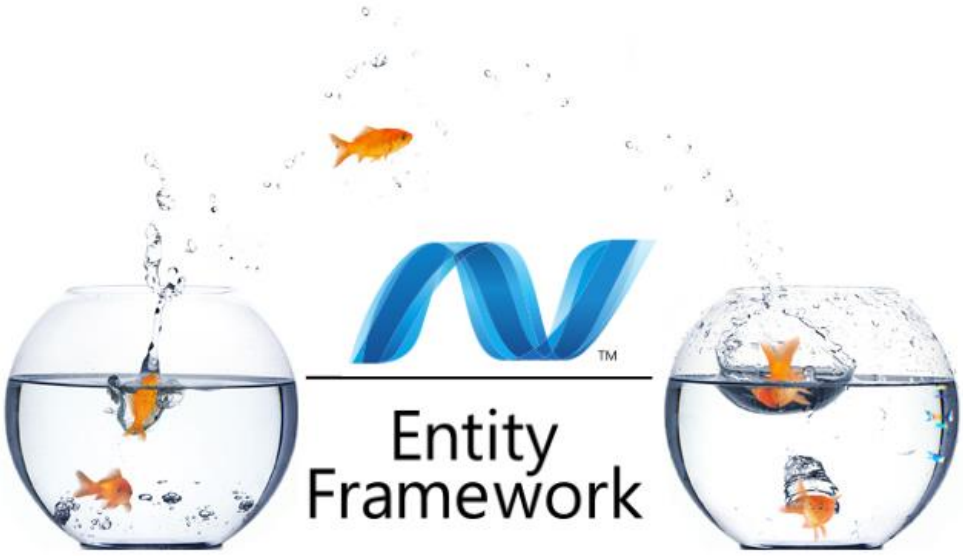
Entity Freamwork & ADO.Net

Yapılan Çalışma

Entity framework 6.0 da TransactionScope vs Ado.Net Transaction



Microsoft C#.net

Melih Hilmi ULUDAĞ

İçindekiler Tablosu

1.Giriş

Transaction kelime olarak 'işlem' anlamına gelmektedir. Programlama da ise daha küçük parçalara ayrılmayan en küçük işlem yığınına denir. Programlama da bir dize işleminin tamamının yolunda gitmesine bağlı durumlarda transaction kullanılır. Kısaca transaction bloğu ya hep ya hiç mantığı ile çalışır. (Tüm işlemler düzgün olarak yolunda gitse, sadece bir kısmı geçersiz olsa tüm blok geçersiz sayılır)

Sözel bir örnek: Bir kullanıcı olarak ATM'ye gittiniz diyelim, yapmak istediğiniz işlem para çekmek olsun. Kartınızı taktınız, şifrenizi girdiniz, para çekmeyi seçtiniz, miktarı belirlediniz ve çekmek istiyorum dediniz. ATM arka planda tüm bu işlemleri yaptı, çekmek istediğiniz para miktarını kontrol etti, paranın var olduğunu gördü, paranızı size vermek üzere para çekilecek Alana yansıttı. Kartınızı da daha önceden verdi (artık genelde önce kart, sonra para veriliyor çünkü, parayı unutma ihtimali daha az mantığı ile bakılıyor olaya) Bu esnada arka planda hesabınızdan bu miktar düştü. Para şimdi parayı alacağınız alanda ama olduki dalgınlık oldu, hemen alamadınız bir kaç saniye geçti ve bankamatikte güvenlik icabı bu parayı geri aldı. Yani parayı hesaptan çektiniz ama elinize alamadınız para geri gitti.Eeee ne olacak şimdi, benim hesabımdaki para azalacak mı? Ben parayı almadım ki daha …İşte burada Transaction işleminin önemi devreye giriyor. Kullanıcı parasını alamadığı için Transaction sonlanmamış olur ve her şeyi eski haline geri çevirir. Yani çekilen miktar tekrar kullanıcının hesabına geri döner. Transaction işleminin önemi budur. Aynı durumu bankaya para yatıran bir şahıs içinde düşenebilirsiniz. Banka önce yatırılacak miktarı sorar, sonra kullanıcıdan parayı ister, tüm işlemler sonlandığında yani en son onay alındığında işlem aslında tamamlanmış olur.

1.1 Transaction' un kullanımı (MS Sql Server' de)

Begin Tran (veya Transaction)

……...

