ADO.NET **Entity**

Abdullah **UĞRAŞKAN**
Web Developer

apoStyLEE.com
twitter.com/**apoStyLEE**

İçindekiler

Ado.Net Entity Nedir ?

Ado.Net Entity, Microsoft' un Orm[1] aracıdır. Bu araçlar, veritabanı işlemlerini nesneler üzerinden yapmamıza olanak sağlayan yapılardır. Veritabanında bulunan her tabloya karşılık gelen bir nesne bulunmaktadır. Veritabanına ekleme, silme, listeleme vb.. işler bu nesneler kullanılarak, güvenli ve hızlı bir şekilde yapılır. Orm araçlarının diğer bir özelliğide veritabanı bağımlılığını ortadan kaldırmasıdır. Projenizi mssql veritabanı kullarak geliştirip sonrasında mysql e çevirebilirsiniz. Buna imkan veren ise; projenizde sql kodu kullanmadan her şeyi nesneler üzerinden gerçekleştirmenizdir.

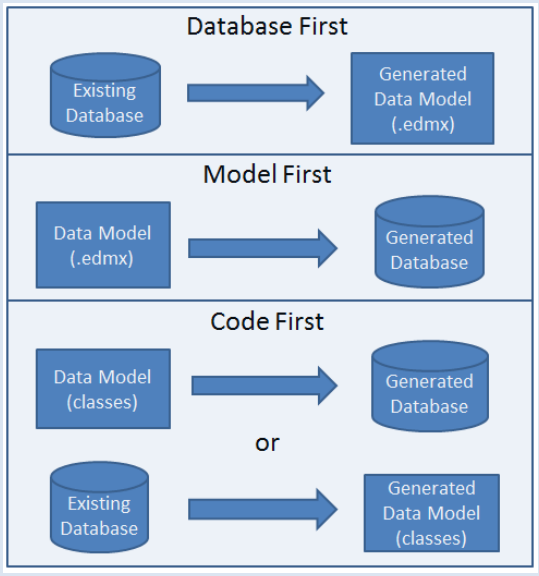
Neden Entity ?

.Net projelerimizde farklı orm araçları kullanmamızda mümkün, örneğin **nHibernate**. Bizim Entity' i seçmemizdeki en büyük etken, Microsoft' un ürünü olması ve Visual Studio ile birlikte gelen özelliklerle birlikte çok daha etkin kullanılabiliyor olması.

Model Yaklaşımları

Ado.Net Entity Framework bize modelimizi oluşturmamız için 3 seçenek (Şekil 1) sunuyor. Bunlar;

1. **Database First** Var olan bir veritabanından modelimizi oluşturduğumuz yaklaşım.
2. **Model First** Oluşturduğumuz data modelden[2], veritabanını oluşturduğumuz yaklaşım.
3. **Code First** Model sınıflarını elle yazdığımız yaklaşım. *Bizim ilgileneceğimiz yaklaşım bu olacak.*



Şekil 1 (Model Yaklaşımları)

[1] Orm; Object-relational mapping kelimelerinin kısaltılmış halidir. Detaylı bilgiye buradan ulaşabilirsiniz.
[2] Data Model; modeli bir editör tarafından tasarladığımız ve modelimize ait sınıfları, metotları ve diğer kodları otomatik oluşturan yaklaşım oluyor.

Basit bir model

Model; veritabanında ki tablomuza karşılık gelen bir sınıftır (class). Modelimiz de aynı zamanda ilişkileride (relationship) tanımlıyoruz, bunu ilerleyen sayfalarda göreceğiz.

```
public class kullanici
{
    public int id { get; set; }
    public string adi { get; set; }
    public string soyadi { get; set; }
    public string eposta { get; set; }
    public string sifre { get; set; }
    public DateTime kayitZamani { get; set; }
}
```

- **kullanici** veritabanında ki tablo adını ifade eder.
- **id, adi, soyadi..** tabloda ki alanları ifade eder.
    - **id** alanı entity framework tarafından otomatik olarak algılanıp, otomatik artan birincil anahtar (primary key) olarak belirlenir. Bu alanı **tabloAdiId** (kullaniciId) şeklinde de kullanabiliriz. Farklı bir alanı birincil anahtar olarak belirlemek istersek, alanın başına [Key] açıklamasını (Annotations) eklememiz yeterli olacaktır. *Bu olayı sağlayan Ado.Net Fluent Api adında ki yapı. İlerleyen sayfalarda detaylı bir şekilde değineceğiz.*
- **int, string, datetime..** alanların türlerini ifade eder.

DbContext

```
public class adoNetKitapContext : DbContext
{
    public DbSet<kullanici> kullanicis { get; set; }
}
```

Oluşturduğumuz modelin anlam kazanması için bir bağlam (context) oluşturmamız gerekiyor. Bu context bizim modelimizi kullanıp veritabanı işlemlerimizi yapmamıza olanak sağlıyor.

