConnection** ob’ekti hosil qilinadi. **EntityCommand** orqali u turli so’rovlarni **DB**ga uzatadi. **EntityDataReader** ob’ekti yordamida esa **DB**dan ma’lumotlarni o’qib oladi.

Ammo dasturchilar ushbu ob’ektlarga to’g‘ridan to’g‘ri murojaat qilishlari shart emas. Barcha amallarni fremwork amalga oshiradi. Dasturchilar asosan **LINQ** yordamida so’rovlarni amalga oshirishlari lozim.

LINQ to Entities so’rovlarini tavsiflashdan avval ushbu bo’lim uchun zarur bo’lgan modellarni aniqlab olamiz:

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

public string Name { get; set; }

public ICollection<Phone> Phones { get; set; } public Company()
```

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


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

public int CompanyId { get; set; }

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

Yuqorida telefon va ishlab chiquvchi kompaniya modeli keltirilgan. Endi ushbu modellarga mos ma'lumotlar konteksti va **DB** initsializatori orqali boshlang'ich ma'lumotlarni shakllantiramiz.

```
class PhoneContext : DbContext
{
static PhoneContext()

{
Database.SetInitializer(new MyContextInitializer());
}
public PhoneContext()
        base("DBConnection")
{ }

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();

}
}
```

So'ngra **App.config** faylida ulanish satri hosil qilish lozim:

```
<connectionStrings>
<add name="DBConnection" connectionString="data source=EBB-PC;initial
catalog=MyDb;integrated

security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

DBda boshlang'ich ma'lumotlarni shakllantirish uchun initsializatorda bir nechta ob'ektlar hosil qilinadi. Initsializatorni ishga tushirish uchun u ma'lumotlar kontekstidagi statik konstruktorda chaqiriladi:

```
static PhoneContext()
{

Database.SetInitializer(new MyContextInitializer());
}
```

Linq to Entities da so'rovlarni amalga oshirish uchun, **Linq to Objects** kabi **LINQ** operatorlarini va **LINQ** kengaytirma metodlarini qo'llashimiz mumkin. **LINQ** ning ba'zi operatorlarini ishlatishga misol:

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

}
```

Xuddi shu so'rovni **LINQ** kengaytirma metodlari orqali amalga oshirish mumkin:

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

Dastur kodi:

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

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

Console.ReadLine();
}

using (PhoneContext db = new PhoneContext())

{
var phones = db.Phones.Where(p => p.CompanyId == 1);

foreach (Phone p in phones)

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

}
}
```

Har ikkala so'rovlar bir xil **sql** ifodani generatsiya qiladi:

```
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]
```

Linq to Entities va **Linq to Objects** ning farqini bilish kerak:

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

Bu yerda ikkita **Where** metodi ishlatilgan bo'lib, ularning qo'llanilishi ikki xil:
db.Phones.Where(p=> p.CompanyId == 1)
ifoda orqali **SQL** ifoda translyatsiya qilinadi. So'ngra **ToList**() metodi so'rov natijasiga ko'ra xotirada ro'yxatni shakllantiradi.

Keyingi qadamda biz xotiradagi ro'yxatga ega bo'lamiz. So'ngra, **Where(p=> p.Id<10)** jamoasi orqali xotiradagi ro'yxatga murojaat qilamiz ва **Linq to Object** dan foydalanamiz.

Endi **LINQ** ni **DB**dan olingan ma'lumotlarga qo'llashni ko'rib chiqamiz.

Ma'lumotlar bazasidan shart asosida tanlash va proeksiya

DBdan shart asosida tanlash uchun **Where** metodidan foydalanish lozim. **DB**da ishlab chiqarilgan tashkiloti **"Samsung"** bo'lgan barcha telefonlarni aniqlash lozim bo'lsin:

```
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);
}
```

DBdan biror modelga mos bitta ob'ektni aniqlash uchun **Find**() metodidan foydalanishimiz mumkin. Ushbu metod **Linq** metodi hisoblanmaydi. U **DbSet** klassida aniqlangan:

```
:  id=3 bo'lgan elementni aniqlash Phone myphone = db.Phones.Find(3);

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

Ammo alternativ sifatida **Linq First()/FirstOrDefault()** metodlaridan foydalanishimiz mumkin. Ular shart asosida aniqlangan elementlar ro'yxatidan birinchi elementi qaytaradi.

FirstOrDefault() metodini ishlatish qulay bo'lib, agar natija bo'sh to'plam bo'lgan holda u **null** qiymatni qaytaradi. **First**() metodi esa ushbu holda xatolikni qaytaradi. Buning uchun quyidagi koddan foydalanish zarur:

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

Console.WriteLine(myphone.Name);
```

Endi proeksiyani amalga oshiramiz. Biz natijaga kompaniya nomini qo'shishimiz lozim bo'lsin. U holda **Include** metodidan foydalanib, ob'ektni boshqa jadvaldagi ob'ekt bilan bog'lash mumkin:

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

Ammo ko'p hollarda tanlanadigan ob'ektning barcha xususiyatlari zarur emas. Ushbu holda **Select** metodi orqali olinayotgan ma'lumotlardan yangi tip hosil qilish mumkin:

```
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);
}
```

Natijada **Select** metodi olingan ma'lumotlardan yangi tipni shakllantiradi. Ushbu holda yangi anonim tip hosil qilinayotgan bo'lib, uning o'rniga foydalanuvchi hosil qilgan tipdan foydalanish ham mumkin. Masalan, bizda Model klassi aniqlangan bo'lsin:

```
class Model
{

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

Biz yuqoridagi misoldagi so'rov natijasini ushbu tipga mos ob'ekt sifatida shakllantiramiz:

```
using (PhoneContext db = new PhoneContext())
{
var phones = db.Phones.Select(p => new Model

{
Name = p.Name,
```

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

Tartiblash

DBdan olingan ma'lumotlarni o'sish tartibida tartiblash uchun, **OrderBy** metodi yoki orderby operatori xizmat qiladi. **Name** xususiyati bo'yicha tartiblashni ko'rib chiqamiz:

```
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);
}
```

Natijada **Entity Framework** quyidagi **SQL** ifodani generatsiya qiladi:

```
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
```

OrderBy metodiga alternativ tarzda orderby operatoridan foydalanish mumkin:

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

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

Teskari tartibda tartiblash uchun **OrderByDescending**() metodidan foydalanish zarur:

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

```
{
var phones = db.Phones.OrderByDescending(p => p.Name); foreach (Phone p in
phones)

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

Agar bizga ma'lumotlarni bir qancha parametrlar bo'yicha tartiblash lozim bo'lsa, **ThenBy**() va **ThenByDescending**() metodlaridan foydalanish lozim. So'rovda ikkita xususiyat bo'yicha tartiblash uchun quyidagi koddan foydalanish mumkin:

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

foreach (var k in tphones)
Console.WriteLine("{0}.{1} - {2}", k.Id, k.Name, k.Price);
```

Jadval hosil qilish

Muayyan shartlar asosida jadvallarni birlashtirish uchun **Join** metodi ishlatiladi. Bizning misolda telefonlar va kompaniyalar jadvallari umumiy kompaniya **Id** siga ega. Shuning uchun ushbu jadvallarni birlashtirish mumkin:

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

var phones = db.Phones.Join(db.Companies, // ikkinchi to'plam p => p.CompanyId, //
birinchi to'plamdagi xususiyat

c => c.Id, // ikkinchi to'plamdagi xususiyat (p, c) => new // natija {

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);
}
```

Join metodi to'rtta parametr qabul qiladi:

// Joriy jadval bilan bog'lash lozim bo'lgan ikkinchi jadval;
// ob'ekta xususiyati – birinchi jadvaldagi ustun. Ushbu ustun orqali ulanish amalga oshiriladi;
// ob'ekt xususiyati - ikkinchi jadvaldagi ustun. Ushbu ustun orqali ulanish amalga oshiriladi;

6. bog‘lanish natijasida hosil qilinadigan yangi ob’ekt. Natijada quyidagi **SQL** ifoda generatsiya qilinadi:

```
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]
```

Yuqoridagi natijaga **join** operatorini qo’llash orqali erishish mumkin:

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

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

Guruhlash

Ma’lumotlarni biror parametr bo’yicha guruhlash uchun **group by** operatori yoki **GroupBy**() metodidan foydalanish zarur. Telefon modellarini ishlab chiqargan tashkilot bo’yicha guruhlashtiramiz:

```
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();
}
}
```

Ushbu holda birlashtirish sharti sifatida kompaniya nomi ishlatildi. Ya’ni bog‘lovchi **Companies** jadvalidagi **Name** ustuni. Guruhlashtirish sharti sifatida

kelgan ustun kalit hisoblanadi. Ushbu kalitga biz guruhdagi **Key** xususiyati orqali ega bo'lishimiz mumkin.

Natijada biz bir qancha guruhlarga ega bo'lamiz. Ushbu guruhlar o'zida bir qancha elementlarni saqlashi mumkin. Bizning misolda quyidagi natija hosil qilinadi:

Samsung
Samsung Galaxy S5 - 20000
Samsung Galaxy S4 - 15000

Apple
iPhone5 - 28000
iPhone4S - 23000

Xuddi yuqoridagi natijaga quyidagi **GroupBy**() metodi orqali ega bo'lish mumkin:

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

Key xususiyatidan tashqari guruhda **Count** xususiyati ham mavjud. U ushbu guruhda mavjud elementlar sonini o'zida saqlaydi. Guruh kaliti va unda mavjud bo'lgan elementlar sonidan iborat yangi elementni hosil qilamiz:

```
using (PhoneContext db = new PhoneContext())
{
var groups = from p in db.Phones
group p by p.Company.Name into g
select new { Name = g.Key, Count = g.Count() };
```

```
5.          alternativ usul
6.          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("Ishlab chiquvchi: {0} modellar soni: {1}", c.Name,

c.Count);
}
```

Bizning misolda quyidagi natija hosil qilinadi:

Ishlab chiquvchi: Apple Modellar soni: 2

Ishlab chiquvchi: Samsung Modellar soni: 2

To’plamlar ustida amallar: birlashma, kesishma va farq

Linq dagi bir qator metodlar tanlash natijasi ustida ish bajaradi. Tanlash natijasi to’plamdan iborat bo’lganligi sababli, ikkita to’plam ustida birlashma, kesishma va farq amallarini bajarish mumkin.
Ammo ushbu to’plamlar bir xil tuzilmaga ega bo’lishlari shart.

Birlashma

Ikkita to’plamni birlashmasini aniqlash uchun **Union()** metodidan foydalaniladi:

```
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);
}
```

Union metodi parametr sifatida ikkinchi to’plamni qabul qiladi va birinchi to’plam bilan birlashtiradi.

Biz ikki xil tuzilmali to’plamlarni birlashtira olmaymiz. Masalan, telefon modellari va kompaniyalar jadvallarini birlashtirish mumkin emas. Ammo quyidagi yozuv o'rinli:

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

Birinchi to’plamda **Select** metodidan so’ng **Name** ustunidan iborat elementlar to’plami shakllantiriladi. Ikkinchi to’plamda **Select** metodidan so’ng kompaniya nomini ifodalovchi **Name** ustunidan iborat elementlar to’plami shakllantiriladi. Shuning uchun har ikkala to’plam bir jinsli bo’lganligi sababli ularni birlashtirish mumkin.

Kesishma

Ikkita to’plamni kesishmasini aniqlash uchun **Intersect()** metodidan foydalaniladi:

```
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);
}
```

Farq

Agar birinchi to'plamdan ikkinchi to'plamda mavjud bo'lmagan elementlarni aniqlashimiz lozim bo'lsa, **Except** metodidan foydalanamiz:

```
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); // natija - iPhone S4

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

Agregat amallari

Linq to Entities orqali **SQL** ning ichki funksiyalariga **Count**, **Sum** va boshqa maxsus metodlar orqali murojaat qilishimiz mumkin.

Tanlashdagi elementlar soni

Count() metodi orqali tanlashdagi elementlar sonini aniqlash mumkin:

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

        :          jami telefon modellari soni int number1 = db.Phones.Count();
\{          Samsung qiymatini o'zida saqlagan modellar sonini aniqlaymiz
int number2 = db.Phones.Count(p => p.Name.Contains("Samsung"));

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

}
```

Minimal, maksimal va o'rta qiymatlar

Minimal, maksimal va o'rta qiymatni aniqlash uchun mos ravishda **Min()**, **Max()** va **Average()** funksiyalardan foydalaniladi. Model bo'yicha minimal, maksimal va o'rta qiymatni aniqlaymiz:

```
using (PhoneContext db = new PhoneContext())
{
// minimal narx
int minPrice = db.Phones.Min(p => p.Price);
```

```
// maksimal narx
int maxPrice = db.Phones.Max(p => p.Price);
\{              Samsung firmasi telefonlarining o’rtacha narxi

            double avgPrice = db.Phones.Where(p => p.Company.Name ==
"Samsung").Average(p => p.Price);

Console.WriteLine(minPrice);

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

Qiymatlar summasi

Sum() metodi orqali qiymatlar summasi aniqlanadi:

```
using (PhoneContext db = new PhoneContext())
{
// barcha telefonlar narxlari summasi
int sum1 = db.Phones.Sum(p => p.Price);