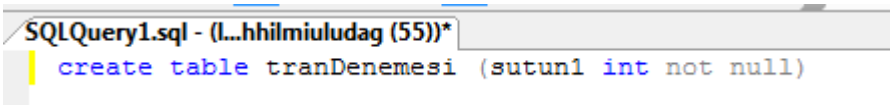
……...

Commit Transaction

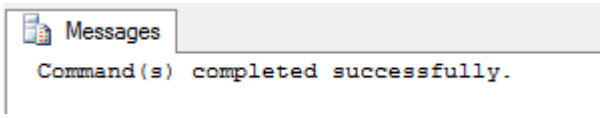
Şeklinde bir kullanıma sahiptir. (……) olan kısımlara **birbirleriyle aynı anda gerçekleşme şartı olan kodlar yazılır.**

Örnek: Yukarıda verilen örneği açıklamaya yönelik MS SQL Server' de bir uygulama yapalım.

Adım1:

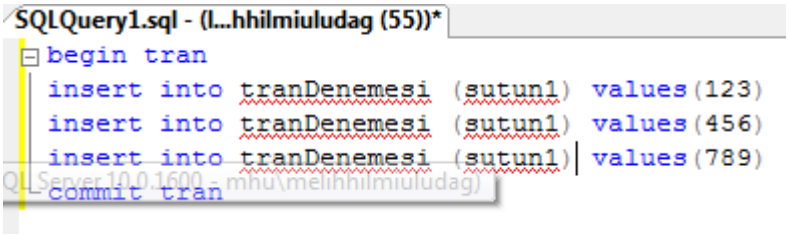


tranDenemesi adlı bir tablo oluşturdum ve sutun1 adlı bir kolona sahip olsun dedim.

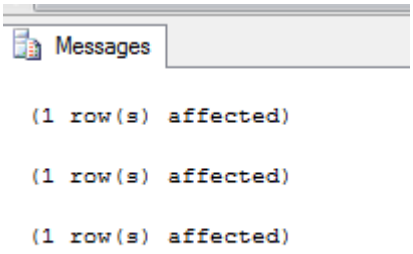


cevabı ile işlemin başarılı olduğunu görüyorum.

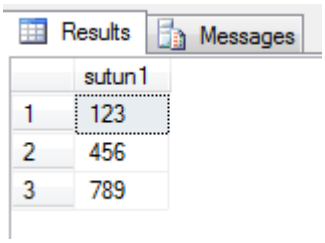
Adım2:



Begin tran/Commit tran blokları arasına yazdığım 3 satır kod ile tranDenemesi adlı tablonun sutun1 kolonuna 3 adet sayısal (int tipinde) değer eklemesi yaptım.

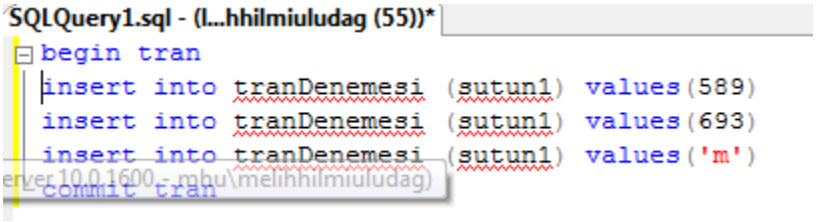


Ve Sql Server, olumlu mesaj verdi.

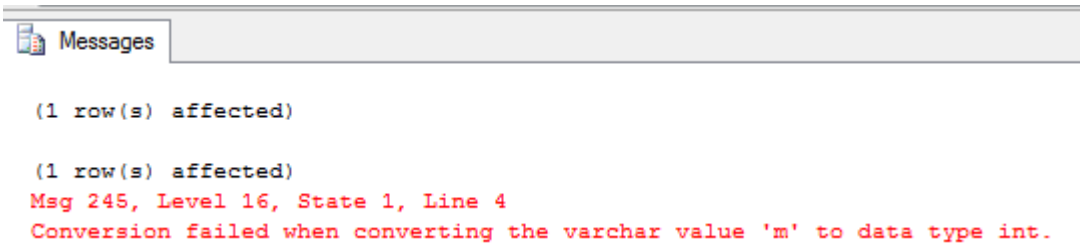


Kayıtlara göz attığımda 3 adet değerinde sorunsuz şekilde eklendiğini görüyoruz.