- **adoNetKitapContext** hem contextimizin hemde **web.config** içerisine yazacağımız bağlantı cümlesinin (connection string) adı oluyor.
- **DbSet<kullanici>** kullanacağımız model sınıfımızı belirtiyoruz.
- **kullanicis** buda veritabanı işlemleri için kullanacağımız methodları içinde barındıran sınıfımızın adını ifade ediyor. Aynı zamanda veritabanı otomatik[3] oluşturulduğunda tablo adınıda ifade ediyor.

Veri katmanımızı basitçe bu şekilde oluşturuyoruz. Şimdi basit bir örnekle veritabanına ekleme, listeleme, güncelleme ve silme (CRUD) işlemlerine bir göz atalım.

[3] Orm araçları veritabanını modelinizdeki tanımlara göre otomatik olarak oluştururlar. Ado.Net Entity projenizi çalıştırdığınızda context içindeki tanımlara göre modelinizi oluşturacaktır.

Ekle, Listele, Düzenle ve Sil (CRUD)

Yukarıda ki modelimizi kullanarak aşağıdaki gibi veritabanı işlemlerini yapabiliriz. Öncelikle **web.config** içerisine veritabanı bağlantısı için gerekli olan bağlantı cümlemizi yazıyoruz. Ben **Sql Compact 4**[4] veritabanını kullanıyorum.

```
<add name="adoNetKitapContext" connectionString="Data Source=|DataDirectory|\db.sdf"
providerName="System.Data.SqlServerCe.4.0" />
```

```
        public void Crud()
        {
            // context imizi tanımlıyoruz.
            adoNetKitapContext context = new adoNetKitapContext();

            // Kullanıcı modelimiz
            kullanici k = new kullanici();
            k.adi = "Ahmet";
            k.eposta = "ahmet@ahmet.com";
            k.kayitZamani = DateTime.Now;
            k.sifre = "123";

            // Veritabanına kayıt ekleme
            context.kullanicis.Add(k);
            context.SaveChanges();

            // Kayıtları listeleme
            var kayitListesi = context.kullanicis.ToList();

            // Kayıt güncelle
            int gKayitId = 1;
            var guncellenecekKayit = context.kullanicis.Find(gKayitId);
            guncellenecekKayit.adi = "yeni değer";
            guncellenecekKayit.soyadi = "yeni değer";
            context.SaveChanges();

            // Kayıt silme
            int sKayitId = 1;
            var silinecekKayit = context.kullanicis.Find(sKayitId);
            context.kullanicis.Remove(silinecekKayit);
            context.SaveChanges();
        }
```

Görüldüğü üzere hiçbir sql kodu yazmadan veritabanı işlemlerini gerçekleştirdik. Şimdi arka planda neler olduğuna bir göz atalım. Gözlemleme işini **Sql Server Profiler** ile yapacağım için öncelikle web.config de bulunan bağlantı cümlemi Mssql veritabanı kullanmak için değiştiriyorum. ** Orm araçlarının bir avantajıda veritabanı bağımsızlığıydı.*

```
<add name=" adoNetKitapContext"
connectionString="Server=localhost;Database=dbName;User ID=sa;Password=pass;"
providerName="System.Data.SqlClient" />
```

[4] Sql Compact 4; ek yükleme yapmadan direk kullanabileceğiniz bir veritabanıdır. 4Gb' a kadar verileri depolayabilir ve ücretsiz olduğundan dağıtımda düşük maliyet sağlar. Transact-SQL kullanımını destekler. Takım günlüğüne http://blogs.msdn.com/b/sqlservercompact/ adresinden ulaşabilirsiniz.

Crud işlemlerini çalıştırdığımızda **Sql Server Profiler** da gözlenen sql sorguları aşağıdaki gibidir. Ado.Net Entity nesnelerle gerçekleştirdiğimiz işlemleri bu şekle çevirip sql servera işletiyor. İşlemler parametre kullanılarak gerçekleştirildiğinden, **sql enjeksiyon**[5] saldırılarına karışıda güvenli bir hale geliyor.

```
// Ekleme işlemi
exec sp_executesql N'insert [dbo].[kullanicis]([adi], [soyadi], [eposta], [sifre], [kayitZamani])
values (@0, null, @1, @2, @3)
select [id]
from [dbo].[kullanicis]
where @@ROWCOUNT > 0 and [id] = scope_identity()',N'@0 nvarchar(max) ,@1 nvarchar(max)
,@2 nvarchar(max) ,@3
datetime2(7)',@0=N'Ahmet',@1=N'ahmet@ahmet.com',@2=N'123',@3='2011-07-12
17:16:30.3015048'
go


// Listeleme işlemi
SELECT
[Extent1].[id] AS [id],
[Extent1].[adi] AS [adi],
[Extent1].[soyadi] AS [soyadi],
[Extent1].[eposta] AS [eposta],
[Extent1].[sifre] AS [sifre],
[Extent1].[kayitZamani] AS [kayitZamani]
FROM [dbo].[kullanicis] AS [Extent1]
go


// Güncelleme işlemi
exec sp_executesql N'update [dbo].[kullanicis]
set [adi] = @0, [soyadi] = @1
where ([id] = @2)
',N'@0 nvarchar(max) ,@1 nvarchar(max) ,@2 int',@0=N'yeni değer',@1=N'yeni değer',@2=4
go


// Silme İşlemi
exec sp_executesql N'delete [dbo].[kullanicis]
where ([id] = @0)',N'@0 int',@0=4
go
```

[5] Sql enjeksiyon bir saldırı şeklidir. Buradan http://ferruh.mavituna.com/sql-injection-a-giris-ve-sql-injection-nedir-oku/ konuyla ilgili detaylı bilgiye ulaşabilirsiniz.

