# **SQL’de HATALARI YÖNETME**

**Sql Server’da hata mesajlarını tanımlamak:**

Veri tabanı ile ilgili hata mesajlarını saklayıp sonrasında onları kontrol edebiliriz. Kısaca oluşan hataları log`layabiliriz. Bu hatalar içerisinde SQL Server ile ilgili (Constraint’lere uymayan duru olusmasi, …) hatalar olduğu gibi, sistemle ilgili (harddisk arazisasi, network arizasi,…) hatalarda olabilir. Bu hataları da fark etmemizi ilgili kişilere yönlendirmemizi sağlar.

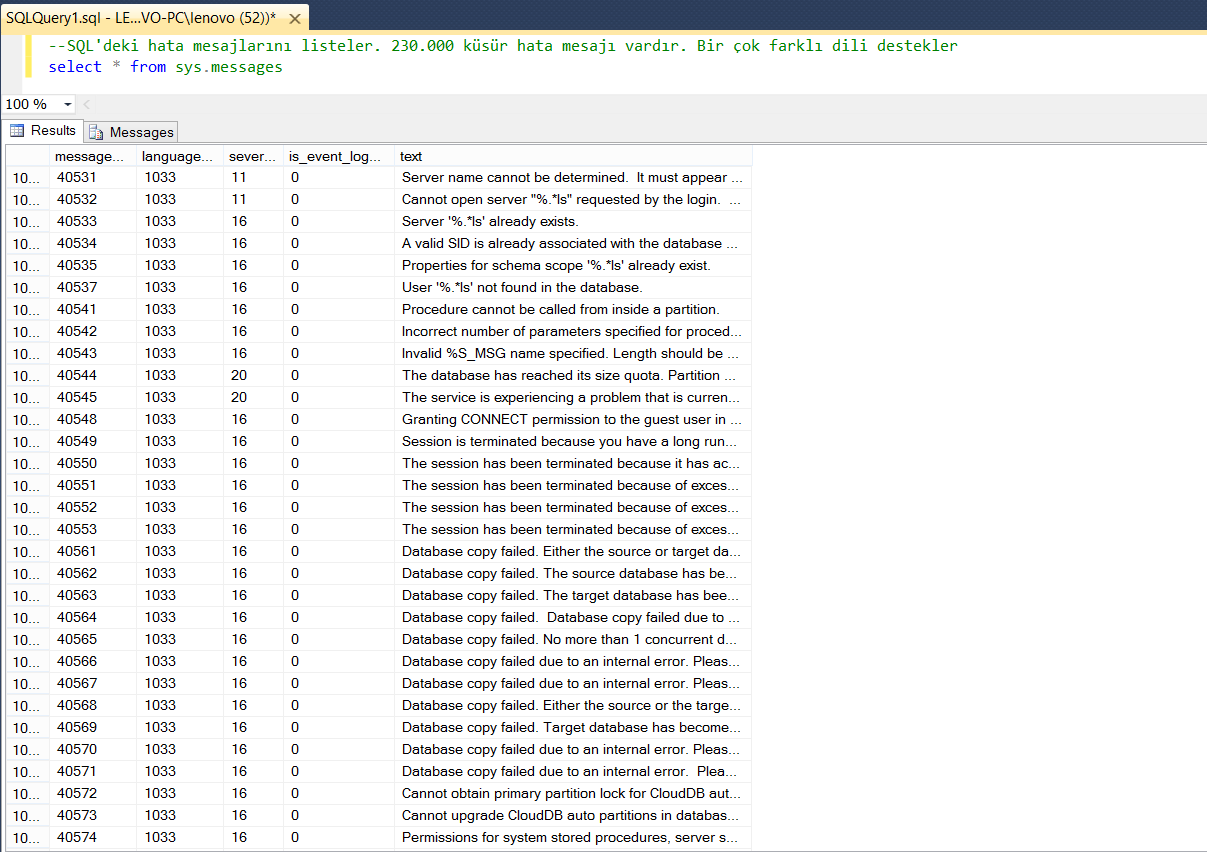
Mesela bir bankada ATM kullanıcıların kullanacağı ekranlarda çalışma yapıyorsun. Yaptığın çalışma kullanıcılarının hesaplarındaki detayı göstermek. Kodu kontrol ettiğinizde her şey yolunda görünüyor ama program çalışmıyor. Burada testler esnasında **hatayı yakalamaya yönelik önlemler** almazsanız hiçbir zaman hatanın nerede gerçekleştiğini bulamazsınız.