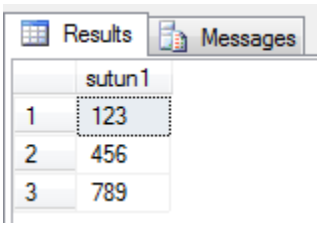
Adım3:



Burada ise 2 tane sayısal 1 tane de sözel bir veri ekleme isteğinde bulundum.



Mesajda ilk 2' sinin olumlu sonuçlandığını diğerinin de int veri türüne dönüştürülmediğinden **eklenemediği mesajını gördük.**



Tekrar tabloyu listelediğimde ilk ikisinin olumlu olmasına ragmen eklenmediğini görüyoruz. **Bunun sebebi transaction içerisinde son satırın eklenememesiydi**. Transaction kavramında **"ya hep yaa hiç"** mantığı olduğundan son satır eklenemediği için ilk iki satırı da işleme almıyor.

Transaction, bu mantıkta geliştirilmiş bir yapıdır.

2. Transaction ve TransactionScope Yapısı

Transaction yapısı sadece Ado.Net üzerinde değil, tüm data accessing teknolojileri üzerinde bulunmaktadır. Ve bulunmak zorundadır. (Transaction Nhibernate üzerinde de vardır, Entity Freamwork üzerinde de.)

ADO.NET 2.0 ile birlikte programcının yerel ve dağıtık transaction işlemlerini daha kolayca yönetebilmesi için System.Transactions.dll kütüphanesi sunulmuştur. ADO.NET 2.0'da **LightWeight Transaction(LT)** ve **OLE Transaction(OleTx)** olmak üzere iki transaction modeli geliştirildi.

2.1.LightWeight Transaction Modeli

Tek veri kaynağı üzerinde sorgulama yapılırken kullanılan transaction modeli,

- var **sqlConnectionNesnesi**=new SqlConnection("*bağlantı cümleciği yazılacak*");

Connection nesnesi oluşturulur.

- **sqlConnectionNesnesi**.Open();

Bu nesneyi kullanabilmek için etkinleştirmek/açmak gerekiyor.

- SqlTransaction **tranNesnesi**=**sqlConnection**.BeginTransaction();

Transaction nesnesi oluşturulur ve connection nesnesi üzerinde bu nesneye atama işlemi yapılmış olunur.

- var **cmd**=new SqlCommand("*sql komutu yazılır*", **sqlConnectionNesnesi**);

cmd nesnesine sql komutu atanır.

- **cmd**.Transaction=**tranNesnesi**;

cmd nesnesinin transaction u yukarıda tanımlanan tranNesnesi olsun diyerek cmd nesnesinikişiselleştirmiş oluyoruz.

try{

- **cmd**.ExecuteNonQuery();

cmd nesnesi içerisindeki sql komutunu aktifleştiriyoruz.

- **tranNesnesi**.Commit();

Eğer try bloğu içerisinde hata meydana gelmezse commit ediyoruz yani veritabanına değişiklikler yansısın diyoruz. }

catch(Exception hata){

- **tranNesnesi**.Rollback();