Doğrulama (Validation)

Veriler eklenirken yada düzenlenirken bazı kurallar çerçevisinde bu işlemler gerçekleştirilir. Bazı alanlar boş olamaz, sadece sayı olabilir yada girilen değer 5 haneli olabilir gibi.. işlemlere doğrulama (validation) diyoruz. Doğrulama işlemlerini modelimizin içindeki ilgili alanlara açıklamalar (Annotations) ekleyerek gerçekleştirebiliyoruz. Temel annotations lar **System.ComponentModel.DataAnnotations**[6] sınıfı içerisinde bulunuyor.

```
    public class kullanici
    {
        public int id { get; set; }

        [Required]
        public string adi { get; set; }

        [Required(ErrorMessage="Bu alan boş olamaz"]
        public string soyadi { get; set; }

        public string eposta { get; set; }

        [MinLength(6,ErrorMessage="En az 6 karakterden oluşmalıdır.")]
        public string sifre { get; set; }

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

Eposta adresi doğrulama gibi özel bir durum[7] için ise kendi kişisel doğrulamamızı yazıp kullanabiliyoruz.

```
public class EmailAttribute : RegularExpressionAttribute
{
       public EmailAttribute()
                : base(@"^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-
zA-Z]\.)+[a-zA-Z]{2,9})$") { }
}
```

```
public class kullanici
{
    [Email(ErrorMessage = "EPosta adresini lütfen doğru giriniz.")]
    public string eposta { get; set; }
}
```

Eğer buraya kadar okuduysanız, öğrendiklerinizle ufak bir proje yapıp bilginizi pekiştirebilirsiniz. İlerleyen kısımlarda konunun detayına doğru iniyor olacağız.

[6] http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx

[7] Özel durumlara; dosya uzantısı, şifre doğrulama, adres formatı gibi şeyleri örnek gösterebiliriz, eğer her biri için ayrı ayrı kontrol yazmak istemiyorsanız ayrı bir kütüphane kullanabilirsiniz. Örnek olarak http://www.apostylee.com/dataannotationsextensions/ adresine bakabilirsiniz.

İlişkiler (Relationship)

Veritabanı tasarımındaki en önemli noktayı hiç şüphesiz tablolar arasında ki ilişkiler oluşturuyor. Veri bütünlüğü için hayati önem taşıyan bu noktada modellerimiz içine basit bir şekilde ilişkilerimizi tanımlayabiliyoruz.

1 den Çoğa (1 to n) İlişki

Senaryomuz şu şekilde; bir kullanıcının birden fazla yazısı olabilir. Bu ilişkiyi gerçekleştirmek için **yazi** adında yeni bir model oluşturup **kullanici** modeliyle ilişkilendiriyoruz.

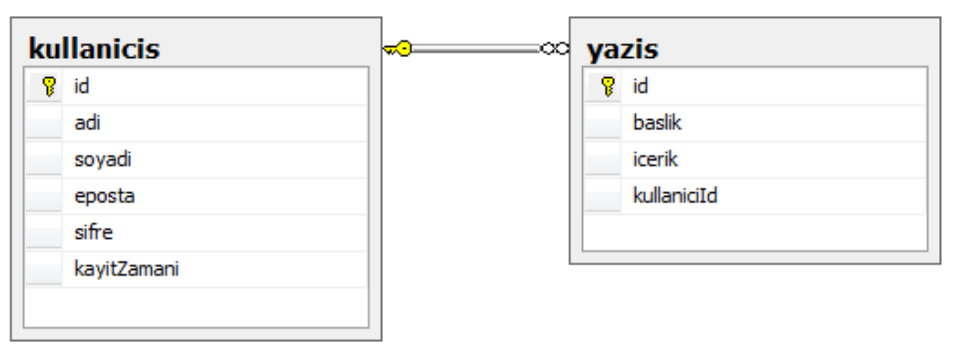
```
public class yazi
{
    public int id { get; set; }
    public string baslik { get; set; }
    public string icerik { get; set; }

    public virtual kullanici kullanici { get; set; }
    public int kullaniciId { get; set; }
}

public class kullanici
{
    public int id { get; set; }
    public string adi { get; set; }
    public string soyadi { get; set; }
    public string eposta { get; set; }
    public string sifre { get; set; }
    public DateTime kayitZamani { get; set; }