// Samsung firmasi telefonlar narxlari summasi
            int sum2 = db.Phones.Where(p => p.Name.Contains("Samsung"))
.Sum(p => p.Price);
Console.WriteLine(sum1);
Console.WriteLine(sum2);
}
```

Entity Framework da IEnumerable va IQueryable

LINQ kengaytirma metodlari orqali ikkita ob’ekt qaytarilishi mumkin: **IEnumerable** va **IQueryable**. Bir tomondan, **IQueryable** interfeysi **IEnumerable** dan vorislanadi. Shuning uchun **IQueryable** ob’ekti o’z navbatida **IEnumerable** ob’ekt hisoblanadi. Ammo real hayotda ushbu amallar biroz murakkab hisoblanadi. Ushbu ob’ektlar o'rtasidagi interfeyslarda farq funksionallikda hisoblanadi. Shuning uchun ularni o’zaro almashtirib bo’lmaydi.

IEnumerable interfeysi **System.Collections** nomlar fazosida joylashgan. **IEnumerable** ob’ekti xotiradagi ma’lumotlar to’plamidan iborat bo’ladi va ushbu ma’lumotlar bilan faqat oldinga harakatlanishni ta'minlaydi. **IEnumerable** ob’ekti orqali aniqlangan so’rov shu lahzada to’liq bajariladi va dastur orqali undan ma’lumot olish juda tez amalga oshiriladi.

IEnumerable so’rovi bajarilishida barcha ma’lumotlar yuklanadi. Agar bizga filtrlash lozim bo’lsa, filtrlash mijoz tomonida amalga oshiriladi.

IQueryable interfeysi **System.Linq** nomlar fazosida joylashgan. **IQueryable** ob'ekti **DB** ga masofaviy dostupni amalga oshiradi va ma'lumotlar ro'yxatida oldinga va orqaga harakatlanish imkonini beradi. So'rovni shakllantirish jarayonida qaytariladigan ob'ekt sifatida **IQueryable** ishtirok etadi va ushbu jarayon optimallashtiriladi. Natijada so'rovni amalga oshirish jarayonida kam xotira talab qilinadi, tarmoq nagruzkaga tushmaydi. Ammo so'rovlar bajarilishi **IEnumerable** ob'ektga nisbatan sekin amalga oshirilishi mumkin.

Ikkita bir xil shakldagi ifodani olamiz. **IEnumerable** ob'ekti:

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

Ushbu ifodaga mos so'rov quyidagicha:

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

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

Where(p => p.Id > id) metodi orqali natijani filtrlash **DB**dan ma'lumotlar olinganidan so'ng, dasturda amalga oshiriladi.

Filtrlarni moslashtirish uchun biz **Where** metodini quyidagicha amalga oshirishimiz lozim:

db.Phones.Where(p => p.Id > id);

IQueryable ob'ekti:

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

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

So'rov quyidagi ko'rinishga ega:

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

```
FROM [dbo].[Phones] AS [Extent1]
WHERE [Extent1].[Id] >3
```

Yuqoridagi kodda barcha metodlar qo'shiladi, so'rov optimallashtiriladi va keyingi qadamda **DB**dan zarur ma'lumot olinadi. Yuqoridagi ikkita ob'ektdan qaysi birini ishlatish lozim? Bu amal qaysi turdagi masala yechilishiga bog'liq. Agar dasturchiga jadvaldagi barcha ma'lumotlar zarur bo'lsa, katta tezlikka ega bo'lgan **IEnumerable** ni ishlatish tavsiya etiladi. Agar bizga barcha yozuvlar o'rniga muayyan shartlarni qanoatlantiruvchi natija zarur bo'lsa, **IQueryable** ni qo'llash maqsadga muvofiq.

AsNoTracking методи

Ma'lumotlar konteksti **DB**dan ma'lumotlarni olganda, **Entity Framework** olingan ma'lumotlarni keshga joylashtiradi va ushbu ob'ektlar ustida amalga oshirilgan o'zgartirishlarni **SaveChanges()** metodi bajarilguncha nazorat qiladi.

Ma'lumotlarni keshga joylashtirmaslik uchun **AsNoTracking()** metodi qo'llaniladi. Ushbu metod qo'llanilganda so'rovdan qaytarilayotgan qiymat keshlashtirilmaydi. Bu esa **Entity Framework** ning qaytarilgan natija uchun biror bir qo'shimcha amal bajarilishi lozim emasligi va ularni saqlash uchun qo'shimcha xotira zarur emasligini anglatadi.

AsNoTracking() metodi **IQueryable** ro'yxatga qo'llaniladi:

```
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();

}
```

Quyidagi misolni ko'rib chiqamiz. Bizning bazamizda bir qancha **Phones** modellari mavjud bo'lsin:

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

{

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

}
```

Oddiy amallarni bajaramiz:

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

Phone firstPhone = db.Phones.FirstOrDefault(); firstPhone.Name = "Samsung Galaxy Ace 2"; db.SaveChanges();

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

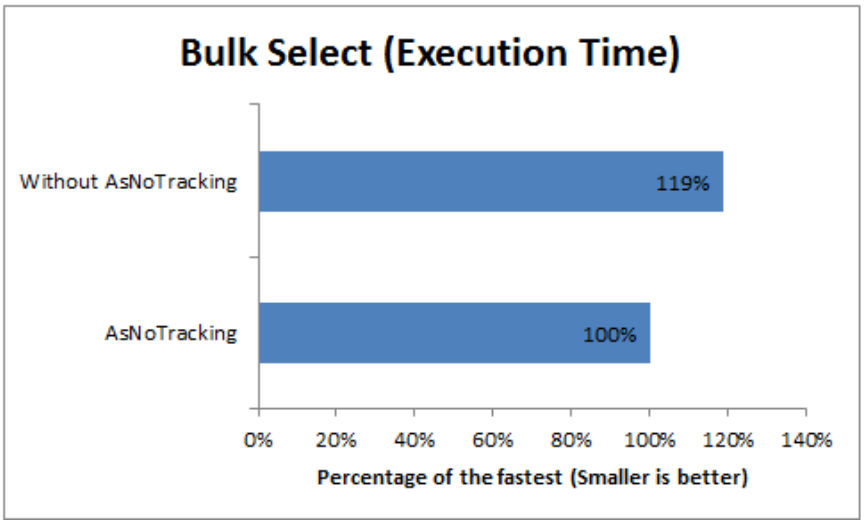
Biz **phones** to'plamda birinchi elementning nomi **"Samsung Galaxy Ace 2"**ekanligiga guvoh bo'lamiz.

Agar biz **AsNoTracking** ni ishlatganimizda natija boshqa xil bo'lar edi:

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

Phone firstPhone = db.Phones.AsNoTracking().FirstOrDefault(); firstPhone.Name = "Samsung Galaxy Ace 2"; db.SaveChanges();

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

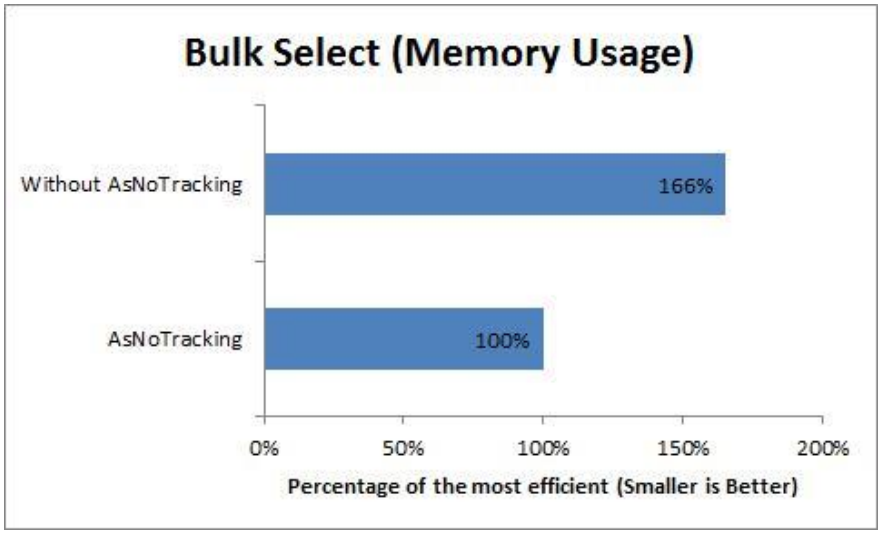
Birinchi elementni olishda **AsNoTracking** ishlatilganligi sababli, u ma'lumotlar ustida hech qanday amalni bajarmaydi. Shuning uchun **db.SaveChanges()** metodi chaqirilganda **DB**da hech qanday o'zgarish amalga oshirilmaydi. Birinchi element esa avvalgidek, **"Samsung Galaxy Note"**ga teng bo'ladi.

Oddiy so'rov bilan **AsNoTracking** metodi orqali amalga oshirilgan so'rovlarning bajarilishini taqqoslaganmizda quyidagi natija hosil qilindi. Bajalishi tezligi bo'yicha taqqoslash:



69-Rasm

Xotiraning ishlatilishi bo'yicha taqqoslash:



70-Rasm

Ushbu natijalar juda katta hajmdagi ma'lumotlar ustida amal bajarilganda to'g'ri natija beradi. Qaysi hollarda **AsNoTracking** metodidan foydalanish zarur? Agar ma'lumotlarni **DB**dan olib faqat namoyish qilish lozim bo'lsa, ular ustida yangilash, o'zgartirish va o'chirish amallari talab etilmasa, **AsNoTracking** dan foydalanish maqsadga muvofiq.

Nazorat savollari:

- Entity Framework-da o'rnatilgan SQL funksiyalarini tavsiflang.
- Entity Frameworkda IEnumerable va IQueryable usullaridan foydalanish xususiyatlari qanday?
- AskNoTracking usuli asosida ma'lumotlar qanday olinadi?
- IEnumerable va IQueryable usullari o'rtasidagi farqni ko'rsatish uchun misollar keltiring.
- AskNoTracking usulidan foydalanishning afzalliklari nimada?

5. ENTITY FRAMEWORK DA SQL. SQL BILAN ISHLASH

Reja:

1. **SQL bilan ishlash.**
2. **Saqlangan funksiyalar.**
3. **Saqlangan protseduralar.**
4. **Nazorat savollari.**

Ko'p hollarda dasturchilar effektiv so'rovlarni **LINQ** operatorlari va metodlari yordamida amalga oshirish mumkin. Ammo **Entity Framework** da ushbu **sql**-so'rovlarni bajarishning o'ziga xos usullari mavjud.

Ushbu **sql**-so'rovlarni **DB**da amalga oshirish uchun ma'lumotlar kontekstidagi **Database** xususiyatidan foydalanish mumkin. Ushbu xususiyat orqali ma'lumotlar bazasi, ulanish haqida ma'lumot olish va **DB**ga turli so'rovlarni amalga oshirish mumkin. Masalan, ulanish satri haqidagi ma'lumotga ega bo'lish uchun:

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

So'rovni hosil qilish uchun **sql**-ifodani parametr sifatida qabul qiluvchi **SqlQuery** metodini ishlatish lozim. Misol sifatida avvalgi mavzuda ko'rib chiqilgan **DB**ni ko'rib chiqamiz. Bunda quyidagi model va ma'lumotlar kontekstlari tavsiflangan:

```
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; }

}
```

Companies jadvaldagi barcha modellarni olamiz:

```
static void Main(string[] args)
{

using (PhoneContext db = new PhoneContext())
{

        var comps = db.Database.SqlQuery<Company>("SELECT * FROM
Companies");

foreach (var company in comps)
Console.WriteLine(company.Name);

}
Console.ReadLine();
}
```

SELECT ifodasi orqali jadvaldagi ma'lumotlar olinadi. Yuqoridagi misolda ushbu jadval **Company** modeliga mos qo'yilgan bo'lib, ushbu modelga mos bo'lgan ob'ektlarni o'zida saqlaydi va ushbu chaqiruv **Company** klassi orqali tiplashtiriladi db.Database.SqlQuery<Company>().

SqlQuery() metodining boshqa versiyasi parametrlarni ishlatishni ta'minlaydi. Masalan, **DB**dan **"Samsung"** satrini o'zida saqlovchi barcha modellarni tanlash lozim bo'lsin:

```
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);
}
```

System.Data.SqlClient nomlar fazosidagi **SqlParameter** klassi orqali **sql** so'rovda ishlatiladigan parametrni uzatish mumkin.

SqlQuery() metodi orqali **DB**dan tanlash amali bajariladi. Ammo ba'zi hollarda **DB**da o'chirish, yangilash va yangi yozuv qo'shish kabi amallarni bajarishimizga to'g'ri keladi. Ushbu maqsadda **ExecuteSqlCommand()** metodidan foydalaniladi. Ushbu metodni quyidagicha qo'llashimiz mumkin:

```
using (PhoneContext db = new PhoneContext())
{
// yangi yozuv

        int numberOfRowInserted = db.Database.ExecuteSqlCommand("INSERT
INTO Companies (Name) VALUES ('HTC')");

// o'zgartirish
        int numberOfRowUpdated = db.Database.ExecuteSqlCommand("UPDATE
Companies SET Name='Nokia' WHERE Id=3");
// o'chirish

        int numberOfRowDeleted = db.Database.ExecuteSqlCommand("DELETE
FROM Companies WHERE Id=3");

}
```

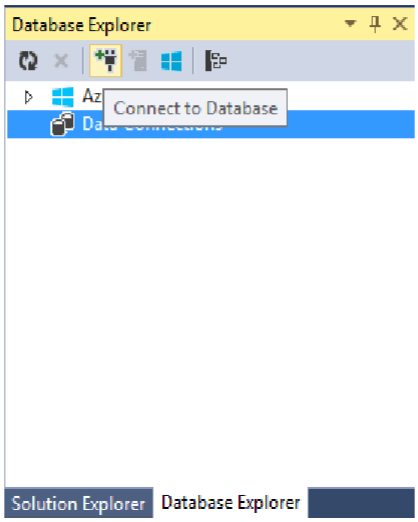
Foydalanuvchi funksiyalari

Sql-so'rovlarlar bilan ishlashda saqlanadigan protseduralar va foydalanuvchi funksiyalari alohida ahamiyatga ega. **C#** tilida **Sql Server** ning foydalanuvchi funksiyasini chaqirishni ko'rib chiqamiz.