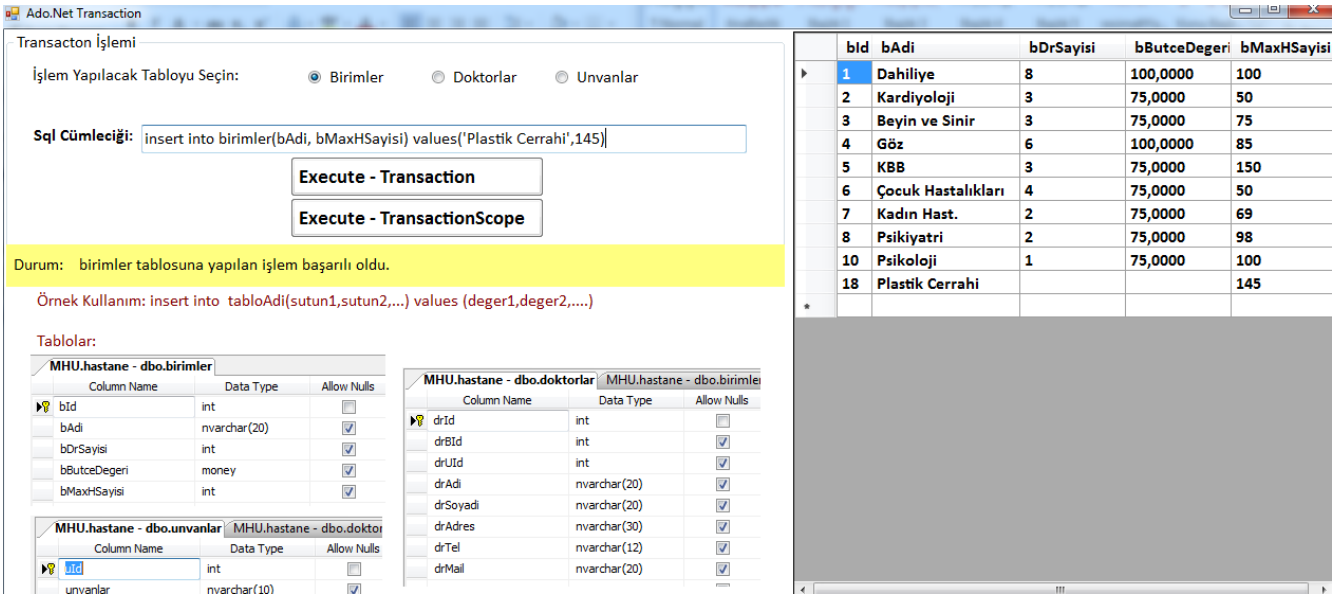
Bu ifade ile transaction nesnesinde yapılan tüm işlemleri gerialıyoruz. Çünkü bu blok çalışıyorsa bir hata meydana gelmiş demektir. }

Örnek Uygulama:

```
2 references
private void btnExec_Click(object sender, EventArgs e)
{
    if (rb == "") MessageBox.Show("Önce eklenecek tabloyu seçin!", "Uyarı");
    else
    {

        sqlConnectionNesnesi.Open();
        SqlTransaction tranNesnesi = sqlConnectionNesnesi.BeginTransaction();
        SqlCommand cmd = new SqlCommand("" + txtCumle.Text + "", sqlConnectionNesnesi);
        cmd.Transaction = tranNesnesi;
        try
        {
            cmd.ExecuteNonQuery();
            tranNesnesi.Commit();
            lblDurum.Text = "" + rb + " tablosuna yapılan işlem başarılı oldu.";
            ilgiliTabloyuGetir();

        }
        catch (Exception hata)
        {
            MessageBox.Show(hata + "  hatası oluştu.");
            tranNesnesi.Rollback();
            lblDurum.Text = "Hatalı!";
            ilgiliTabloyuGetir();

        }
        finally
        {
            sqlConnectionNesnesi.Close();
            cmd.Dispose();
        }
    }
```

Resim1. Visual Studio' da C# Ado.Net ile transaction örneği

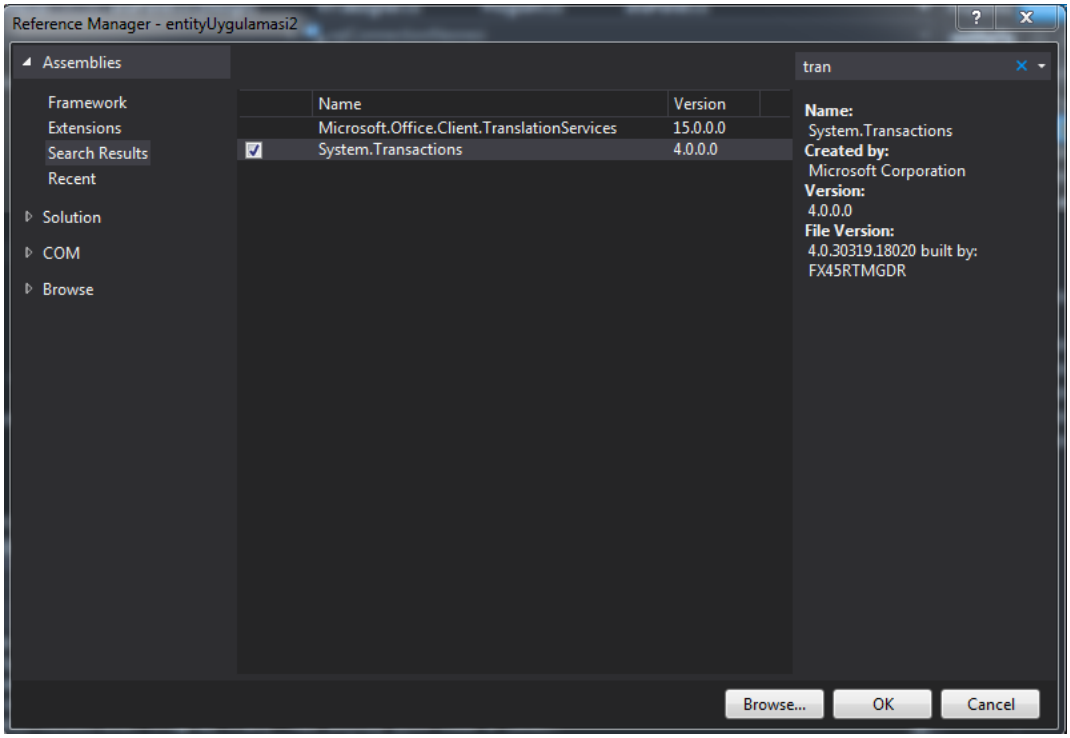
İşlem Sonucu:



Resim2. MS Sql Server' de oluşan değişiklik sonucu (bDrSayisi ve bButceDegeri alanlarına veri girilmedi. Server üzerinde ilgili birimde trgigger ile hesaplanarak very giriliyor.)

Bu işlemi TransactionScope sınıfını kullanarak daha az kodla gerçekleştirebiliriz. Transaction işlemini başarılı olduğunda işlemleri kalıcı yapmak için TransactionScope sınıfının Complete() metodu kullanılır. TransactionScope nesnesinin kullanım biçimi şu şekildedir;

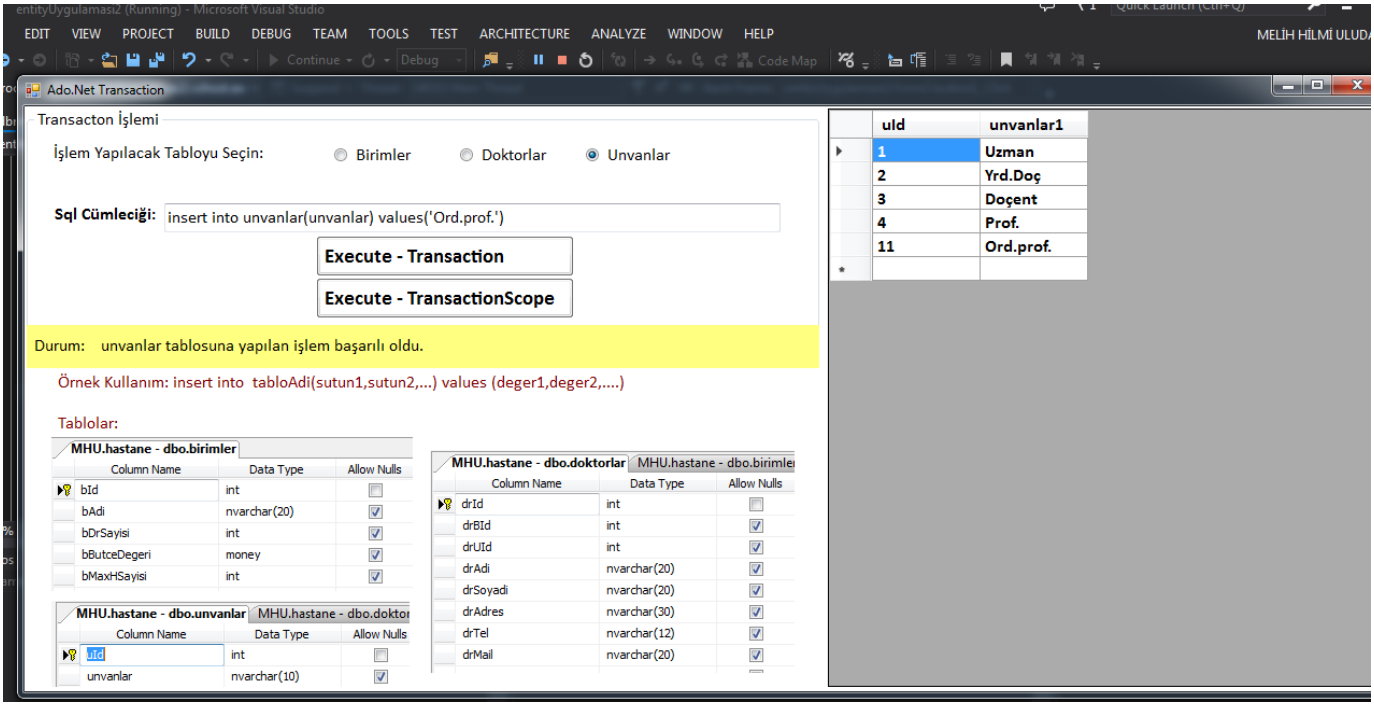
Öncelikle aşağıdaki gibi projemize "Solution Explorer-Add-Reference" yolunu izleyerek System.Transactions.dll dosyasını ekleyelim ki Transaction yönetimi için TransactionScope sınıfını kullanabilelim.