    public virtual ICollection<yazi> yazilar { get; set; }
}
```

Yukarıda ki modellerimize baktığımızda **yazi** içerisinde **kullanici** ve **kullaniciId** yi belirtip, bir yazının bir kullanıcıya ait olduğunu belirtiyoruz. **Kullanici** tarafında ise **ICollection<yazi> yazilar** tanımını yapıp, bir kullanıcının birden fazla yazısı olacağını ifade ediyoruz. Oluşan ilişki aşağıda ki (Şekil 2) gibi olacaktır.



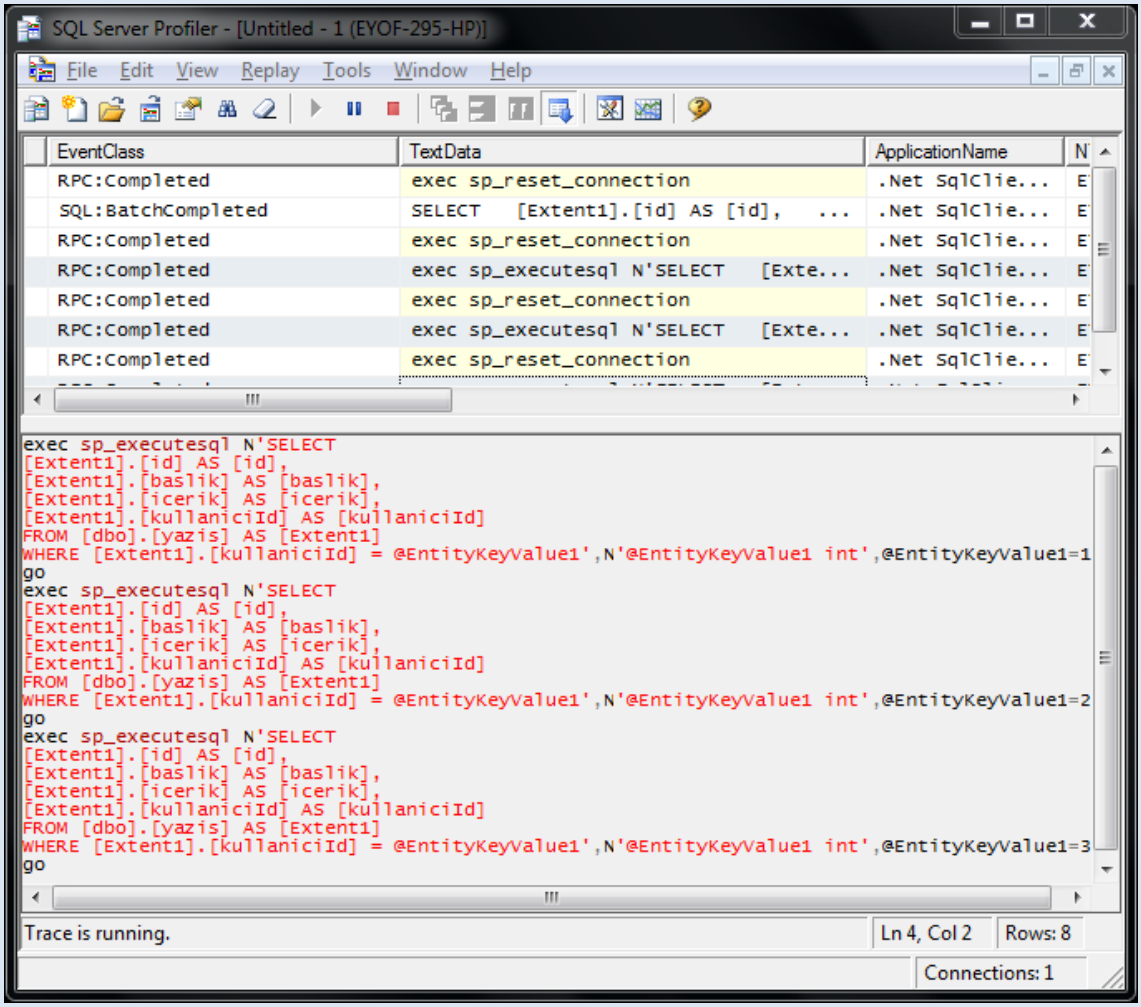
Şekil 2 (Oluşan tabloların ilişkisi)

Şimdi orm nin en zevkli kısmına sıra geldi! Uzun uzun sql cümleleri yazmadan ilişkili veriyi nasıl listeliyoruz bir bakalım.