Avvalo funksiyani hosil qilamiz. Bizning ma'lumotlar kontekstimizda quyidagi modellar aniqlangan bo'lsin:

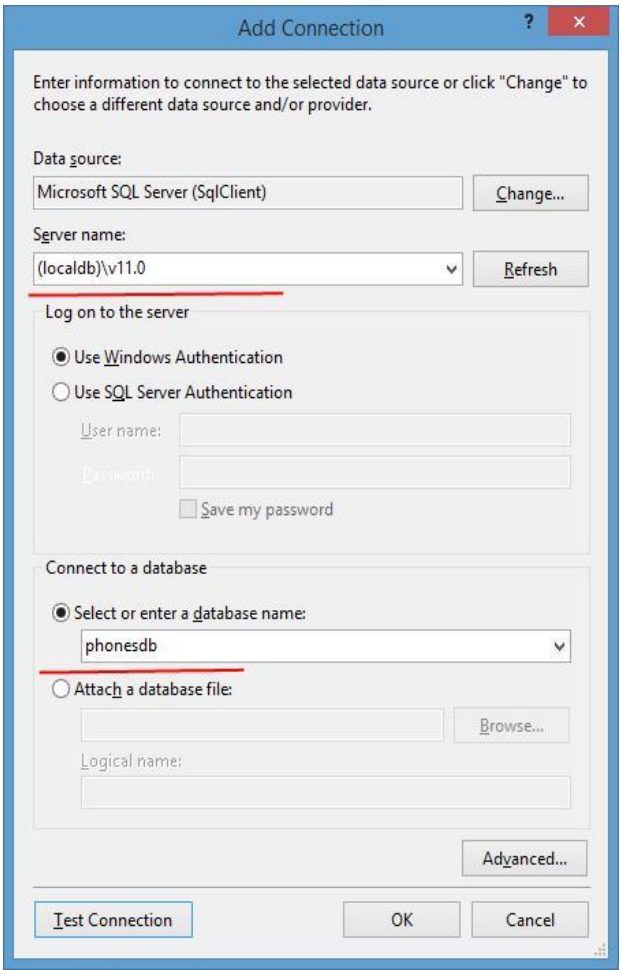
```
class PhoneContext : DbContext
{
public PhoneContext()
        base("DBConnection")
{ }

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

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

public string Name { get; set; }

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

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

public int Price { get; set; }

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

Endi **Visual Studio** muhitining **Database Explorer** oynasidan **DB**ni ochib olamiz. Buning uchun **Database Explorer** oynasidan **Connect to Database** tugmasini bosamiz:



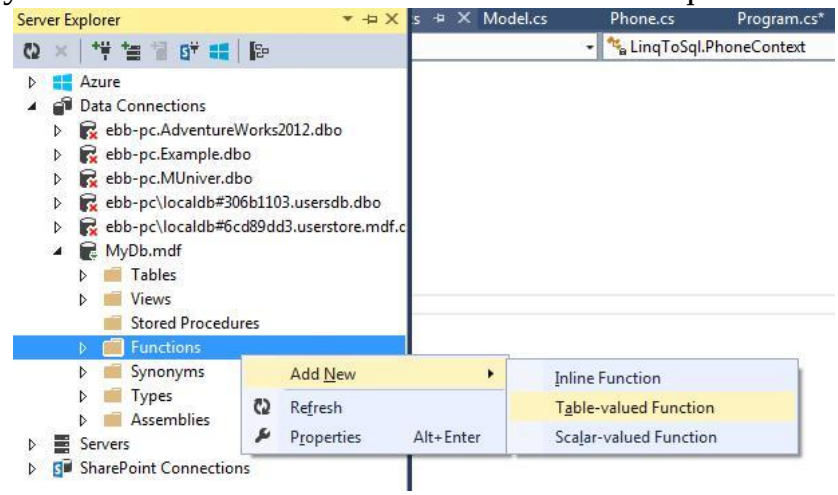
71-Rasm

Ushbu ro'yxatdan zarur **DB**ni tanlaymiz. Bizning misolda **DB** sifatida **phonesdb** ishlatiladi.



72-Rasm

DB ochilganidan so'ng **Database Explorer** oynadan **Functions** qismini topamiz va unga sichqonchaning o'ng tugmasini bosish lozim. Hosil qilingan kontekstli menyudan **Add New -> Table-valued Function** qismini tanlash lozim:



73-Rasm

So'ngra, **Visual Studio** quyidagi kodga ega bo'lgan skriptni generatsiya qiladi:

```
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
```

Ushbu skript (funksiya)ni quyidagicha o'zgartirib olamiz:

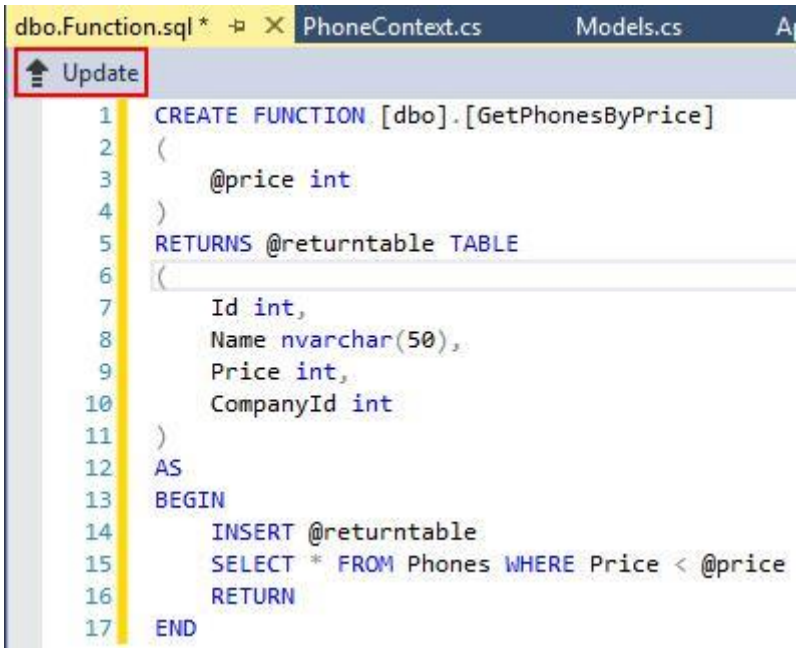
```
CREATE FUNCTION [dbo].[GetPhonesByPrice]
(
@price int
)
RETURNS @returntable TABLE

(
```

```
Id int,
Name nvarchar(50),
Price int,
CompanyId int
)
AS
BEGIN

INSERT @returntable
SELECT * FROM Phones WHERE Price < @price
RETURN
END
```

CREATE FUNCTION [dbo].[GetPhonesByPrice] ifodasi orqali **GetPhonesByPrice** funksiyasi hosil qilinadi. Ushbu yozuvdan so'ng parametrlar ro'yxati keltiriladi. Bizning funksiyamiz faqat bitta **int** tipiga mansub **@price** parametrini qabul qiladi.

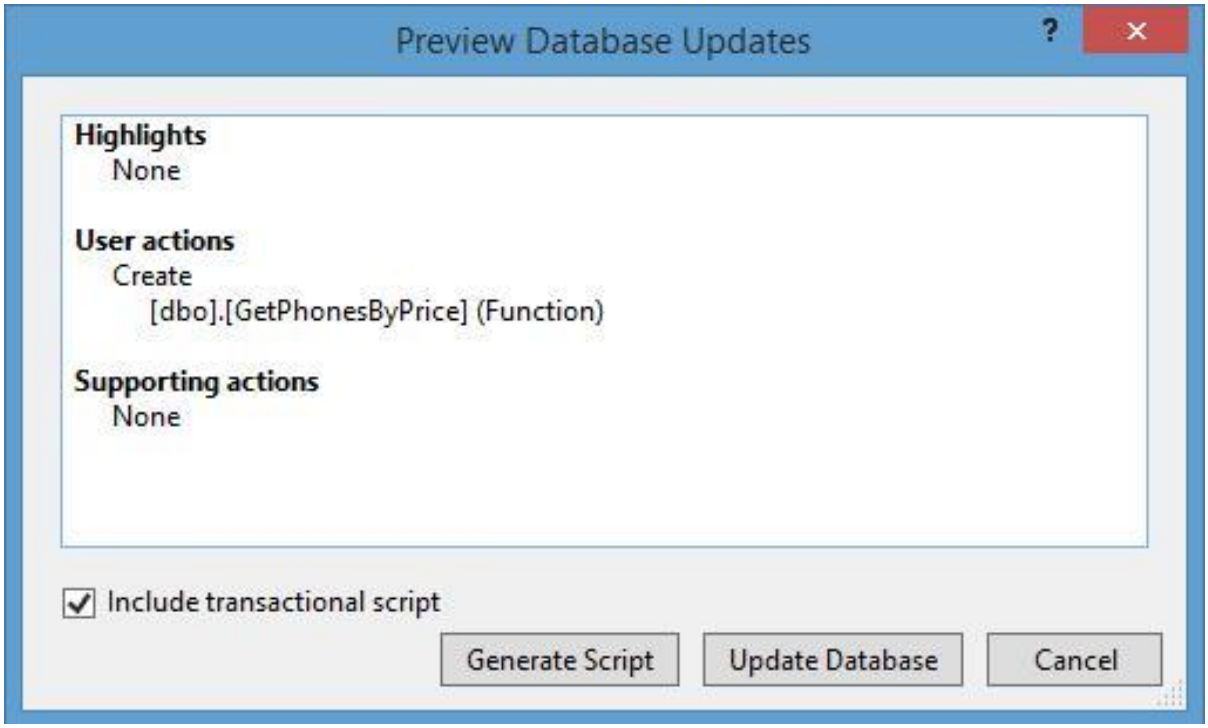
So'ngra funksiya qaytaradigan natija RETURNS @returntable TABLE ifodasi orqali aniqlangan. Qavslar ichida qaytariluvchi jadval o'zining ustunlari bilan aniqlangan. Bizning misolda ushbu natija tipi **Phones** jadvaliga mos tuzilmaga ega. Ya'ni ushbu jadval **Phone** klassiga mansub ob'ektlarni o'zida saqlaydi.

BEGIN i **END** ifodasi orasida funksiya mazmuni joylashtiriladi. Bizning misolda **WHERE** operatori orqali **Price** ustuni qiymati **@price** dan kichik bo'lgan yozuvlardan iborat. Endi ushbu funksiyani **DB**da shakllantiramiz. Buning uchun **Update** tugmasini bosamiz:



74-Rasm

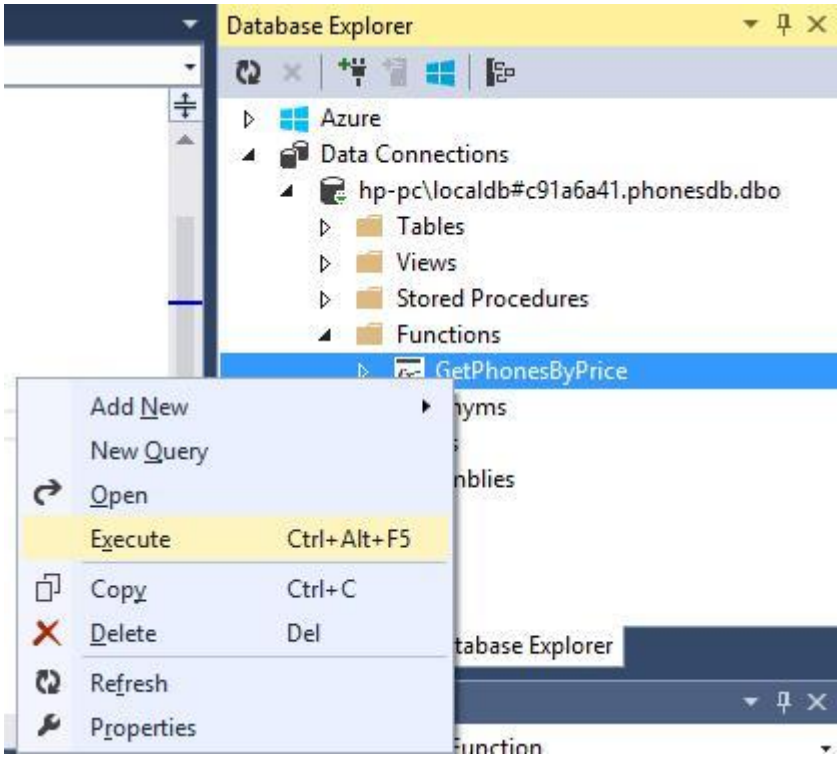
So’ngra, hosil qilingan muloqot oynasidagi **Update Database** tugmasini bosamiz:



75-Rasm

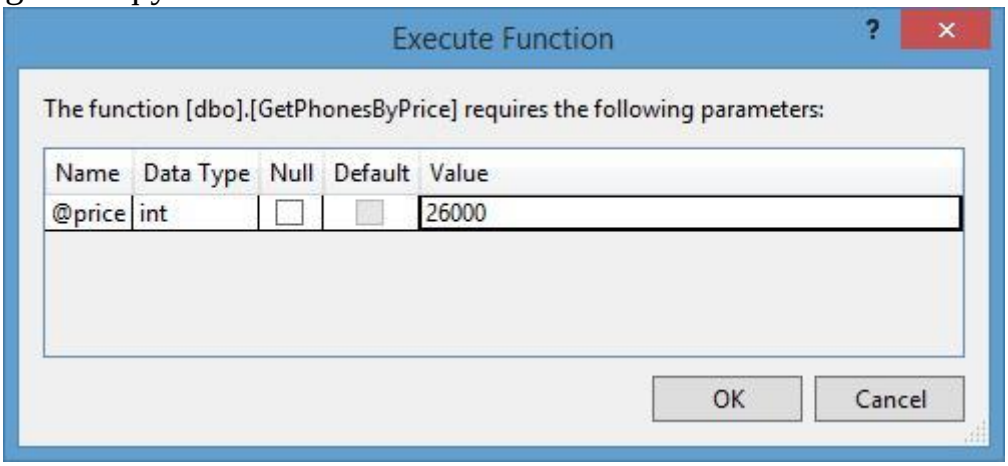
Shundan so’ng, **Database Explorer** oynasidagi **Functions** qismini topib, yangi hosil qilingan funksiyamizni ko’rishimiz mumkin. Endi biz ushbu funksiyani ishlatishimiz mumkin. Ushbu funksiyaga **C#** tilidan murojaat qilishdan avval, uning to’g‘ri ishlashini tekshiruvdan o'tkazamiz.

Buning uchun ushbu funksiya nomiga sichqonchaning o'ng tugmasini bosamiz va hosil bo'lgan menyudan **Execute** qismini tanlaymiz:



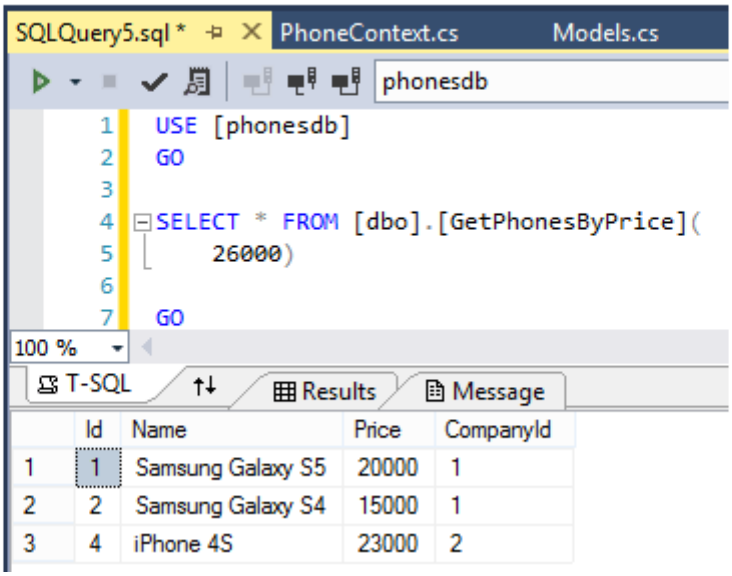
76-Rasm

Shundan so'ng, funksiyaning kiruvchi parametrlari qiymatini kiritish oynasi shakllantiriladi. Ushbu muloqot oynasidagi **Value** maydonga funksiyaning **@price** parametriga mos qiymatni kiritish lozim:



77-Rasm

Shundan so'ng, **Visual Studio** funksiya chaqirish skriptini generatsiya qiladi:



78-Rasm

Yuqoridagi natijadan biz yaratgan funksiyaning to'g'ri ishlashini anglash mumkin. Endi ushbu funksiyaga **C#** kodidan murojaat qilamiz:

```
static void Main(string[] args)
{
using (PhoneContext db = new PhoneContext())
{

        System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@price", 26000);

        var phones = db.Database.SqlQuery<Phone>("SELECT * FROM
GetPhonesByPrice (@price)", param);

foreach (var phone in phones)
Console.WriteLine(phone.Name);
}
Console.ReadKey();

}
```

Ushbu dasturda **SQL Server** da joylashgan, PhoneContext ma'lumotlar kontekstiga mos **DB**dagi **GetPhonesByPrice** funksiyasi (@price) parametri orqali chaqirilgan.

Xuddi yuqoridagi tartibda chegirmaga mos narxni aniqlovchi funksiyani shakllantiramiz:

```
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
```

Ushbu funksiya parametr sifatida chegirma foizini **(masalan, 10%)** qabul qiladi. Natija sifatida esa ikkita maydondan iborat jadvalni qaytaradi. Ushbu maydonlar **model nomi** va chegirma inobatga olingan **narx**.

Funksiya yangi **Name** va **Price** xususiyatlardan iborat ob'ekt qaytarganligi, shuningdek **Price** xususiyatning decimal tipiga mansubligi sababli **C#** tilida mos klassni shakllantiramiz:

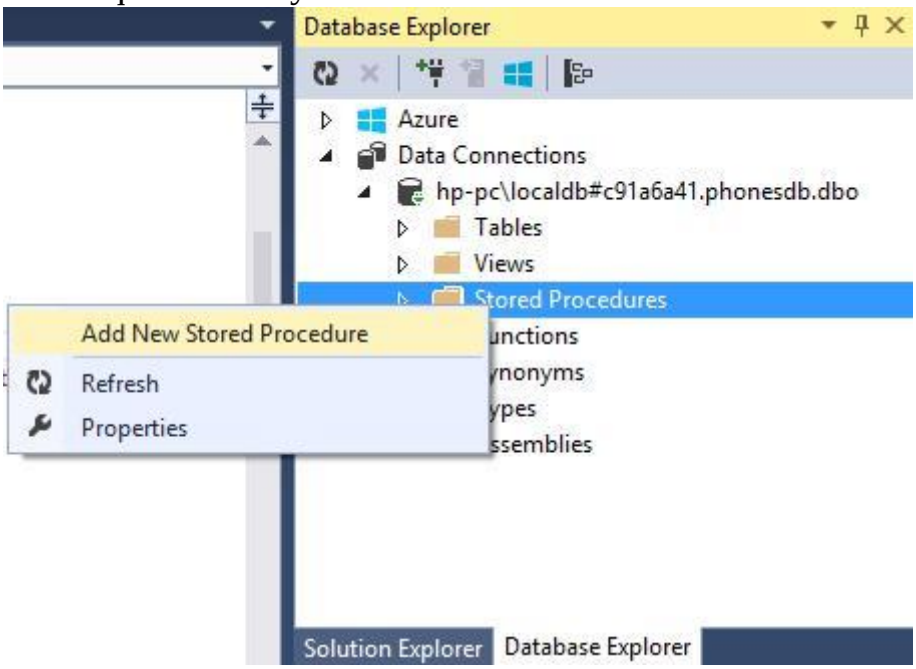
```
class DiscountPhone
{
public string Name { get; set; }
public decimal Price { get; set; }
}
```

Endilikda funksiya natijasini quyidagicha olishimiz mumkin:

```
static void Main(string[] args)
{
using (PhoneContext db = new PhoneContext())
{
// 15% li chegirma

        System.Data.SqlClient.SqlParameter param = new
System.Data.SqlClient.SqlParameter("@discount", 15);

        var phones = db.Database.SqlQuery<DiscountPhone>("SELECT * FROM
GetPriceWithDiscount (@discount)", param);

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

Saqlanadigan prodseduralar

DBda saqlanadigan protseduralar bilan ishlash foydalanuvchi funksiyalari kabi amalga oshiraladi. Avvalgi mavzuda ko'rib chiqilgan **DB** ustida ish ko'ramiz. Buning uchun **Database Explorer** oynasidagi mos **DB** ga ulanamiz va tuzilmadan **Stored Procedures** (Saqlangan protseduralar) qismini tanlaymiz. Ushbu qism ustiga sichqonchaning o'ng tugmasini bosamiz va taqdim qilingan menyudan **Add New Stored Procedure** qismni tanlaymiz:



79-Rasm

Shundan so'ng **Visual Studio** quyidagi protsedura kodini generatsiya qiladi:

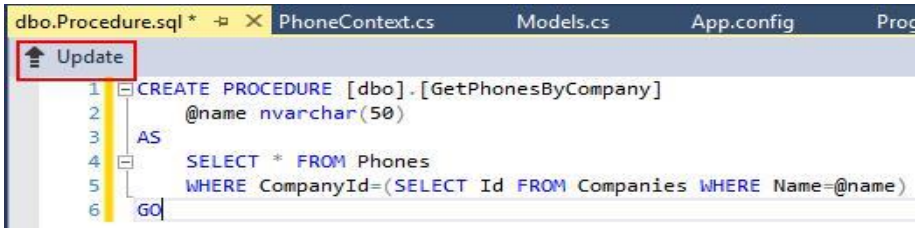
```
CREATE PROCEDURE [dbo].[Procedure] @param1 int = 0,

    @param2 int
    AS
    SELECT @param1, @param2

    RETURN 0
```

Ushbu kodni quyidagicha o'zgartirib olamiz:

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

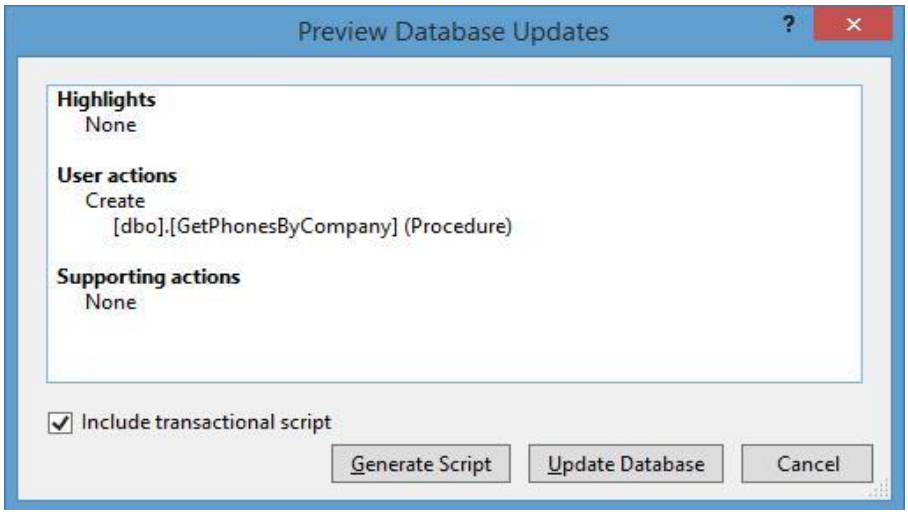
Ushbu protsedura **@name** parametr ga teng bo'lgan telefon kompaniyalari nomiga teng bo'lgan satrlarni topib bo'ladi.

Shundan so’ng, saqlanadigan protsedurani xotiraga saqlash uchun **Update** tugmasini bosamiz:



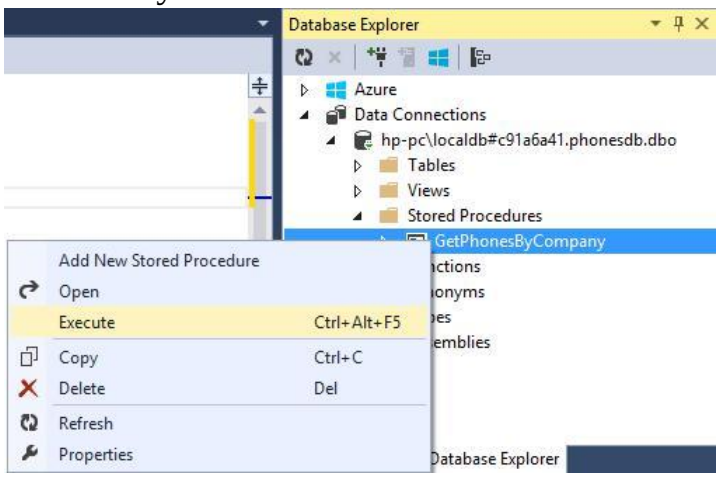
80-Rasm

Shundan so’ng hosil qilingan oynadan **Update Database** tugmasini bosish lozim:



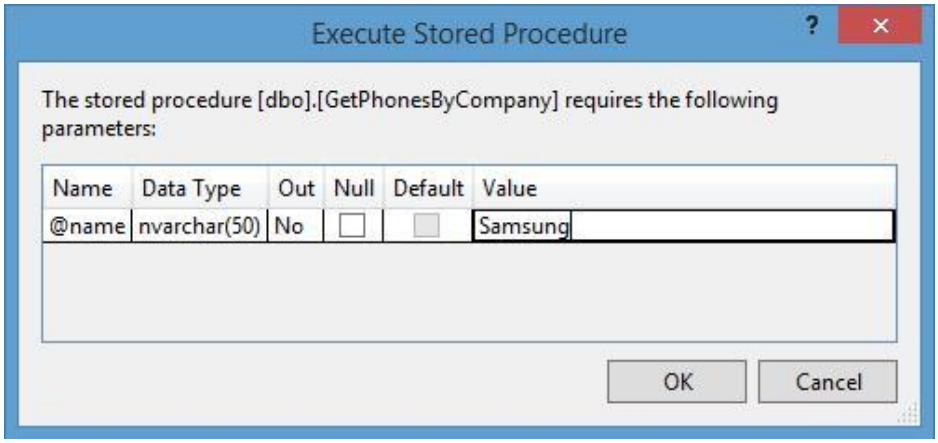
81-Rasm

Shundan so’ng, **Stored Procedures** qismda yangi protsedura hosil qilinadi. Ushbu protsedurani ishlatishdan avval uning to’g‘ri ishlashini tekshiramiz. Buning uchun protsedura nomi ustiga sichqonchaning o’ng tugmasini bosib, taklif qilingan menyudan **Execute** qismni tanlaymiz:



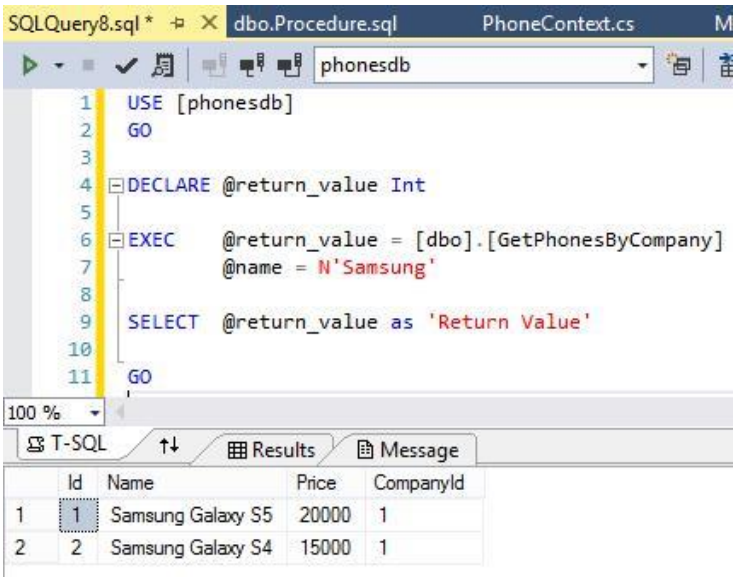
82-Rasm

Shundan so’ng kiruvchi parametri mavjud bo’lgan oyna taqdim etiladi. Ushbu oynadagi **Value** maydoniga biror qiymatni kiritamiz va **OK** tugmasini bosamiz:



83-Rasm

Shundan so’ng **Visual Studio** avtomatik tarzda **SQL** tilida mos skriptni generatsiya qiladi va protsedurani kiritilgan parametr orqali ishga tushiradi:



84-Rasm

Xuddi yuqorida keltirilgan tartibda **GetPhonesPrice** saqlanuvchi protsedurasini yaratamiz:

```
CREATE PROCEDURE [dbo].[GetPhonesPrice] @bprice int, @eprice int
  AS

  SELECT * FROM Phones
  WHERE Price BETWEEN @bprice and @eprice
```

Endi **C#** tili ushbu protsedura bilan ishlashni ko’rib chiqamiz:

```
static void Main(string[] args)
{
using (PhoneContext db = new PhoneContext())
{
==              Kompaniya nomiga mos telefonlar markalarini aniqlash
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);

\{              Narxi 20000 va 35000 rubl orasida bo'lgan telefon markalarini aniqlash
System.Data.SqlClient.SqlParameter param1 = new

System.Data.SqlClient.SqlParameter("@bprice", "20000");
System.Data.SqlClient.SqlParameter param2 = new
System.Data.SqlClient.SqlParameter("@eprice", "50000");

            var phones1 = db.Database.SqlQuery<Phone>("GetPhonesPrice @bprice,
@eprice", param1, param2);
foreach (var p in phones1)
Console.WriteLine("{0} - {1}", p.Name, p.Price);
}
Console.ReadKey();
}
```

SqlQuery metodidagi parametr sifatida protsedura nomini qabul qiladi. So'ngra ushbu protsedurada mavjud parametrlar ro'yxat keltiriladi:

```
GetPhonesByCompany @name
```

Ushbu protsedura **Phones** jadvalidagi satrlarni qaytarganligi sababli, **SqlQuery** metodini klass orqali tiplashtirishimiz mumkin:

```
db.Database.SqlQuery<Phone>("GetPhonesByCompany @name", param);
```

Nazorat savollari:

// LINQ usullari va operatorlari asosida SQL so'rovlarini yaratish tartibini aytib bering.