Resim3. ReferenceManager ile dll import etme

```
if (rb == "") MessageBox.Show("Önce eklenecek tabloyu seçin!", "Uyarı");
else
{
    using(TransactionScope ts=new TransactionScope()){
        SqlCommand cmd = new SqlCommand("" + txtCumle.Text + "", sqlConnectionNesnesi);
        // sql cümleciğini kullanıcıdan alıyorum
        sqlConnectionNesnesi.Open();
        try
        {
            cmd.ExecuteNonQuery();
            ts.Complete();
        }
        catch (Exception hata)
        {
            MessageBox.Show(hata + "  hatası oluştu.");
        }

        /* finally de database'yi kapatmaya gerek yok.
         *
         using(TransactionScope ts=new TransactionScope()){    satırı otomatik olarak
         * dispose ve close yi sağlıyor.
         */
    }


}
```

Resim4. TransactionScope kullanımı



Resim5. TransactionScope ile yapılan sql işlemi sonucu

Aynı şekilde iki farklı Command nesnesini de transaction yönetimine alabiliriz. Bu örnekte tek veri kaynağı üzerinde işlem yapıldığı için **LightWeight Transaction** tipi kullanılmış oldu.

TransactionScope nesnesinin asıl kolaylığını **OleTx Transaction** tipi üzerinde görmek mümkün. Aşağıdaki örnekte iki farklı veritabanı sunucusu üzerinde işlem yapılarak dağıtık transaction ortamı oluşturulmuştur. İki farklı veritabanı sunucusu üzerinde işlem yapıldığı için ADO.NET, programcının ek birşey yapmasına fırsat vermeyerek bu işlem dağıtık transaction yöntemini kullanacaktır.

2.2. OleTx Transaction Modeli

Bu ise dağıtık transaction yapısında olup ayrı veri kaynakları üzerinde işlem yapılırken kullanılan modeldir.

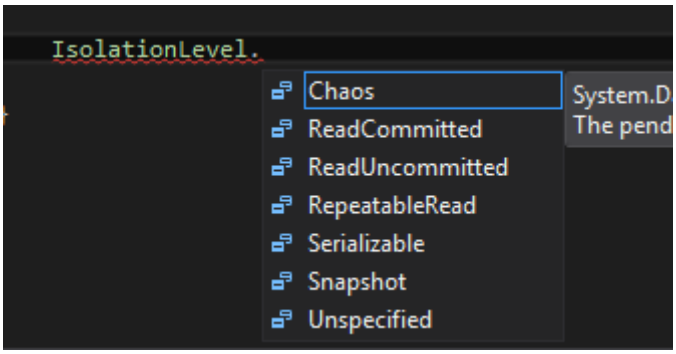
```
private void button1_Click_1(object sender, EventArgs e)
{
    using (TransactionScope oTrs = new TransactionScope())
    {

        string CnnStr1 = "Server=Sunucu1;Database=DB;Uid=sa;Pwd=sa;";
        string CnnStr2 = "Server=Sunucu2;Database=DB;Uid=sa;Pwd=sa;";

        SqlConnection oCnn1 = new SqlConnection(CnnStr1);
        SqlConnection oCnn2 = new SqlConnection(CnnStr2);

        string sQry1 = "UPDATE Musteri SET AdSoyad='Ayşe Güler' WHERE MusteriId=3;";
        string sQry2 = "UPDATE Urun SET Fiyat='35' WHERE UrunId=12;";

        SqlCommand oCmd1 = new SqlCommand(sQry1, oCnn1);
        SqlCommand oCmd2 = new SqlCommand(sQry2, oCnn2);

        try
        {
            oCnn2.Open();
            oCnn1.Open();

            oCmd1.ExecuteNonQuery();
            oCmd2.ExecuteNonQuery();

            oTrs.Complete();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
```