```
var result = db.kullanicis.ToList();
var result2 = db.kullanicis.Include(x => x.yazilar);
```

Kullanıcılara ait yazıları yukarıdaki gibi iki şekilde çekebiliyoruz. Uygulamada birbirinden farksız gibi görünsede bu iki yöntem arasında ciddi bir fark bulunuyor. Bu farkı Sql Server Profiler ile inceleğimizde görüyoruz.

result işleminde **Lazy Loading**[8] adı verilen bir durum oluşmaktadır. Bu durumda döngü sayısı kadar sql işlemi gerçekleşmektedir. Yani kullanıcılara ait yazıların listelenmesi için, kullanıcı sayısı kadar sql işlemi gerçekleştirilip ait olan yazılar gelmektedir. Sql server profillerde durumu izlediğimizde;

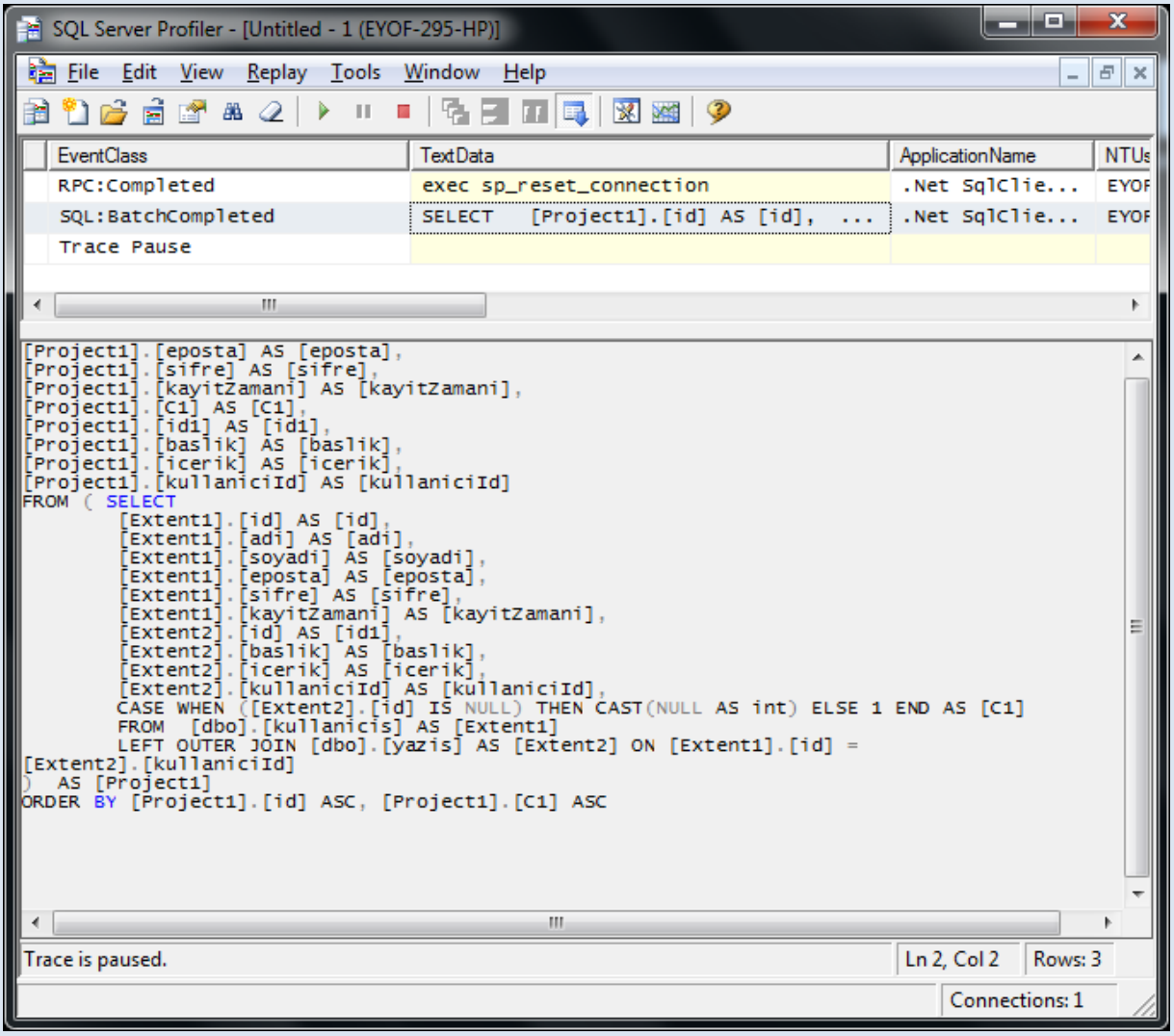


Şekil 3 (Lazy Loading)

Veritabanında ki 3 kullanıcı olduğundan her kullanıcı için yazılar isteniyor. Bu, çok yoğun olmayan durumlarda göz ardı edilebilir olsada aksi durumda ciddi bir performans kaybına yol açacaktır.

[8] http://en.wikipedia.org/wiki/Lazy_loading

result2 de bu durumu engellemek için yazi modelini kullanici modeline ekliyoruz (include). Bu sayede Ado.Net Entity Lazy Loading i Eager Loading (Şekil 4) e çeviriyor ve daha farklı bir durum oluşuyor.



Şekil 4 (Eager Loading)

Eager Loading durumunda tablolar bağlanarak tüm alanları çekilir. Yani tek bir sorguda istenilen tüm kayıtlar getirilir. Diğer yönteme göre daha performanslı olduğunu söyleyebiliriz.

Çoktan Çoğa (n to n) İlişki

Bu seferki senaryomuz, birden çok yazının birden çok kategoriye ait olması. Bu durumu gerçekleştirmek için yukarıdaki modellerimizin yanına birde **kategori** adında bir model ekliyor ve gerekli değişiklikleri yapıyoruz.

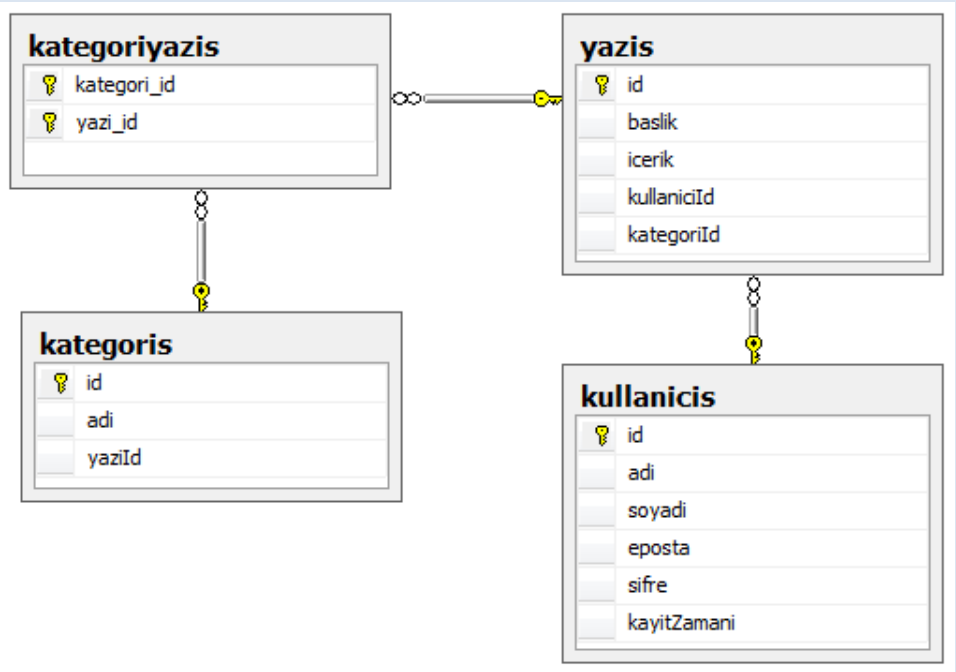
```
public class kategori
{
    public int id { get; set; }
    public string adi { get; set; }