// SQL so'rovlari bilan ishlashda saqlanadigan funksiyalar va protseduralar qanday ta'minlanadi?

// SQL so'rovlari bilan ishlashda saqlangan funksiya va protseduralarni ta'minlashga misol keltiring.

// LINQ usullari va operatorlari asosida ma'lumotlar bazasiga qanday ulanish mumkin?

6. FLUENT API VA ANNOTATSIYA

Reja:

1. **Fluent API qoidalari.**
2. **Fluent API modellari o'rtasidagi munosabat.**
3. **Fluent API izohlari**
4. **4.Nazorat savollari**

Agar loyihamizda biz **Code First** yondashuvidan foydalansak, modellarga mos klasslar **DB**dagi jadvallarga **Entity Framework**dagi bir qator qoidalar asosida mos qo'yiladi. Ammo ba'zi hollarda ushbu qoidalarni o'zgartirishga yoki qayta ishlashga to'g'ri keladi. Buning uchun **Fluent API** va ma'lumotlar annotatsiyasidan foydalanamiz.

Fluent API

Fluent API metodlar to'plamidan iborat bo'lib, ular orqali klass va uning xususiyatlari **DB** jadvallari va ustunlari bilan mos qo'yiladi.

Fluent API funksionali **OnModelCreating** metodini qayta aniqlash orqali amalga oshiriladi:

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

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

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
// Fluent API dan foydalanish

base.OnModelCreating(modelBuilder);
}
}
```

Ekspremental model sifatida quyidagi modeldan foydalanamiz:

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

Klassni jadval bilan mos qo'yish

EF boshlang'ich holda modelni nomiga mos jadval bilan moslashtiradi. Ammo biz **ToTable()** metodi orqali ushbu moslikni o'zgartirishimiz mumkin:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
modelBuilder.Entity<Phone>().ToTable("Mobiles");
//            Fluent API дан фойдаланиш

base.OnModelCreating(modelBuilder);
}
```

Endi **Phone** klassiga mos barcha ob'ektlar **Mobiles** jadvalida saqlanadi. Shuningdek, ushbu ob'ektlar bilan **db.Phones** xususiyati orqali ishlash imkoniyati mavjud.

Agar biror ob'ektni jadvalda shakllantirish lozim bo'lmasa, **Ignore()** metodi orqali amalni bekor qilishimiz mumkin:

```
modelBuilder.Ignore<Company>();
```

Birlamchi kalitni qayta aniqlash

Entity Framework boshlang'ich holda birlamchi kalitni **Id** yoki model nomiga **mos [Klass_nomi]Id** kabi shakllantiradi. Misol: **PhoneId**.

Fluent API orqali birlamchi kalitni qayta aniqlash uchun **HasKey()** metodidan foydalaniladi.

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

Ushbu holda birlamchi kalit sifatida **Phone** klassining **Ident** xususiyati tushuniladi. Agar birlamchi kalitlar sifatida ikkita maydon mos bo'lsa, u holda quyidagi koddan foydalaniladi:

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

Yuqoridagi modellarga mos dastur kodi quyidagicha:

App.config:

```
<?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="DBConnect" connectionString="data source=.;initial
catalog=MyData;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
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>
```

Phone.cs:

```
class Phone
```

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

FluentContext.cs:

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

using System.Text;
using System.Data.Entity;

namespace Fluent
{
class FluentContext : DbContext

{
public FluentContext()
:           base("DBConnect")
{ }

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

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
modelBuilder.Entity<Phone>().ToTable("Mobiles");
modelBuilder.Entity<Phone>().HasKey(p => p.Ident);
// Fluent API dan foydalanish
base.OnModelCreating(modelBuilder);

}
}
}
```

Program.cs:

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

namespace Fluent
```

```
{
class Program
{
static void Main(string[] args)
{
using (FluentContext db = new FluentContext())
{

//              Samsung markali va narxi 650000 so'mdan kam telefonlar ro'yxati var
phones = db.Phones.Where(p => p.Price < 650000)

.Union(db.Phones.Where(p => p.Name.Contains("Samsung")));

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

//              so'rovni amalga oshirish

var tels = db.Database.SqlQuery<Phone>("SELECT * FROM Mobiles"); foreach
(var phone in tels)

Console.WriteLine(phone.Name);
}
Console.ReadKey();
}

}
}
```

Xususiyatlarni moslashtirish

Klass xususiyatini muayyan ustun bilan mos qo'yish uchun **HasColumnName()** metodidan foydalaniladi::

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

Ushbu holda klassning **Name** xususiyati bilan jadvalning **PhoneName** ustuni mos qo'yilgan. Agar biror xususiyat bilan jadvalning biror ustuni **ususman** mos qo'yilmasligi lozim bo'lsa, **Ignore()** metodidan foydalaniladi:

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

Endi **Phone** klassidagi **Discount** xususiyati **DB**dagi jadalning biror ustuni bilan mos qo'yilmaydi.

DBdagi jadval maydoni **NULL** qiymatni qabul qilishi mumkin. **Code First** yondashuviga ko'ra barcha ustunlar agar ma'lumotlar annotatsiyasi qo'llanilmagan bo'lsa, **NULL** qiymat qabul qilish mumkin. Ammo **IsRequired()** metodi orqali muayyan ustun qiymat qabul qilishi shartligini ko'rsatishimiz mumkin:

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

Agar bizda biror ustun **NULL** qiymat qabul qilish mumkin bo'lsa, **IsOptional()** metodidan foydalanishimiz mumkin:

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

Satrlarni sozlash

Klass xususiyati satr tipga mansub bo'lsa, uning uzunligini **HasMaxLength()** metodi orqali ko'rsatish mumkin. Masalan, xususiyat satr tipga mansub bo'lib, uzunligi **50** dan oshmasligi lozim bo'lsa:

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

Shuningdek, satrni aniqlashda uning **Unicode** kodirovkasida qiymat qabul qilish mumkinligi ham ko'rsatish mumkin:

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

Ushbu misolda **false** parametri barcha satrlarning **Unicode**-kodirovkada saqlanishini anglatadi.

decimal sonlarni sozlash

Agar klassda **decimal** tipga mansub xususiyat mavjud bo'lsa, ushbu xususiyat qabul qiladigan sondagi raqamlar aniqligi va verguldan keyingi raqamlar sonini ko'rsatish mumkin:

```
// Price xususiyati – decimal bo'lsin
modelBuilder.Entity<Phone>().Property(p => p.Price).HasPrecision(15, 2);
```

Endi **decimal** soni **15** ta raqam va verguldan keyin **2** ta raqamdan iborat bo'lishi mumkin. Agar **decimal** soni parametrlari ko'rsatilmasa, boshlang'ich holatda **18** ta raqam va verguldan keyin **2** ta raqamdan iborat bo'ladi.

Ustun tipini sozlash

EF DBdagi jadval ustuniga mos xususiyatga boshlang'ich tipni o'zi tanlaydi. Ammo ustun uchun **DB**da mos tipni **HasColumnType()** metodi orqali ko'rsatish mumkin:

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

FluentContext.cs

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

namespace Fluent

{
class FluentContext : DbContext
{
public FluentContext()
:           base("DBConnect")
{ }

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

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{

modelBuilder.Entity<Phone>().ToTable("Mobiles");
modelBuilder.Entity<Phone>().HasKey(p => p.Ident);
modelBuilder.Entity<Phone>().Property(p =>
p.Name).HasColumnName("PhoneName");
modelBuilder.Entity<Phone>().Property(p => p.Name).HasMaxLength(50);
modelBuilder.Entity<Phone>().Property(p =>
p.Name).HasColumnType("varchar");
modelBuilder.Entity<Phone>().Property(p => p.Price).HasPrecision(15, 2);
// Fluent APIdan foydalanish
base.OnModelCreating(modelBuilder);
}
}
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Data.Entity;
namespace Fluent
{
class Program
```

```
{
static void Main(string[] args)
{
using (FluentContext db = new FluentContext())
{

//              Samsung markali va narxi 650000 so'mdan kam telefonlar ro'yxati 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("{0} - {1}", item.Name, item.Price);
}
Console.ReadKey();
}
}
}
```

Modelni bir qancha jadvallarga mos qo'yish

Fluent API yordamida modelning bir nechta xususiyatlarini bitta jadvalga, boshqa xusuisiyatlarini boshqa jadval ustunlariga mos qo'yish mumkin:

```
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");

});
```

Yuqoridagi usul orqali **Name** xususiyati **Mobiles** jadvalida saqlanadi. **Price** va **Discount** xususiyatlari esa **MobilesInfo** jadvalida saqlanadi. Identifikator ustuni esa har ikkala jadval uchun umumiy hisoblanadi.

Model va Fluent API o'rtasidagi munosabat

Birga nol yoki bir bog'lanish (One-to–Zero-or-One)

Ushbu turdagi bog'lanish orqali tashkil qilingan modelda bir modelga mos ikkinchi model bo'lishi shart emas. Masalan:

```
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; }

}
```

Ushbu klasslar asosida tashkil qilingan modelda telefon albatta o'zining ishlab chiqargan kompaniyasi haqidagi ma'lumotlarni o'zida saqlaydi. Ya'ni ushbu holda aloqa **birga - nol yoki ko'p bog'lanish** kabi tashkil qilingan. Ushbu model **Fluent API** da quyidagicha tashkil qilinadi:

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

modelBuilder.Entity<Phone>()
.HasRequired(c => c.Company)
.WithOptional(c => c.BestSeller);

base.OnModelCreating(modelBuilder);
}
```

Ushbu kodda **HasRequired()** metodi **Phone** ob'ektning **Company** xususiyati albatta mavjud bo'lishi lozimligini anglatadi. **WithOptional()** metodi esa avvalgi ifodadagi **Company** va uning **BestSeller** xususiyati bilan aloqa shart emasligini anglatadi.

Quyida ushbu **birga - nol yoki ko'p bog'lanish** kabi tashkil qilingan model elementlari ustida so'rov amalga oshirilgan:

```
using (FluentContext db = new FluentContext())
{

// Telefon va u ishlab chikilgan kompaniya xakidagi ma'lumotni chikarish
IEnumerable<Phone> ps = db.Phones.Include(p => p.Company); foreach (var item in
ps)
```

```
Console.WriteLine("{0} - {1} ", item.Name, item.Company.Name);
}
```

Birga bir aloqa (One-to-One)

Ushbu aloqa orqali tashkil qilingan modelda har ikkala ob'ektda bir-biriga aloqa mavjud bo'lishi lozim:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
modelBuilder.Entity<Phone>()

.HasRequired(c => c.Company)
.WithRequiredPrincipal(c => c.BestSeller);
//yoki
//modelBuilder.Entity<Company>()
//              .HasRequired(c => c.BestSeller)
//              .WithRequiredPrincipal(c => c.Company);

base.OnModelCreating(modelBuilder);
}
```

WithRequiredPrincipal() metodi orqali majburiy aloqa va bitta ob'ekt asosiy sifatida o'rnatiladi. Ushbu holda asosiy ob'ekt sifatida **Phone** modeli **Phone**:

WithRequiredPrincipal(c => c.BestSeller) kabi ko'rsatilgan.

Company modeli namoyish qilinadigan jadval esa **Phones** jadvaliga ikkilamchi kalitdan iborat bo'ladi.

Ko'pga-ko'p aloqa (many-to-many)

Yuqoridagi modelga o'zgartirish kiritamiz. Bizdagi har ikkala model boshqa modellar ro'yxatini o'zida saqlasin. Masalan, kompaniyada bir qancha turdagi telefonlar ishlab chiqarilishi va bitta telefon bir nechta kompaniyalar hamkorligida ishlab chiqilsin:

```
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>();
```

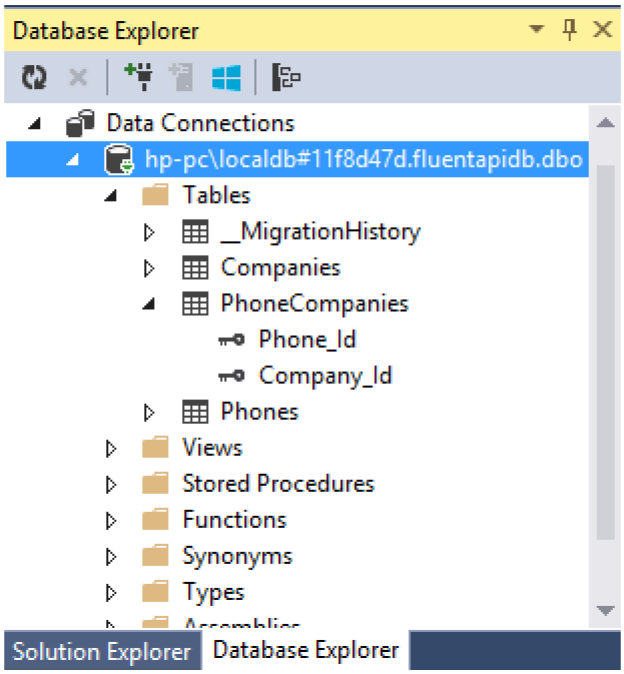
```
}
}