Bu tarz hataları almanız ve anlamamanız çok normal çünkü işe girdiğinizde var olan sistemlerde geliştirme yapacaksınız.

Biz SQL tarafından bakacak olursak, SQL’de tanımlı olan hata mesajlarını sys.messages isimli sistem tablosunda tutar. Bu tabloya ekleme yapılabilir. Yani kullanıcı kendisi de hata mesajı ekleyebilir. **@@error** fonksiyonu en son gerçekleşen hatanın kodunu tutar. **Raiserror** komutu ile SQL ortamına bilgi veriyoruz. Çalışma anında gerçekleşen hataları bulabilmek için ise TRY- Catch yapısını kullanıyoruz.

Sırasıyla gidecek olursak ;

## Öncelikle sistemde var olan hata mesajları için aşağıdaki sorguyu kullanabilirsiniz.



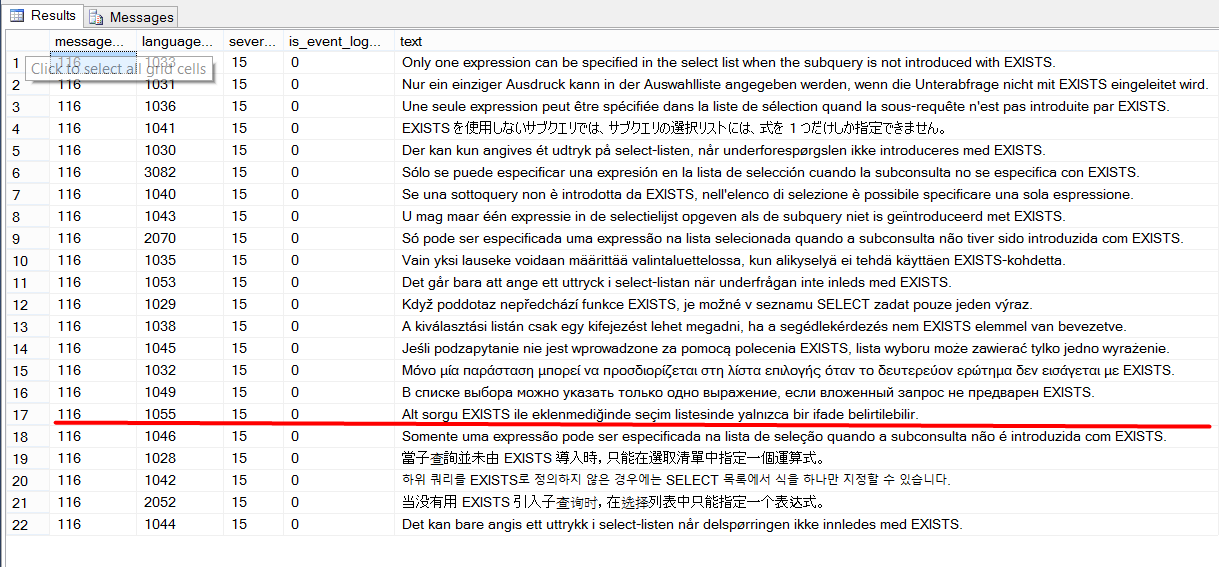
Sorgu sonucunda çıkan ekranda bulunan sutunlar:

**Message\_id**

* + Mesaj ID’sidir. Kullanıcı tanımlı olan hata mesajları 50.001 ile 2,147,483,647 değerleri arasında olmalıdır. Mesajları her dilde yazabileceğimizden Msg\_id ve language unique olmak zorundadır. Aynı ID’den hata mesajı girilirse hata verir.
  + 0-49.999 : Sistem hata mesajları için ayrılmıştır. **Kod geliştirirken karşımıza çıkan kodlar yani**
  + 50.000: anlık olarak Raiserror fonksiyonu ile üretilen hata mesajları için kullanılır.
  + 50.001: Kullanıcının (database yoneticisi, database programcisi) tanımlayabileceği mesaj kodları
* **Language \_id**
  + Belirtilen hata mesajının hangi dilde olduğunu belirtiyoruz. **Language sysname’dir ve default olarak null tanımlanmıştır.** Çünkü aynı server üzerinde birden fazla dil yüklüdür.
  + 1033,1046 vb. gibi numalar görürüz. Sistemde tanımlı olan dillerin grup kodudur.Bu kod hatanın hangi dile ait olduğunu gösterir.
  + Dilleri kullanabilmek için aşağıdaki sorguda bulunan name sütununu kullanabilirsiniz.

SELECT \* FROM sys.syslanguages

* **Text**
  + Ekranda gösterilecek olan hata mesajıdır.
  + Değişken türü nvarchar(255)’dir.
  + Default değeri NULL’dır.
  + Dışarıdan parametre alarak da yapılabilir.
* **Is\_event\_logged**
  + 0 ya da 1 değerini alır. Gerçekleşen hata loglandı mı bunun cevabını verir.
  + Değişken türü Varchar(5) ‘dir.
  + Default olarak tanımlanan değeri false’dur.Eğer True dersek herhangi bir hata gerçekleştiğinde \*Windows application log’a yazılacak. False olduğunda ise sadece kritik hata meydana geldiğinde loglama islemi yapacak. **Sysyadmin** server rolune sahip kişiler bu seçeneği kullanabilirler.
* **Severity (Örnek Bul)**
  + Hatanın seviyesini gösterir.
  + Değişken türü smallint’tir.
  + Default olarak tanımlanan değeri Null’dır .
    - 0 – 10 : Kullanıcının girdiği verilerden kaynaklanan bir hata
    - 11 – 16 :Kullanıcının düzeltebileceği bir hatadır.
    - 17 : Disk dolu, tablo okumaya karşı korumalı gibi. (Örnek bul)
    - 18 : Yazılımdan kaynaklanan hatalardır.
    - 19 : SQL Server’in kısıtlamalarına takılması sonucunda ortaya çıkan hatadır.
    - 20 – 25 : Sadece admin durumundaki kişiler bu tür hataları ekleyebilirler.

select \* from sys.messages where message\_id=116

## **Sisteme hata mesajı ekleme**

**Kullanıcı tarafından tanımlanacak olan hata mesajlarını eklemek için sistem tarafından hazırlanan store proceduru kullanabilirsiniz.**

sp\_addmessage @msgnum = msg\_id ,

@severity = severity ,

@msgtext = 'msg' ,

@lang = 'language' ,

@with\_log = { 'TRUE' | 'FALSE'},

@replace= 'replace'

EXEC sp\_addmessage 50005, 16,

'İstediğiniz kitap stoklarımızda bulunmamaktadır.','Türkçe';

EXEC sp\_addmessage 50005, 16,

'Libro desiderata non è disponibile a magazzino.','Italiano';

select \* from sys.messages where message\_id=50004

**Tanımlanan hata mesajlarını yine sistem tarafından tanımlanan store procedure ile silebilirsiniz.**

sp\_dropmessage @msgnum = msg\_id,@lang ='language'

exec sp\_dropmessage 50005,'Italiano'

Oluşturduğunuz dili belirterek silmeniz gerekmektedir.

# **Hata mesajlarını kullanmak**

Hata mesajlarını **Raiserror** ve **Throw** kelimeleri ile kullanıcıya ya da geliştiriciye gönderebiliriz.