    public virtual ICollection<yazi> yazilar { get; set; }
    public int yaziId { get; set; }
}
public class yazi
{
    public int id { get; set; }
    public string baslik { get; set; }
    public string icerik { get; set; }

    public virtual kullanici kullanici { get; set; }
    public int kullaniciId { get; set; }

    public ICollection<kategori> kategoriler { get; set; }
    public int kategoriId { get; set; }
}
```

Kategori modelinin içine, birden fazla yazı olabileceği için bir tanımlama yapıyoruz, aynı tanımın tersini yazı modeli içinde yapıyoruz. Bu durum sonucunda oluşan tablo yapısı aşağıdaki (Şekil 5) gibi oluyor.



Şekil 5 (Oluşan tabloların ilişkisi)

Yukarıda ki diyagramda görüldüğü üzere bizim modelimizde belirtmediğimiz **kategoriyazis** adında bir tablo oluşmuş. Bu tablo **n to n** ilişkinin varlığı algılanıp otomatik olarak ado.net tarafından oluşturuluyor. İlişkimizin kilidi de bu tablo.

ComplexType

Şöyle bir senaryo düşünelim, kullanıcımız var ve bu kullanıcının adres bilgileri, profil bilgileri gibi belli başlıklar altında toplanabilen alanları bulunuyor. Bu alanları kümeleyip, daha derli toplu hale getirdiğimiz yaklaşıma complextype diyoruz. Örneğin kod yazarken, kullanıcının adres ile ilgili bir bilgisine ihtiyaç duyduğumuzda adres kümesi içerisinde bu bilgiyi arıyoruz. Diğer türlü olsaydı örneğin 20 alan olsa 20 alan içerisinde aradığımı bulmaya çalışacak zaman kaybedecektik. Bir model üzerinde incelersek olayı daha iyi anlayabiliriz.