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>();

}
}
```

Ushbu modelga mos aloqa quyidagicha shakllantiriladi:

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

HasMany() metodi orqali **Phone** va **Company** ob'ektlari o'rtasida ko'pga-ko'p aloqa o'rnatiladi. **WithMany()** metodi esa **Phone** va **Company** ob'ektlari o'rtasida teskari ko'pga-ko'p aloqa o'rnatiladi.

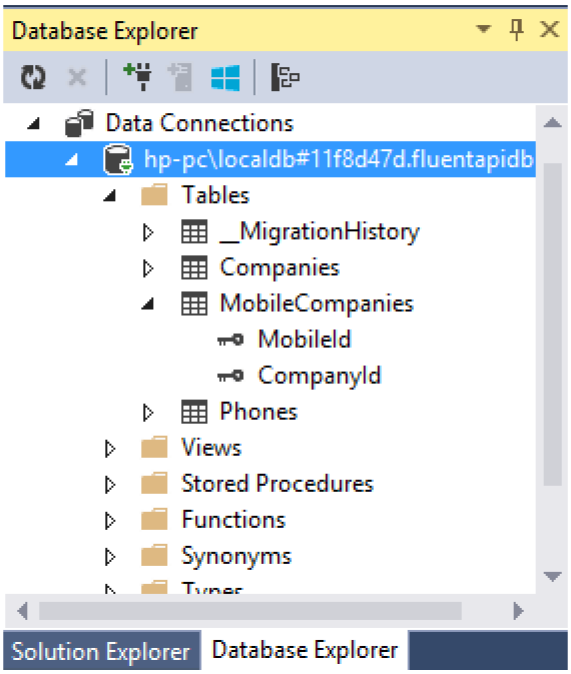
Yuqoridagi turdagi aloqaga mos ma'lumotlar bazasida uchinchi bog'lovchi jadval shakllantiriladi:



85-Rasm

Ammo bizni ushbu turdagi bog‘lovchi jadval va ustunlari nomi qanoatlantirmaydi. Shuning uchun quyidagicha moslashtirishlarni amalga oshiramiz:

```
modelBuilder.Entity<Phone>()
.HasMany(p => p.Companies)
.WithMany(c => c.Phones)
.Map(m =>
{

m.ToTable("MobileCcompanies");
m.MapLeftKey("PhoneId");
m.MapRightKey("CompanyId");
});
```



86-Rasm

Yuqorida keltirilgan modelaga mos quyidagi so'rovni amalga oshirishimiz mumkin:

```
using (FluentContext db = new FluentContext())
{
var items = from i in db.Phones
join fi in db.Companies on i.Id equals fi.Id
where fi.Id == 1

select i;
foreach (var item in items)
{
Console.WriteLine(item.Name);
}
}
```

Birga ko'p bog'lanish (One-to-Many)

Birga – ko'p tarzda tashkil qilingan bog'lanishda bitta model boshqa modellar to'plamiga uzatmaga ega bo'ladi. Misol sifatida quyidagi modelni ko'rib chiqamiz. Bir kompaniya bir nechta turdagi telefonlarni ishlab chiqsin.

```
public class Phone
{

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

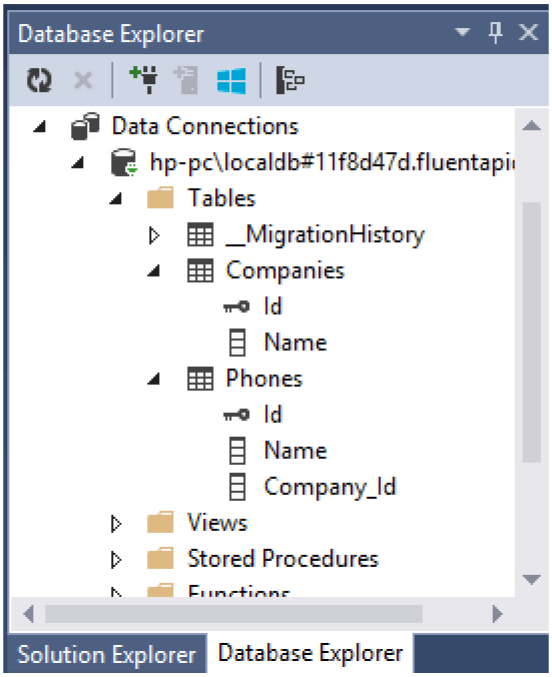
```
public Company Company { get; set; }
}

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>();
}

}
```

Fluent API orqali aloqa quyidagicha shakllantiriladi:

```
modelBuilder.Entity<Company>()

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

HasMany() metodi orqali **Company** va **Phone** ob'ektlari o'rtasida birga ko'p aloqa o'rnatiladi. **WithRequired()** metodi esa **Phone** klassining **Company** xususiyati qiymati bo'lishi shartligi talab qilinadi.

Ushbu bog'lanishga ega bo'lgan modellarga **Phone** modeli o'zida **Companies** jadvali bilan bog'lanishni ta'minlovchi **Company_**Id tashqi kalitni o'zida saqlaydi:



87-Rasm

Tashqi kalitni sozlash

EF da tashqi kalit va ustun nomi bizni qanoatlantirmasa, **HasForeignKey()** metodi orqali ushbu qiymatlarni qayta aniqlash mumkin. Buning uchun quyidagi koddan foydalanish mumkin:

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

public string Name { get; set; }

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

Endi **Companies** jadvaliga tashqi kalitni ifodalovchi **Manufacturer** xususiyatini sozlaymiz:

```
modelBuilder.Entity<Company>()

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

Kaskadli o’chirishni bekor qilish

Agar tashqi kalit **NOT NULL (Nullable)** qiymatga ega bo’lishi mumkin bo’lsa, bosh element o’chirilishi natijasida u bilan bog‘liq boshqa jadvallarda kaskadli o’chirish amalga oshiriladi.

Yuqorida keltirilgan misolda **Manufacturer** xususiyati tashqi kalit vazifasini bajaradi va u **int** tipiga mansub. Shuning uchun jadval generatsiya qilinayotgan vaqtda **ON DELETE CASCADE** qoidasiga amal qilinadi.

Agar bizda **Manufacturer** xususiyati **int**? kabi aniqlangan bo’lsa, **WithRequired** metodi o’rnida **WithOptional()** metodidan foydalaniladi:

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

Ushbu turda tashkil qilingan aloqada tashqi kalit zarurligi talab etilmaydi.

Fluent API da kaskadli o’chirishni olib tashlash uchun **WillCascadeOnDelete(false)** metodidan foydalanish lozim:

```
modelBuilder.Entity<Company>()

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

WillCascadeOnDelete(true) metodi orqali kaskadli o’chirish ta'minlanadi.

Shuningdek, kaskadli o’chirishni quyidagi tarzda amalga oshirish mumkin:

```
using System.Data.Entity.ModelConfiguration.Conventions;

.....................................................

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();
```

Annotatsiyalar

Annotatsiyalar orqali modellar va jadvallar o'rtasidagi moslik atributlar yordamida ta'minlanadi. Annotatsiyalarni ta'minlovchi ko’pgina klasslar System**.ComponentModel.DataAnnotations** nomlar fazosida joylashgan.

Kalitni sozlash

Muayyan xususiyatning kalit ekanligini o’rnatish uchun **[Key]** atributidan foydalanish lozim:

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

Endilikda **Ident** xususiyati birlamchi kalit sifatida qabul qilinadi. Kalitni identifikator sifatida o'rnatish uchun **DatabaseGenerated(DatabaseGeneratedOption.Identity)** atributidan foydalaniladi:

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

Agar xususiyat **Key** atributi bilan belgilangan bo'lsa va tipiga **int** mansub bo'lsa, unga **DatabaseGenerated(DatabaseGeneratedOption.Identity)** atributini qo'llash shart hisoblanmaydi.

Required atributi

Required atributi orqali xususiyatning qiymatga ega bo'lishi shartligi ko'rsatiladi. Ushbu atribut orqali shakllantirilgan atribut **DB**dagi mos ustunga **NOT NULL** sharti qo'yilganligini anglatadi.

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

[Required]
public string Name { get; set; }
}
```

Endilikda **Phone** ob'ektining **Name** xususiyatiga qiymat berilmagan holda **DB**da saqlansa, xatolik yuzaga keladi.

MaxLength va MinLength

MaxLength va **MinLength** atributlari orqali satr tipiga mansub xususiyatning uzunligi ko'rsatiladi:

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

NotMapped atributi

Modelning barcha **public** xususiyatlari jadvaldagi muayyan ustunga mos qo'yiladi. Ammo bu turdagi moslikga doim ehtiyoj tug'ilmaydi. Ba'zi hollarda modeldagi ba'zi xususiyatga mos **DB** jadvalida ustun shakllantirishga zaruriyat tug'ilmasligi mumkin. Ushbu maqsadda **NotMapped** atributidan foydalaniladi:

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

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

NotMapped atributidan foydalanish uchun **System.ComponentModel.DataAnnotations.Schema** nomlar fazosini loyihada yuklash lozim.

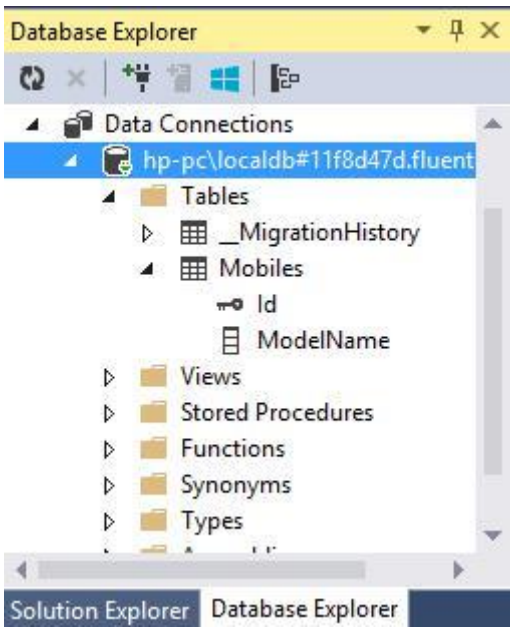
Jadval va ustunlarni moslashtirish

Entity Framework da jadval va uning ustunlari shakllantirilayotganda model va undagi xususiyat nomlaridan foydalaniladi. Ammo zaruriyat tug'ilganda **Table** va **Column** atributlaridan foydalaniladi:

```
[Table("Mobiles")]
public class Phone
{

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

Ushbu tarzda shakllantirilgan modelda **Phone** elementi **Mobiles** jadvaliga,

Name xususiyati jadvaldagi **ModelName** ustuniga mos qo'yiladi:



88-Rasm

Tashqi kalitni o'rnatish

Bir modeldagi xususiyatni boshqa model xususiyatiga ikkilamchi kalitligini o'rnatish uchun **ForeignKey** atributidan foydalaniladi:

```
public class Phone

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

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

}

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

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

Ushbu holda **Company** modeli bilan tashqi kalit vazifasini **CompId** xususiyati bajaradi.

ConcurrencyCheck

ConcurrencyCheck atributi orqali parallellashtirish muammosi hal qilinadi va jadval ustida bir nechta foydalanuvchilar amallarni parallel bajarishlari mumkin:

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

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

Ikkita foydalanuvchi bitta qiymatni o'zgartirishlari mumkin:

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

db.SaveChanges();
}
```

An'anaviy holda **Entity Framework** yangilash jarayonida jadval va model **Id** si mos bo'lsa, jadvaldagi satr yangilanadi. **ConcurrencyCheck** atributini qo'llash orqali **EF Id** va **Name** xususiyatlar qiymatlari mosligini ham tekshiradi. Agar ushbu qiymatlar o'zaro mos bo'lsa, yozuv yangilanadi. Agar qiymatlar mos bo'lmasa, **EF DbUpdateConcurrencyException** xatolikni qaytaradi.

Kompleks tiplar ustida amallar

Entity Framework da har bir model alohida jadvalga mos qo'yiladi. Ammo ba'zi hollarda muayyan klass modelning qo'shimcha ma'lumotlarini o'zida saqlashi mumkin. Masalan:

```
public class PhoneInfo

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

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

public PhoneInfo Info { get; set; }

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

PhoneInfo klassi **Phone** modeliga nisbatan bir nechta qo'shimcha ma'lumotni o'zida saqlaydi. **Phone** modeli **PhoneInfo** ob'ektini boshlang'ich qiymatlar bilan hosil qiladi. Ushbu tarzda shakllantirilgan tiplar kompleksli tiplar deb yuritiladi.

Kompleks tiplar ustida bir qator cheklagichlar mavjud:

- ularda kalitlar mavjud emas;
- ular o'zida oddiy tiplarga mansub xususiyatlarni saqlaydi;
- ushbu turdagi modellarda kolleksiyalardan foydalanib bo'lmaydi. Ya'ni **Phone** modeli **PhoneInfo** kolleksiyasini o'zida saqlay olmaydi.