## RAISERROR

Sisteme eklediğimiz hata mesajları biz herhangi bir işlem yapmazsak kullanıcının karşısına çıkmaz. Kullanıcının karşısına çıkartmak için **Raiserror** fonksiyonunu kullanmamız gerekir.

RAISERROR (

msg\_id | msg\_str | @local\_variable ,

severity ,state ,

argument [ ,...n ] ) WITH option [ ,...n ]

Bilinmeyen kısımlarından bahsedeceğim.

**msg\_str:** Kullanıcı tanımlı mesajdır. C kütüphanesinde bulunan printf fonksiyonuna benzer yazım tarzı bulunur. Hata mesajı max. 2047 karakter barındırır. Eğer 2048 karakter olur veya daha fazlası olursa ilk 2047 karakter görünür sonrası kesilir.

Msg\_str sadece 50000 numaralı hata fırlatılırken kullanılır. Yani anlık olarak oluşturulacak hatalarda kullanılır. Mesaj içeriğine dışarıdan parametrede çekebiliriz. Desteklediği veri tipleri: tinyint, smallint int char, varchar, nvarchar, binary, varbinary

|  |  |
| --- | --- |
| **Type specification** | **Represents** |
| d or i | Signed integer (tam sayı) |
| o | Unsigned octal (işaretsiz sekizlik tam sayı) |
| s | String |
| u | Unsigned integer (işaretsiz tam sayı) |
| x or X | Unsigned hexadecimal (İşaretsiz 16lık sayı) |

Genelde d , i ve s kısaltmaları kullanılıyor.

**@local\_variable :** Char veya varchar olarak kullamılır.

**State:** 0 ile255 değerleri isteğe bağlı parametreler olup,bir hata mesajı birden fazla yerde oluştuğunda bu yerleri birbirinden ayırt etmeye yarar.

Anlık olarak yazılmış, yani **sys.messages view’ından gelmeyen bir mesajı** hata mesajı olarak göstermek için **Raiserror** fonksiyonu kullanılır.

**Option:**

|  |  |
| --- | --- |
| **Deger** | **Tanımlama** |
| LOG | Microsoft SQL Server Database Engine’nin içerisine program içerisinde gerçekleşen logları kaydeder. Loglanan hataların limiti **max 440 byte’dır**. Bu hataları sadece server role hakkına sahip sistem yöneticisi düzeltebilir. |
| NOWAIT | Kullanıcıya hemen hata mesajını gönderir. |
| SETERROR | Hata seviyesini dikkate almadan hatanın içerisindeki değerleri @@ERROR ve ERROR\_NUMBER’e aktarır. |

Örnek1:

RAISERROR ('This is message %s %d.', -- Message text.

10, -- Severity,

1, -- State,

'number', -- %s string.

5)WITH LOG; -- %d decimal

Örnek2:

exec sp\_addmessage

@msgnum=50007,

@severity=11,

@msgtext='Üzgünüz , %s isimli bir kategori bulunmamaktadır',

@with\_log='true',

@lang='us\_english'

DECLARE @categoryName varchar(20)