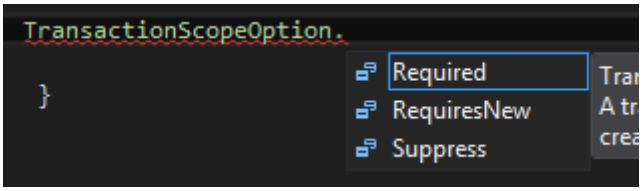
Resim6. OleTx Transaction modeli

Dağıtık transaction işlemleri Microsoft Distributed Transaction Coordinator(MSDTC) tarafından yönetilir. **Bu servisi doğru bir şekilde yapılandırdıktan sonra aynı application domain içerisindeki iki farklı sunucu bağlantısı üzerinde etkin bir transaction yönetimi gerçekleştirmiş oluruz.** DTC, her iki veritabanı sunucu üzerindeki DTC'lerden işlemleri doğru yapılıp yapılmadığına dair geri bildirim bekler. Gelen sonuç olumluysa TransactionScope nesnesi, Complete() edilir. Görüldüğü gibi ADO.NET 2.0'daki bu özellik, programcının **COM+ programlamayla uğraşmasına gerek kalmadan System.Enterprise servislerini kullanmadan dağıtık transaction işlemini daha az kodla kolaylıkla ve daha performanslı gerçekleştirmeyi sağlamıştır.**

TransactionScope sınıfının constructor'ü overload edilmiş olup aktif transaction'ın davranış biçimini ve zaman aşım süresini parametre olarak alır. Bu parametreler

daha çok iç içe kullanılmış transaction(nested transaction) yönetiminde işlev kazanmaktadır. Nested transaction, bir transaction'nın başarıyla sonuçlanması için başka bir transaction'ın başarıyla bitmesini beklemesidir. ADO.NET ile bu içiçe hareket işlemleri de kolaylıkla yapılabilmektedir. Nested transaction, direk veya dolaylı yapılabilir.





TransactionOptions sınıfı kullanılarak transaction bloğu için izolasyon seviyesi ve zaman aşımı süresi de tanımlanabilir. Aynı zamanda nested transaction işleminde içteki transaction blokları için hangi scope'da etkili olacağını belirleyebiliriz. Bunun için **TransactionScopeOption** enum'u kullanılır.

IsolationLevel, kullanıcı işlemlerin veya başka transaction'ın o anki transaction'a olan duyarlılığını yani aktif transaction çalışırken diğer transaction'ların aktif transaction yönetimindeki verilere nasıl davranacağını bildirir. Transaction'daki izolasyon seviyeleri (Isolation Level), aynı verilere yapılan ortak zamanlı erişim sorunlarını çözmek için kullanılır. Burada bu seçeneklerle ilgili bir iki cümle yazmak faydalı olacaktır.

Chaos: Değiştirmeye çalıştığımız veriler üzerinde başkası güncelleme yapıyorsa bizim değiştirmemize izin verilmez. Bu seçenek, SQL Server tarafından desteklenmemektedir.

ReadCommited: Okumaya çalıştığımız veriler üzerinde o anda başkası

güncelleme yapıyorsa o kişinin ancak COMMIT veya ROLLBACK ettiği verilerini okuyabiliriz. Örneğin bir tablodaki X satırını okuduk ve okuma transaction'u açık bıraktık. Bu arada başka bir transaction bu satırı güncelledi fakat COMMIT etmedi. Bu durumda okuma transaction'ı aynı satırı okumaya çalıştığında hata oluşur(Dirty Read sorunu). Transaction işlemi, default olarak ReadCommited seçeneğine sahiptir.

ReadUncommited: En düşük isolation seviyesi olup diğer transaction'ların henüz COMMIT edilmemiş verilerini de okuyabilmeyi sağlar. İkinci transaction tarafından yapılmış değişikliklerin birinci transaction tarafından okunabilmesi için ikinci transaction'ın COMMIT veya ROLLBACK yapması beklenmez. Yani önceki seçenekte bahsettiğimiz "Dirty Read" durumuna onay verir.