```
public class kullanici {

    public kullanici() {
        iletisim = new iletisim();
        profil = new profil();
    }

    public int id { get; set; }
    public int adi { get; set; }
    public int eposta { get; set; }

    public iletisim iletisim { get; set; }
    public profil profil { get; set; }

}

public class iletisim {
    public string adres { get; set; }
    public string sehir { get; set; }
    public string sabitTelefon { get; set; }
    public string cepTelefon { get; set; }
    public string faks { get; set; }
}

public class profil {

    public string arkaPlanRengi { get; set; }
    public string arkaPlanResmi { get; set; }
    public string fontRengi { get; set; }
    public string avatar { get; set; }
}
```

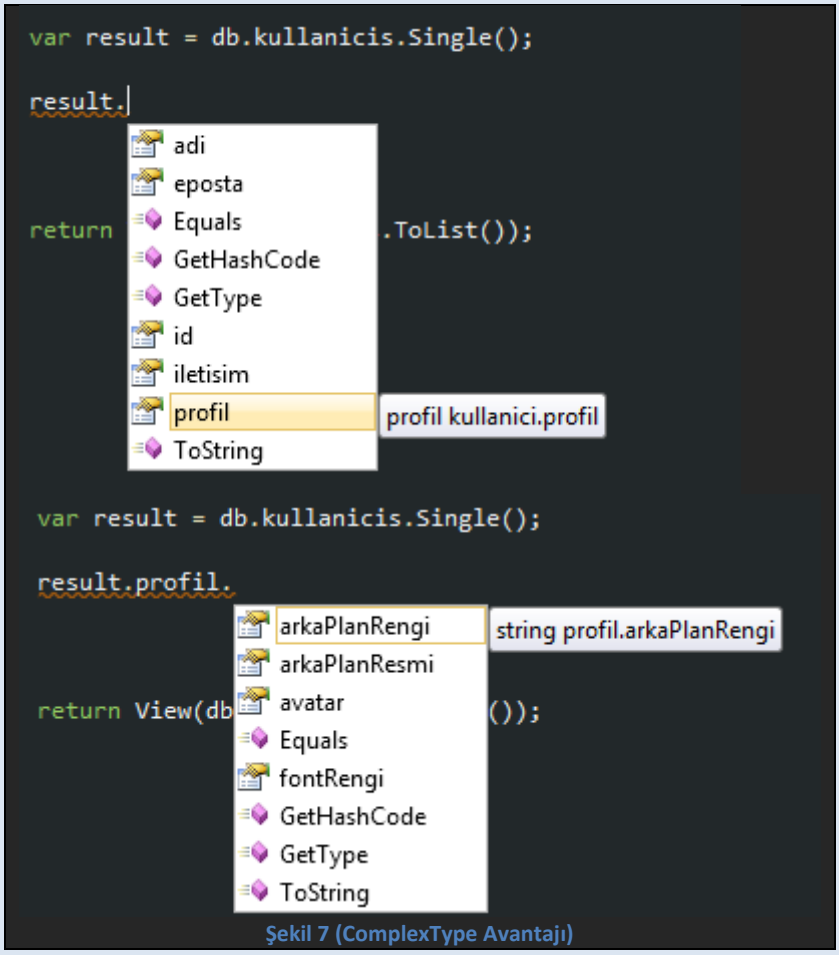
Kullanıcı modeline baktığımızda 3 modelmiş gibi görünüyor fakat **iletişim** ve **profil** sınıfları kümelediğimiz bilgileri tutmamıza olanak sağlıyor. Yani bizim complextype larımız bunlar ;) Bu modelden oluşan tablo aşağıdaki (Şekil 6) gibi oluyor. Görüldüğü üzere alanlar complextype adını ön ek olarak almışlar (**iletişim_adres**) bunun sebebi farklı kümelerde aynı alan adlarının oluşturacağı karışıklığı ve hataları önlemek.

EYOF-295-HP.testDb - dbo.kullanicis

Column Name	Data Type	Allow Nulls
id	int	☐
adi	int	☐
eposta	int	☐
iletisim_adres	nvarchar(MAX)	☑
iletisim_sehir	nvarchar(MAX)	☑
iletisim_sabitTelefon	nvarchar(MAX)	☑
iletisim_cepTelefon	nvarchar(MAX)	☑
iletisim_faks	nvarchar(MAX)	☑
profil_arkaPlanRengi	nvarchar(MAX)	☑
profil_arkaPlanResmi	nvarchar(MAX)	☑
profil_fontRengi	nvarchar(MAX)	☑
profil_avatar	nvarchar(MAX)	☑

Şekil 6 (ComplexType)

Kod yazarken bize sağladığı faydayı aşağıdaki resimde (Şekil 7) görebiliriz.



Şekil 7 (ComplexType Avantajı)

Eğer ComplexType kullanmasaydık, hangi bilginin nerede olduğunu arayacaktık fakat bu şekilde hangi bilgi nerde olabiliri rahatlıkla anlayıp, bilgiye hemen ulaşabiliyoruz. Onlarca alana sahip bir modelde bu yöntemin işimizi bir hayli kolaylaştırdığını rahatlıkla söyleyebiliriz.

Ado.Net Fluent Api

Fluent Api[9] en kaba tabirle kod tarafıyla sql tarafında ki mapping işlemlerini düzenlememize olanak sağyalan bir api. Örneğin model sınıfımızdaki bir alanın adı **adi** iken veritabanında bunun **name** olması gerekebilir yada kurduğumuz bir ilişkinin adının değişmesi gerekebilir. Bu ve benzeri düzenlemeleri fluent api sayesinde gerçekleştiriyoruz. Modelimiz üzerinde fluent api kullanımına bakacak olursak;