SELECT @categoryName=CategoryName from Categories

IF @categoryName IS NULL

BEGIN

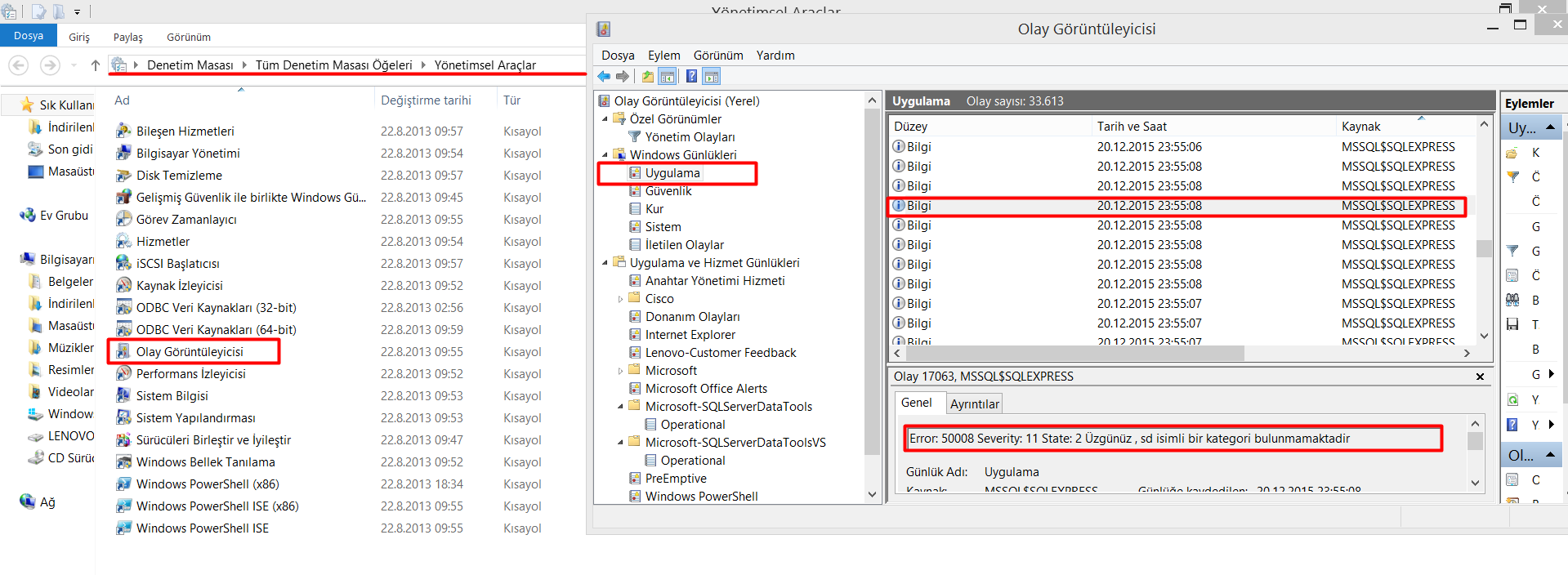
RAISERROR(50007,11,2,'sd')

END

Windows Aplication Logda loglanan hataları görmek için sırayla aşağıdaki adımları uygulayın.

* Denetim Masası(Control Panel)’ne gidin.
* Yönetimsel araçlar(Administrative Tools)’na tıklayın.
* Olay Görüntüleyicisi(Event Viewer)’ne tıklayın.
* Windows Log’a tıklayın.
* Application’na tıklayın.

Listelenen loglardan kendi hatanızı bulabilirsiniz.



Hatanın detayına üstüne çift tıklayarak erişebilirsiniz.



## **THROW**

Raiserror gibi Throw ifadesi de hata fırlatmak için kullanılır. Eğer Throw catch blogunda kullanılırsa parametre verilmez yakaladigi hatayi disari gönderir.

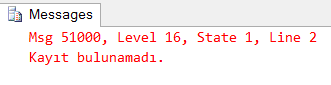
THROW error\_number | @local\_variable,

        message | @local\_variable,

        state | @local\_variable;

THROW 51000, 'Kayıt Bulunamadı.', 1;

Çıktısı aşağıdaki gibidir.

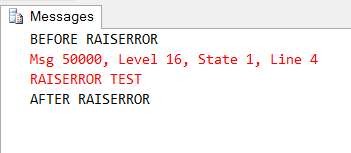


Throw ile hata fırlatıldığında hata seviyesi her zaman 16’dır. % karakteri Throw komutundaki mesaj kısmında kullanılan ozel bir karakterdir. Raiseerror’da kullandığımız gibi parametreleri burada da kullanabiliriz.