Phone va **PhoneInfo** tiplar o'rtasida aloqani tashkil qilish uchun **ComplexType** atributidan foydalanish lozim. Ushbu atribut orqali turli klasslardan yagona element hosil qilinadi. Misol:

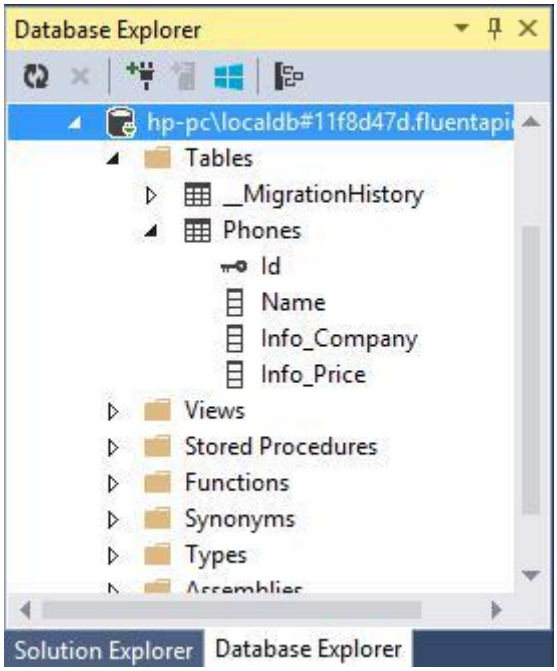
```
[ComplexType]
public class PhoneInfo
{
public string Company { get; set; }
public int Price { get; set; }
}
```

Phone klassida hech qanday o'zgarish amalga oshirilmaydi. Yuqoridagi tarzda tashkil qilingan tiplarni quyidagicha ishlatish mumkin:

```
using (FluentContext db = new FluentContext())
{
db.Phones.Add(new Phone
```

```
{
Name = "Samsung Galaxy S5",

   Info = new PhoneInfo { Company = "Samsung", Price = 17000 } });

db.Phones.Add(new Phone
{
Name = "Nokia Lumia 930",

Info = new PhoneInfo { Company = "Nokia", Price = 15000 }
});
db.SaveChanges();
foreach (Phone p in db.Phones)
Console.WriteLine("{0} - {1}", p.Name, p.Info.Price);
}
```

Ikkita klass orqali hosil qilinayotgan jadval yagona element sifatida shakllantiriladi:



89-Rasm

ComplexType metodidan **FluentAPI** da foydalanib atributlarni shakllantirish quyidagicha:

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

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

modelBuilder.ComplexType<PhoneInfo>(); base.OnModelCreating(modelBuilder);
}
}
```

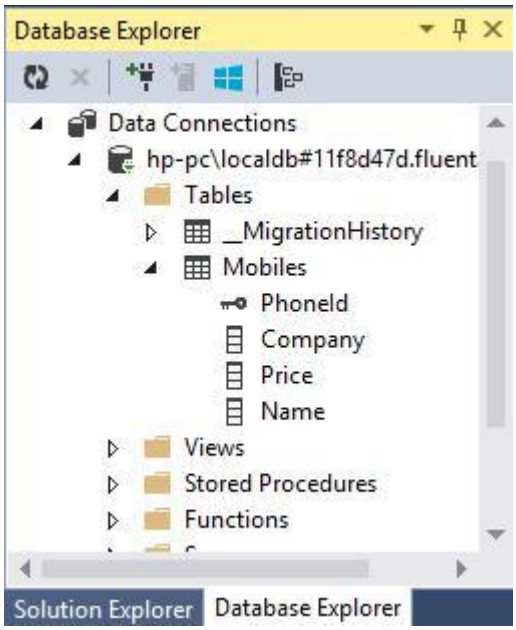
Ikkita modeldan yagona jadvalda foydalanish

Avvalgi mavzuda ikkita klass yagona element sifatida bitta jadvalda saqlangan edi. Endi boshqa modelni ko'rib chiqamiz. Ikkita model o'zaro bog'langan bo'lib, ularni yagona jadvalga saqlash uchun o'lar o'zaro birga-bir bog'lanishga ega bo'lishi lozim:

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

namespace FluentAPIApp
{
[Table("Mobiles")]
public class PhoneInfo
{
[Key, ForeignKey("Phone")]

public int PhoneId { get; set; }
public string Company { get; set; }
public int Price { get; set; }

public Phone Phone { get; set; }
}

[Table("Mobiles")]
public class Phone
{
[Key, ForeignKey("Info")]
public int PhoneId { get; set; }
public string Name { get; set; }

public PhoneInfo Info { get; set; }
```

```
}

class MobileContext : DbContext
{
public MobileContext()
:           base("DefaultConnection")
{ }

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

}
}
```

Phone va **PnoneInfo** modellari bir xil birlamchi kalitga ega bo'lib, o'lar o'z navbatida ikkilamchi kalit vazifasini bajaradi. Natijada quyidagi jadval shakllantiriladi:



90-Rasm

Yuqoridagi modellarni quyidagicha ishlatish mumkin:

```
using (MobileContext db = new MobileContext())
{
            PhoneInfo pi1 = new PhoneInfo { PhoneId = 5, Company = "Samsung",
Price = 14000 };

PhoneInfo pi2 = new PhoneInfo { PhoneId = 6, Company = "Nokia", Price

= 8000 };

Phone p1 = new Phone { PhoneId = 5, Name = "Samsung Galaxy S5", Info
= pi1 };
```

```
Phone p2 = new Phone { PhoneId = 6, Name = "Nokia Lumia 630", Info =
pi2 };

db.Infos.Add(pi1);
db.Infos.Add(pi2);
db.Phones.Add(p1);
db.Phones.Add(p2);

db.SaveChanges();

foreach (Phone p in db.Phones.Include(p => p.Info))
Console.WriteLine("{0} ({1}) - {2}", p.Name, p.Info.Company,
p.Info.Price);
}
```

Dastur ishga tushirilgach, quyidagi natija ekranga chiqariladi:
Samsung Galaxy S5 (Samsung) - 14000
Nokia Lumia 630 (Nokia) - 8000

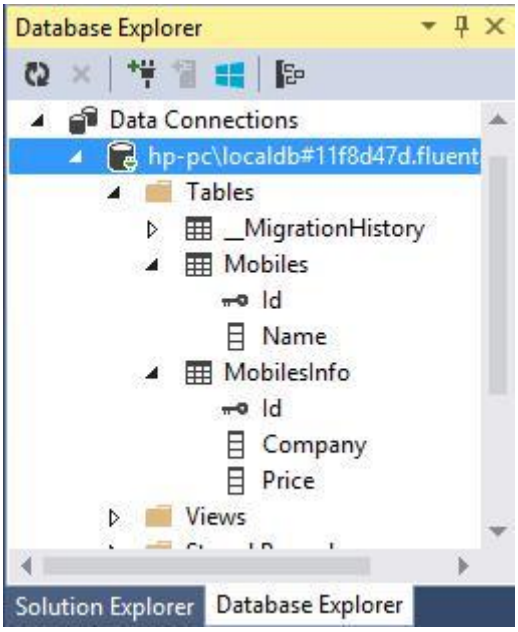
Elementlarni bir nechta jadvallarga taqsimlash

Avvalgi mavzuda ikkita klass yagona jadvalga birlashtirilgan edi. Ushbu amalga teskari bo'lgan amalni ham bajarish mumkin. Modelning turli xususiyatlarini turli jadvallarga saqlash mumkin. Modelning muhim xususiyatlarini bitta jadvalga, qo'shimcha xususiyatlarini boshqa jadvalga yozish mumkin:

```
public class Phone

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

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

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

```
modelBuilder.Entity<Phone>().Map(m =>
{
m.Properties(d => new { d.Name, d.Id });
m.ToTable("Mobiles");
}).Map(m =>
{

m.Properties(d => new { d.Company, d.Price }); m.ToTable("MobilesInfo");

}); ;
base.OnModelCreating(modelBuilder);
}
}
```

Map metodi orqali alohida tarzda shakllantirilgan xususiyatlarni jadvalga birlashtirish mumkin. **Name** xususiyatlari **Mobiles** jadvaliga, **Company** va **Price** xususiyatlari **MobilesInfo** jadvaliga saqlanadi. **Id** xususiyatini tashlab ketish mumkin. Chunki bu ma'lumot har ikkala jadvalda aloqani ta'minlashda ishlatiladi. Natijada ikkita jadval shakllantiriladi:



91-Rasm

Model ustida amal quyidagicha shakllantirilishi mumkin:

```
using (MobileContext db = new MobileContext())

{
Phone p1 = new Phone { Id = 1, Name = "Samsung Galaxy S5", Company
=   "Samsung", Price = 14000 };
```

```
            Phone p2 = new Phone { Id = 2, Name = "Nokia Lumia 630", Company =
"Nokia", Price = 8000 };

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

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

Nazorat savollari:

1. Fluent API xususiyatlarini tavsiflab bering.
2. Fluent API qanday usullardan foydalanishi mumkin?
3. Fluent API modellari orasidagi munosabatni keltiring va tavsiflang: birdan nolga, birdan birga, ko‘pdan ko‘pga, birdan ko‘pga.
4. Xorijiy kalitni sozlash tartibini aytib bering.
5. Kaskadli o'chirishni o'chirishdan maqsad nima?
6. Atributlar yordamida modellar va jadvallar o'rtasidagi xaritalashning moslashuvini keltiring va tavsiflang.

7. FRAMEWORK DA VORISLASH. TPH YONDASHUVI

Reja:

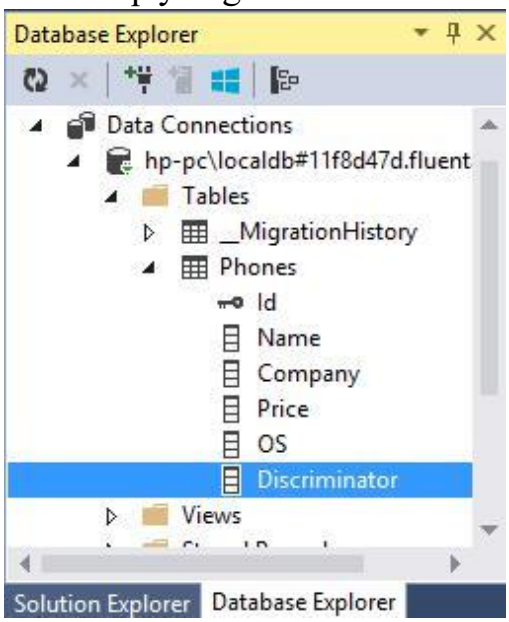
1. **TPH yondashuvi.**
2. **TPT yondashuvi.**
3. **TPC yondashuvi.**
4. **Nazorat savollari.**

TPH yondashuvi asosida klasslar ierarxiyasi uchun bitta jadval shakllantiriladi. Bosh va vorislanuvchi klass ma’lumotlari yagona jadvalda saqlanadi. Jadval ma’lumotlarini farqlash uchun jadvalda qo’shimcha ustun hosil qilinadi. Quyida telefon va smart telefonlarni ifodalovchi klasslar ierarxiyasi keltirilgan:

```
public class Phone
{

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

```
public string Name { get; set; }
public string Company { get; set; }
public int Price { get; set; }
}
public class Smartphone : Phone
{
public string OS { get; set; }
}
class MobileContext : DbContext
{
public MobileContext()

:           base("DefaultConnection")
{ }

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

Ushbu klasslar ierarxiyasida **Smartphone** klassi **Phone** klassidan vorislangan bo'lib, unga mos jadval tuzilmasi quyidagicha:

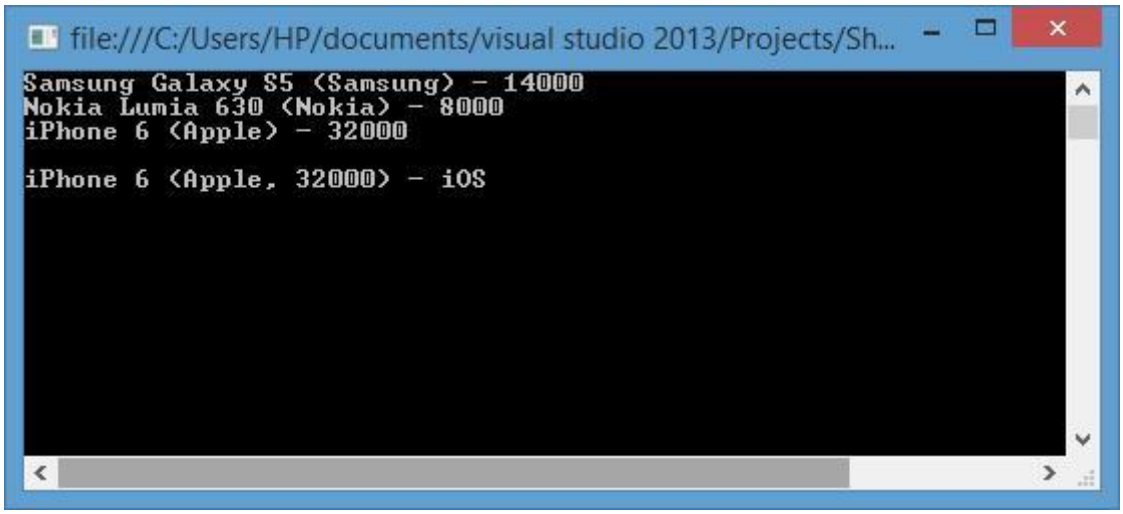


92-Rasm

Phone va **Smartphone** klasslarda barcha xususiyatlardan tashqari **Discriminator** ustuni ham mavjud bo'lib, u **nvarchar(128)** tipiga mansub qiymatni o'zida saqlaydi. Ushbu ustun jadval qiymatining **Phone** yoki **Smartphone** tipiga mansubligini ifodalaydi. Ushbu klasslardan dasturda quyidagicha foydalaniladi:

```
using (MobileContext db = new MobileContext())
{

        db.Phones.Add(new Phone { Name = "Samsung Galaxy S5", Company =
"Samsung", Price = 14000 });

        db.Phones.Add(new Phone { Name = "Nokia Lumia 630", Company =
"Nokia", Price = 8000 });

        Smartphone s1 = new Smartphone { Name = "iPhone 6", Company =
"Apple", Price = 32000, OS = "iOS" };

db.Smarts.Add(s1);
db.SaveChanges();

foreach (Phone p in db.Phones)

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

foreach (Smartphone p in db.Smarts)
Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name, p.Company, p.Price,
p.OS);
}
```

Natijada konsolga quyidagi natija hosil qilinadi:



93-Rasm

Smartphone ob'ekti o'z navbatida **Phone** ob'ekti hisoblanganligi sababli, unga ham **db.Phones** orqali murojaat qilish mumkin.