```
    [Table("user")]
    public class kullanici
    {
        [Key]
        public int id { get; set; }

        [Column("name")]
        public string adi { get; set; }
        public string soyadi { get; set; }
        public string eposta { get; set; }
        public string sifre { get; set; }
        public DateTime kayitZamani { get; set; }
    }
```

Bu durumda veritabanında oluşan tablonun adı **user**, modelde **adi** olan alanın **name** olarak oluştuğunu görebiliriz. Fluent Api esas olarak **context** içine yazacağımız **OnModelCreating** methodunun içinde kullanılıyor, burada ki amaç model oluşturulurken yapılacak işlemlerin tek bir yerde toplanması. Yukarıda ki örneği şimdide context içinde gerçekleştirelim.

```
public class adoNetKitapContext : DbContext
{
  public DbSet<kullanici> kullanicis { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
     modelBuilder.Entity<kullanici>().ToTable("user");
     modelBuilder.Entity<kullanici>().Property(x => x.adi).HasColumnName("name");
  }
}
```

Fluent apiyi elimizden geldiğince yaymadan context içinde kullanmamızda fayda var. Sebebi ise bu işlemlerin tek bir çatı altında toplanması ve bir problem yada güncelleme olacağı zaman değişikliklerin buradan yapılacak olması.

[9] Fluent Api örneklerine http://blogs.msdn.com/b/adonet/archive/2010/12/14/ef-feature-ctp5-fluent-api-samples.aspx adresinden bakabilirsiniz.

İpuçları

Where

```
var result1 = context.kullanicis.Where(x=>x.id > 5);
var result2 = context.kullanicis.Where(x => x.id > 10 && x.adi == "ahmet");
var result3 = context.kullanicis.Where(x => x.adi.Contains("ahmet"));
```

result1 bildiğimiz where işlemi, **id** si 5 den büyük kayıtları getiriyor.

result2 yine bildiğimiz where işlemi, **id** si 10 dan büyük adi ahmet olan kayıtları getiriyor.

result3 İçinde ahmet geçen kayıtları listeliyoruz. **Contains** ile veritabanında arama işlemini yapıyoruz, yani sql deki **like** ın işini yapıyor.

Select

```
var result6 = context.kullanicis.Select(x => new { ID = x.id, ADI = x.adi });
```

Bazı durumlarda modelimizdeki tüm alanlara ihtiyaç duymayız, select ile istediğimiz alanları belirtebiliyoruz. Yukarıda ki işlemin sql de ki karşılığı; *select id as ID, adi as ADI from kullanicis*

OrderBy

```
var result4 = context.kullanicis.OrderByDescending(x => x.id);
var result5 = context.kullanicis
                    .OrderBy(x => x.id).ThenByDescending(x=>x.kayitZamani);
```

result4 de **id** si büyükten küçüğe doğru listeliyoruz. Eğer direk *OrderBy (x => x.id);* deseydik **id** si küçükten büyüğe doğru sıralanacaktı.

result5 burada ise **id** si küçükten büyüğe ve **kayitzamani** büyükten küçüğe doğru sıralama işlemini yapıyoruz.

Top, Max, Min

```
var result7 = context.kullanicis.Take(10);
var result9 = context.kullanicis.Max(x => x.id);
var result10 = context.kullanicis.Min(x => x.id);
```

result7 10 kaydı listeler. *context.kullanicis.OrderByDescending(x => x.id).Take(10);* şeklinde ise en son girilen 10 kaydı listeleyebiliriz.

result9 en büyük id değerini döndürür.

result10 en küçük id değerini döndürür.

SqlQuery, ExecuteSqlCommand

```
var result8 = context.Database.SqlQuery<kullanici>("Select * from kullanici");
context.Database.ExecuteSqlCommand("update from kullanici set adi = 'test' ");
```

result8 de kendi sql sorgumuzu direk yazıp kullanabiliyoruz, burada dikkat edilecek husus veritabanından dönen alanların türleri ve isimleri yüklenecek sınıftakilerle birebir aynı olmalıdır. Bu yöntemi veritabanında ki bir **view** dan kayıtları çekmek için kullanabiliriz.

Direk sql komutu çalıştırmak için **ExeCuteSqlCommand** özelliğini kullanabiliyoruz. Bu özellik bazı durumlarda hayati önem taşıyabiliyor.

Son Söz

Ado.Net Entity her yeni versiyonda biraz daha iyileştirilen bir orm aracı. Özellikle web uygulaması geliştirirken MVC3 ile birlikte bize sunduğu kolaylıklar göz ardı edilecek gibi değil, günlüğümde Mvc3 ile birlikte nasıl kullanıldığına dair bilgileri bulabilirsiniz.

Umarım yararlı bir doküman olmuştur. Gelen istekler doğrultusunda bu dokümanı güncellemeye çalışacağım. *Tüm güzellikler sizinle olsun.*

Abdullah **UĞRAŞKAN**
Web Developer

apoStyLEE.com
twitter.com/**apoStyLEE**