## **THROW ile RAISERROR arasındaki farklar**

|  |  |
| --- | --- |
| RaIse Error | Throw |
| Msg\_id direkt yazildiysa, Msg\_id sys.messages’de tanımlı olmalı. | Error\_Number’a gönderilen parametre sys.messages’de tanımlı olmak zorunda değil |
| Msg\_str ifadesi printf yazım stilinde yazılabilir. | Message kısmında printf yazım stili kabul edilemez. |
| Tüm hata seviyelerini kabul eder. | Hata seviyesi yoktur. Tüm hataları 16 olarak fırlatır. |

* Raiserror hata fırlattıktan sonra da çalışmaya devam eder



BEGIN

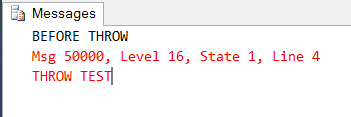
PRINT 'BEFORE RAISERROR'

RAISERROR('RAISERROR TEST',16,1)

PRINT 'AFTER RAISERROR'

END

* Throw ile hata fırlattığınızda sonrasında çalışmaya devam etmez.

BEGIN

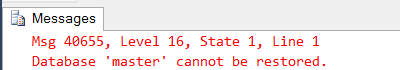
PRINT 'BEFORE THROW'

THROW 50000,'THROW TEST',1

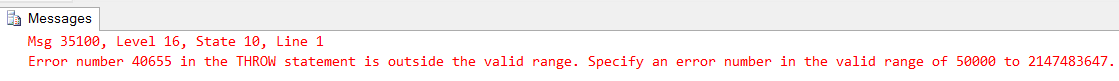
PRINT 'AFTER THROW'

END

* Throw’u tek başına kullandığınızda sonuna ‘;’ işareti koymak durumundasınız.
* Raiserror ile sistem hatalarını fırlatabiliriz. Ama throw ile yapamayız.

  
RAISERROR (40655,16,1) ;

THROW 40655, 'Database master cannot be restored.', 1



Bazen hatalarda belli şablon içerisinde isim, sayı gibi bazı değerlerin dışarıdan alınabilmesine ihtiyaç duyarız. Raiserror ile doğaçlama hata mesajı oluştururken kullanabiliriz.

## **Oluşan son hatanın kodunun yakalamak: @@Error**

Sistemde bir hata oluştuğunda hata kodu @@error tarafından yakalanır. İçerisinde bulunan eski hata kodu silinir ve yeni değerini tutmaya başlar. Dolayısıyla iç içe sorgularda hata kontrolu yaparken içine düşen hatanın başka bir değişkene aktarılması sonrasında @@error’un kullanımına devam etmesi önerilir.

Not: @@Rowcount degeride her çalıştığı anda değeri yenilenir.

**Açıklama:** Bu hataların genel kullanımını ben procedurlerde gördüm. Dolayısıyla procedurle yapmacağım.

Not: Bir store procedur aynı anda sadece 1 hata koduna erişebilir. Çünkü @@error en son oluşan hatanın kodunu tutar.

create proc spCalisanEkle(

@calisanID int,

@calisanAdi varchar(50),

@calisanSoyadi varchar(50),

@notlar varchar(550)

) as

insert Employees (EmployeeID, FirstName, LastName, Notes) values (@calisanID,@calisanAdi,@calisanSoyadi,@notlar)

if @@error<>547

print 'Yanlış kullanıcı Idsi girdiniz'

else if @@error <>0

print 'Bilinmeyen bir hata oluştu'

exec spCalisanEkle 2,'Kadriye','Doğutas','not mu o da ne?'



Not: Uyarı dışında gerçekleşen hata primary key ile tanımlanan alana kendim değer girmeye çalıştım.

Not: hiç hata oluşmadıysa @@error’un değeri 0’dır.

## Hata kontrolü ve TRY CATCH Yapısı

C# ve C++ dillerinde bulunan hata yakalama kodlarına benzer yapıda olan SQL’de bulunan hata yakalama kodudur.