TPT yondashuvi

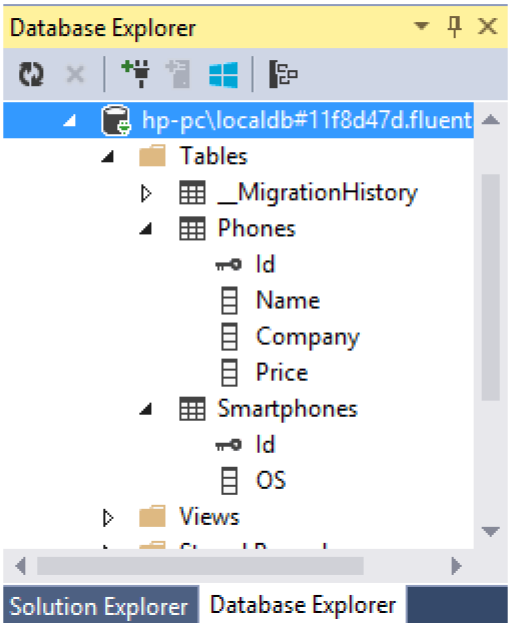
TPT yondashuvi asosida jadvalda bosh klassda aniqlangan hamda vorislangan xususiyatlar jadvalda saqlanadi. Faqat vorislangan klassga mansub qiymatlar esa alohida jadvalda saqlanadi.

Ushbu yondashuvni amalga oshirish uchun avvalgi mavzuda keltirilgan

Smartphone klassiga **Table** atributini qo'llaymiz:

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

public string Name { get; set; }
public string Company { get; set; }
public int Price { get; set; }
}
[Table("Smartphones")]

public class Smartphone : Phone
{
public string OS { get; set; }
}
class MobileContext : DbContext

{
public MobileContext()
:          base("DefaultConnection")
{ }

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

Qolgan amallar **TPH** yondashuviga o'xshash tarzda amalga oshiriladi. Ammo **TPT** yondashuviga mos ma'lumotlar bazasidagi jadvallar quyidagicha shakllantiriladi:



94-Rasm

Smartphonlarni o'zida saqlovchi jadval faqat bitta **OS** maydonidan iborat bo'lib, **Id** maydoni orqali **Phones** jadvali bilan bog'langan.

```
using (MobileContext db = new MobileContext())
{

        db.Phones.Add(new Phone { Name = "Samsung Galaxy S5", Company =
"Samsung", Price = 14000 });

        db.Phones.Add(new Phone { Name = "Nokia Lumia 630", Company =
"Nokia", Price = 8000 });

        Smartphone s1 = new Smartphone { Name = "iPhone 6", Company =
"Apple", Price = 32000, OS = "iOS" };

db.Smarts.Add(s1);
db.SaveChanges();

foreach (Phone p in db.Phones)

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

Console.WriteLine();
foreach (Smartphone p in db.Smarts)
Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name, p.Company, p.Price,
p.OS);
}
```

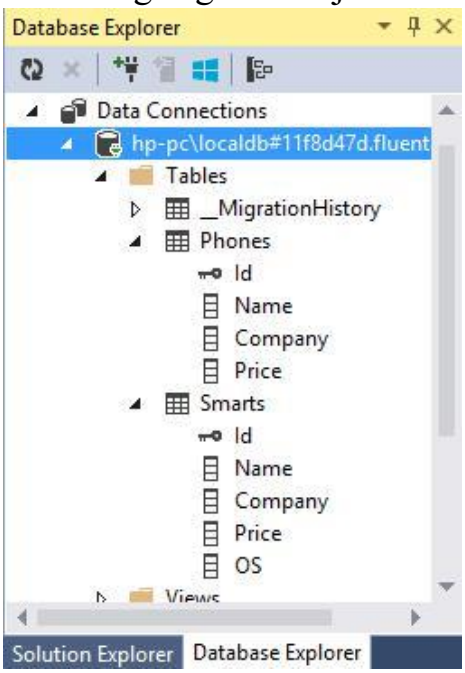
TPC yondashuvi

TPC yondashuvida har bir model uchun alohida jadvalni shakllantirish amalga oshiriladi. Har bir jadval ustunlari modelning xususiyatlari mos holda shakllantiriladi. Ushbu yondashuvni tadbiq qilish uchun model va ma'lumotlar konteksti quyidagicha hosil qilinadi:

```
public class Phone
{
[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public Guid Id { get; set; }

public string Name { get; set; }
public string Company { get; set; }
public int Price { get; set; }
}
public class Smartphone : Phone
{
public string OS { get; set; }
}
class MobileContext : DbContext
{
public MobileContext()

:           base("DefaultConnection")
{ }
public DbSet<Phone> Phones { get; set; } public DbSet<Smartphone> Smarts { get; set;
}
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
modelBuilder.Entity<Phone>()
.Map(m =>
{
m.MapInheritedProperties();
m.ToTable("Phones");
});
modelBuilder.Entity<Smartphone>().Map(m =>

{
m.MapInheritedProperties();
m.ToTable("Smarts");
});
}
}
```

Yuqoridagi kodda **Phone** klassida kalit sifatida **int** o'rniga **Guid** dan foydalanilgan. Ushbu usul kalit bilan bog'liq bo'lgan ba'zi noqulayliklarni oldini oladi. Shuningdek, model va jadvalni moslashtirishda har bir modelda **MapInheritedProperties()** metodi chaqirilgan. Ushbu metodda modellar vorislanganda ular o'rtasidagi aloqani qayta aniqlash amalga oshiriladi.

TPC yondashuvi asosida ma'lumotlar bazasi generatsiya qilinganda ma'lumotlar bazasida bir xil tuzilmaga ega ikkita jadval hosil qilinadi:



95-Rasm

Yuqorida keltirilgan modeldan quyidagicha foydalanish mumkin:

```
using (MobileContext db = new MobileContext())
{

        db.Phones.Add(new Phone { Name = "Samsung Galaxy S5", Company = 
"Samsung", Price = 14000 });

        db.Phones.Add(new Phone { Name = "Nokia Lumia 630", Company = 
"Nokia", Price = 8000 });

        Smartphone s1 = new Smartphone { Name = "iPhone 6", Company = 
"Apple", Price = 32000, OS = "iOS" };

db.Smarts.Add(s1);

db.SaveChanges();

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

```
Console.WriteLine();
foreach (Smartphone p in db.Smarts)
Console.WriteLine("{0} ({1}, {2}) - {3}", p.Name, p.Company, p.Price,
p.OS);
}
```

Smartphone ob’ekti **Phones** jadvali bilan hech qanday bog‘lanishga ega bo’lmasada, ma’lumotlarni olishda ushbu jadval ham **db.Phones** ro'yxatda bo’ladi.

Nazorat savollari:

1. TPH, TPT, TPC yondashuvlarining asosiy farqlarini keltiring.
2. TPH yondashuvi ierarxik tuzilishga qanday aloqasi bor?
3. TPT yondashuvida merosning xossalari to`liq aks ettirilgan deb ayta olamizmi?
4. Barcha meros xossalari uchun alohida jadvallar yaratish uchun qanday yondashuv qo'llaniladi?
5. Kalitni o'rnatish jarayonini tasvirlab bering.

8. ENTITY FRAMEWORK DA ASINXRONLIK. ASINXRON AMALLAR

Reja:

1. **Asinxron amallar.**
2. **Nazorat savollari.**

Entity Framework 6.0 versiyadan asinxron amallar qo’llab-quvvatlanadi. Natijalarni ma’lumotlar bazasiga asinxron tarzda saqlash uchun **SaveChangesAsync** metodidan foydalaniladi.

Ob’ektni **Id** bo’yicha asinxron tarzda olish uchun **DbSet** klassida **FindAsync** metodi aniqlangan.

Linq to Entities: da ba’zi metodlar asinxron nusxalarga ega:

- **ForEachAsync**: ma’lumotlarni asinxron tarzda olish va ular ustida muayyan amallarni bajarish;
- **AllAsync**: barcha elementlarning muayyan shartni bajarishini aniqlash;
- **AnyAsync**: tanlash operatoridagi kamida bitta element shartni qanoatlantirishini aniqlash;
- **AverageAsync**: o'rtacha qiymatni asinxron tarzda olish;

- **ContainsAsync**: muayyan elementning ro'yxatda mavjudligi aniqlanadi;
- **CountAsync**: ro'yxatdagi elementlar soni aniqlanadi;
- **FirstAsync**: ro'yxatdagi birinchi elementni olish;
- **FirstOrDefaultAsync**: ro'yxatdagi birinchi elementni yoki boshlang‘ich qiymatni olish;
- **LoadAsync**: ma’lumotlarni asinxron tarzda keshga yuklash;
- **MaxAsync**: ro'yxatdagi maksimal elementni olish;
- **MinAsync**: ro'yxatdagi minimal elementni olish;
- **SingleAsync**: ro'yxatdagi bitta elementni olish;
- **SingleOrDefaultAsync**: ro'yxatdagi bitta elementni yoki boshlang‘ich qiymatni olish;
- **SumAsync**: qiymatlar yig‘indisini asinxron tarzda olish. Ushbu metodlar **Task** yoki **Task<T>** ob’ektini qaytaradi.

Ma’lumotlar bazasidan asinxron tarzda ma’lumotlarni olamiz va saqlaymiz:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using System.Threading.Tasks;
class Program
{
static void Main(string[] args)
{

Phone p = new Phone { Name = "Nokia Lumia 930", Price = 13000 };
SaveObjectsAsync(p).Wait();

Task t = GetObjectsAsync();
t.Wait();

Console.Read();
}
public static async Task GetObjectsAsync()
{
using (MobileContext db = new MobileContext())
{
```

```
await db.Phones.ForEachAsync(p =>
{
   Console.WriteLine("{0} ({1})", p.Name, p.Price); });

}
}
private static async Task SaveObjectsAsync(Phone p)
{
using (MobileContext db = new MobileContext())
{
db.Phones.Add(p);
await db.SaveChangesAsync();

}
}
}
```

Linq to Entities asinxron amallaridan tashqari **DB**da **ExecuteSqlCommandAsync** metodi yordamida buyruqlarni asinxron tarzda amalga oshirish mumkin:

```
private static async Task DbCommandAsync(Phone p)

{
using (MobileContext db = new MobileContext())
{
        System.Data.SqlClient.SqlParameter name = new
System.Data.SqlClient.SqlParameter("name", p.Name);

        System.Data.SqlClient.SqlParameter price = new
System.Data.SqlClient.SqlParameter("price", p.Price);

        await db.Database.ExecuteSqlCommandAsync("INSERT INTO Phones
(Name, Price) VALUES (@name, @price)", name, price);

}
}

Phone p2 = new Phone { Name = "iPhone 6", Price = 33000 };
bCommandAsync(p2).Wait();
```

Nazorat savollari.

1. Ma'lumotlar bazasidan ma'lumotlarni olishda asinxron operatsiyalarni qo'llash mumkinmi?

2. ExecuteSqlCommandAsync usuli yordamida ma lumotlar bazasiga buyruqlar qo llash jarayonini tasvirlab bering.

3. Linq to Entities dasturida namuna olish va proyeksiyalash qanday amalga oshiriladi?

XULOSA

Entity Framework 6 texnologiyasini o’zlashtirish orqali talabalar relyatsion ma’lumotlar bazasi jadvallari ustida konseptual ob’ektli modellarni qurishni mustaqil tarzda amalga oshirishlari mumkin. Ma’lumotlarning konseptual ob’ektli modellari orqali shakllantirilgan ma’lumotlar ustida **LINQ** vositalari orqali ularni boshqarishni, o’zgartirish va turli so’rovlarni amalga oshirishlari mumkin.

Foydalanilgan adabiyotlar

Asosiy adabiyotlar

1. Roger Pressman, Bruce Maxim, Software Engineering: A Practitioner's Approach, John Wiley & Sons, USA 2014.
2. Ian Sommerville. Software Engineering Hardcover. Pearson 2010 USA

Qo’shimcha adabiyotlar

3. Orit Hazzan, Yael Dubinsky Agile Software Engineering Springer; 2009
4. Akao, Y., Quality Function Deployment, Productivity Press, 2004
5. Ambler, S., The Object Primer, 2d ed., Cambridge University Press, 2001
6. Larman K Primeneniye UML i shablonov proyektirovaniya, M,2002.
7. Trofimov «Rational Rose», SPb, Piter – 2002.
8. Maklarov S. V. Sozdaniye informasionnых sistem s AllFusion Modeling Suite. M.: DIALOG – MIFI, 2002. – 224 s.
9. Maklarov S. V. BPWin i ERWin. CASE – sredstva razrabotki informasionnых sistem. - M.: DIALOG – MIFI, 1992. – 256 s.
10. R. Lafore. Obyektno-oriyentirovannoye programmirovaniye v S++ [Tekst].
M.: BINOM 2004.

Elektron manbalar

1. Computerworld, PC Week, PC Magazine, PC Computing, Macworld, Inter@ctive week, Computer Shopper , PC World jurnallari.
2. Apple Computer, Dell Computer, Gateway, IBM, Hewlett-Packard, Compaq, and Sun Microsystems, Microsoft, Lotus, IBM, Oracle WWW saytlari.
3. springer.com

O‘ZBEKISTON RESPUBLIKASI
OLIY VA O‘RTA MAXSUS
TA’LIM VAZIRLIGI

GUVOHNOMA

O‘QUV ADABIYOTINING NASHR RUXSATNOMASI

O‘zbekiston Respublikasi Oliy va o‘rta maxsus ta’lim vazirligining 20 22 yil “ 9 ” sentabr dagi “ 302 ” -sonli buyrug‘iga asosan

D.T.Muxammadiyeva, L.P.Varlamova, S.A.Baxromova, B.B.Elov
(mualifning familiyasi, ismi-sharifi)

5330100-Axborot tizimlarining matematik va dasturiy ta’minoti

(ta’lim yo‘nalishi (mutaxassisligi))

ning

talabalari (o‘quvchilari) uchun tavsiya etilgan

Dasturiy injiniring Entity freamwork nomli o‘quv qo‘llanmasi
(o‘quv adabiyotining nomi va turi: darslik, o‘quv qo‘llanma)

ga

O‘zbekiston Respublikasi Vazirlar Mahkamasi tomonidan litsenziya berilgan nashriyotlarda nashr etishga ruxsat berildi.

Vazir _____ A.Toshkulov
(imzo)

Ro‘yxatga olish raqami

302-0384