RepeatableRead: Diğer transaction'lara ait COMMIT edilmiş verilere erişilir ancak veriler okunurken diğer kullanıcıların o veriler üzerinde değişiklik yapmasını engeller. Yani SELECT işlemi çalışırken o anda başka bir kullanıcının UPDATE, DELETE işlemi yapması engellenir fakat INSERT yapmasına izin verilir.

Serializable: En yüksek isolation seviyesi olup verileri okurken veritabanını kilitleyerek diğer kullanıcıların hiçbir şekilde hiçbir ekleme, güncelleme yapmaması sağlar. Bir transaction'ın ekleme, güncelleme, silme yapabilmesi için aktif transaction'ın bitmesini bekler. UPDATE, DELETE, INSERT işlemi yapılırken hiçbir şekilde o anda başka bir kullanıcı aynı verileri SELECT edemez. Bu seçenek, performans açısından tercih edilmez ve aktif transaction uzun sürdüğü zaman deadlock denilen ölümcül kilitlenme sorunu yaşanabilir.

Diğer konu, iç içe olan transaction bloklarının faaliyet alanlarının belirlenmesidir. Bunun için**TransactionScopeOption** seçenekleri kullanılır. Bu seçenekler, içteki transaction'ın üstteki transaction'ın scope'ına eklenip eklenmeyeceğini bildirir;
Required: Varsa önceki transaction'a(Ambient Transaction) ait scope'a ekler yoksa yeni bir scope yaratır. Yani daha önce oluşturulmuş bir transaction varsa yeni işlemi o transaction'a dahil eder yoksa yeni bir transaction başlatır.
RequiresNew: Yeni bir transaction scope yaratır varsa öncekini askıya alır. Root scope kendisi olmuş olacak.

Suppress: Herhangi bir transaction içerisine dahil etmez.

```
private void btnEkle_Click_1(object sender, EventArgs e)
{
   using(KYDataContext kK=new KYDataContext()){

    doktorlar dr = new doktorlar();
    dr.drAdi = txtAdi.Text;
    dr.drSoyadi = txtSoyadi.Text;
    dr.drAdres = txtAdres.Text;
    dr.drMail = txtMail.Text;
    dr.drTel = txtTelNo.Text;
    dr.drBId = (int)cbBirim.SelectedValue;
    dr.drUId = (int)cbUnvan.SelectedValue;

       using(TransactionScope scope=new TransactionScope()){
        try
        {
            kK.doktorlars.InsertOnSubmit(dr); // ekleme işlemini yapar.
            kK.SubmitChanges(); // ekleme silme güncelleme gibi işlemleri gerçekleştirecek son cümlecik
                  //kK.doktorlar.InsertAllOnSubmit(                dizi şeklinde verileni topluca ekler
            scope.Complete();
        }
        catch (Exception hata)
        {
            MessageBox.Show(hata+" hatası oluştu");
        }

       }

    Drlistele();

    lblDurum.Text = txtAdi.Text + " isimli doktor  " + cbBirim.SelectedText + " birimine eklenmiştir.";
    txtAdi.Text = txtSoyadi.Text = txtAdres.Text = txtMail.Text = txtTelNo.Text = "";
   }
}
```

Resim7. EntityFreamwork6.0 da TransactionScope kullanımı

3. Sonuç

Ele aldığımız gibi basit veritabanı modellerinin söz konusu olduğu durumlarda **Transaction** kullanımına dikkat edilmemesi muhtemel olabilir. Hatta gerekmeyebilir. Nitekim Transaction oluşturmanın da veritabanı kaynağı üzerinde bir maliyeti vardır. Ancak **Enterprise** seviyedeki uygulamalar ve kullandıkları veritabanı sistemleri ile karmaşık ve bütünlüğü önem arz eden iş kuralları söz konusu olduğunda, **TransactionScope** kullanımına ciddi anlamda dikkat edilmesi gerekmektedir. Tabi bilindiği üzere **TransactionScope** nesne örneği üzerinden yapılabilecek farklı ayarlamalar da söz konusudur. Araştırmalar sonucu gördüğüm tavsiyelerde, söz konusu **farklı sunucular** veya **SQL Instance**' ları üzerindeki n sayıda veritabanında ele alacak şekilde değiştirmeniz ve gözlemlemenizdir.

4. Kaynaklar

- stackoverflow.com
- msdn.microsoft.com/en-us/library/
- simpleisbest.net
- codeproject.com
- buraksenyurt.com
- ismailgursoy.com.tr
- ahmetkaymaz.com