TRY CATCH yapısı TRY ve CATCH bloğundan oluşur. Eğer TRY bloğunda hata seviyesi 10’dan büyük bir hata oluşursa kontrol CATCH bloğuna geçer. Bir hata oluşmamışsa CATCH bloğu devreye girmez. Bir hata varsa CATCH bloğunda hata yakalanır.  CATCH bloğunun işletilmesi tamamlandıktan sonra akış bloktan sonraki kodlarla devam eder.

BEGIN TRY

{   T-SQL kodları } ---Hata alınabilecek bir kod bloğu

END TRY

BEGIN CATCH

{  T-SQL kodları } ---Hata olduğunda hatanın yakalandığı kısım

END CATCH

[ ; ]

Try içerisindeki kodu her aşamada kontrol etmek yerine oluşacak hatalara önlemeyi catch kısmında yapıyoruz. Herhangi bir hata geldiğinde catch kısmındaki kodlar devreye giriyor.

Try blogu tanımlı ise mutlaka catch blogunu da tanımlamak zorundayız.

MS SQL'de TRY CATCH fonksiyonunun çeşitli işlevsel özellikleri vardır. Bu özellikler CATCH bloğu içinde kendi değerlerini korurlar, CATCH bloğu dışında ise geriye NULL dönerler.

|  |  |
| --- | --- |
| **Fonksiyon** | **İşlevleri** |
| ERROR\_NUMBER () : | Hata numarasını elde eder. |
| ERROR\_SEVERITY () | Hata derecesini verir. |
| ERROR\_SEVERITY() | Hata Dereceleri   * 0 veya 10: Kullanıcı veri girişinden kaynaklanan hata * 11-16 arası: Kullanıcının düzeltebileceği bir hata * 17: Yetersiz kaynak hatası (Diskin dolu olması veya tablonun salt okunur olması vb.) * 18: Yazılımdan kaynaklanan hata * 19: Constraint’lere takılan bir hata * 20-25 arası: Kritik hatalar |
| ERROR\_STATE () | Hatanın kod değerini verir. |
| ERROR\_PROCEDURE () | Hataya neden olan trigger veya store procedurun ismini verir. |
| ERROR\_LINE () | Hatanın meydana geldiği satır numarasını verir. |
| ERROR\_MESSAGE () | Hatayı açıklayan hata mesajı oluşturmamızı sağlar. |

Bu fonksiyonlar Catch blogunun dışında kullandığınız takdirde null değer döner. Hata bilgilerine sadece catch bloğunda olduğu sürece ulaşılabilir.

# Transaction

Daha küçük parçalara ayrılamayan en küçük işlem yığınına Transaction denir. Bazı durumlarda birden fazla işlem bir bütünün parçası olarak düşünülür. Özellikle bir grup işlemin arka arkaya gerçekleşiyor olmasına rağmen, seri işlemler halinde ele alınması gerektiğinde kullanılır. Bu işlemlerden biri bile gerçekleşmese bu işlemlerin hepsi anlamsız kalır. Bu tür işlemlerin bir bütün olarak yapılması gerekir. Transaction bloğu içerisindeki işlemlerin tamamı gerçekleşinceye kadar hepsi gerçekleşmemiş varsayılır.

Kısaca Transaction bloğu ya hep ya hiç mantığı ile çalışır. Ya tüm işlemler düzgün olarak gerçekleşir ve geçerli kabul edilir veya bir kısım işlemler yolunda gitse bile, blok sona ermeden bir işlem bile yolunda gitmese hiçbir işlem olmamış kabul edilir.

Veri tabanı sistemlerini yüzeysel kullanan geliştiriciler için çok basit gelen işlemler aslında mimari olarak geri planda birçok iş sürecinin doğru yürütülmesiyle meydana gelmektedir. Bu iş süreçleri veri tabanı motoru tarafından yönetilir. Ancak, depoladığımız veriyle ilgili işlemlerin bütünlüğü tamamen bize, yani geliştiriciye aittir.

## Transaction Prensipleri

Veri tabanı mimarisinde bu tür iş süreci(transaction) gerektiğinde uyulması gereken prensibe ACID (Atomicity,Consistency, Isolation, Durability) denir.

#### Bölünemezlik(Atomicity)

Bir transaction bloğu yarım kalamaz. Yarım kalan transaction bloğu veri tutarsızlığına neden olur. Ya tüm işlemler gerçekleştirilir, ya da transaction başlangıcına geri döner. Yani transaction’ın gerçekleştirdiği tüm değişiklikler geri alınarak gerçekleşmeden önceki haline döner.

#### Tutarlılık(Consistency)

Bölünemezlik kuralının alt yapısını oluşturduğu bir kuraldır. Transaction veri tutarlılığı sağlamalıdır. Yani bir transaction içerisinde güncelleme işlemi gerçekleştiyse ve ya kalan tüm işlemler de gerçekleşmeli ya da güncelle işlemi de geri alınmalıdır. Bu veri tutarlılığı açısından çok önemlidir.

#### İzolasyon(Isolation)

Her transaction veri tabanı için bir istek paketidir. Bir istek paketi (transaction) tarafından gerçekleştirilen değişiklikler tamamlanmadan bir başka transaction tarafından görülememelidir. Her transaction ayrı olarak işlenmelidir. Transaction’ın tüm işlemleri gerçekleştikten sonra bir başka transaction tarafından görülebilmelidir.

#### Dayanıklılık (Durability)

Transaction’lar veri üzerinde karmaşık işlemler gerçekleştirebilir. Bu işlemlerin bütününü güvence altına almak için transaction hatalara karşı dayanıklı olmalıdır. SQL Server’da meydana gelebilecek sistem sorunu, elektrik kesilmesi, işletim sisteminden ya da farklı yazılımlardan kaynaklanabilecek hatalara karşı hazırlıklı ve dayanıklı olmalıdır.

Bilgisayar bilimlerinde imkânsız diye bir şey yoktur. Bir sistemin hata vermesi yüzlerce sebebe bağlı olabilir. Örneğin; geçtiğimiz yıllarda büyük bir GSM operatörünün veri merkezinin bulunduğu binaya sel basması sonucu veri kaybı yaşamışlardı. Bu durum olasılığı düşük gibi görünebilir. Ancak bahsi edilen konu önemli veriler olunca, hesaplanması gereken olasılıkların sınırı yoktur. Tüm hata olasılıklarına karşı dayanıklı bir sistem geliştirilmelidir.

## **Hangi durumlarda transaction kullanılır?**

Bir banka uygulamasını düşünün. Bir kullanıcı başka bir kullanıcıya havale yaptığında ne olacağına bakalım. Öncelikle havale yapanın hesap bilgilerinden havale yaptığı miktar düşülür. Ardından alıcının hesabına bu miktar eklenir ve havale gerçekleşmiş olur. Ancak her zaman şartlar istendiği gibi olmayabilir.

Mesela, gönderenin hesabından para düşüldüğü anda elektrik kesilebilir ya da program takılabilir. Bu durumda, ne olur? Örneğin, gönderenin hesabından para düşülmüştür ama alıcının hesabına da geçmemiştir. Bu aşamada para gönderenden çıktığı için onun bilgilerine erişemez. Alıcıya ulaşmadığından onun bilgilerine de erişilemez. Yani paranın sahibinin kimliği kaybedilmiş olduk. Bu da sistemin olası durumlar dışında veri kaybetmeye müsait bir hal alması demektir. Bu durumun bir şekilde önlenmesi gerekir. İşte bu gibi durumlarda biz Transaction’ı kullanıyoruz.

* Veri tabanının içindeki bir tablodan veri silinmesi,
* Silinen verinin başka bir tabloda eklenmesi
* Verinin güncellenmesi gibi durumlarda kullanılır.

Yapılan bu işlemler birbirleriyle ilintili ve ardışık işlemler olduğu için bir tanesinin gerçekleşmemesi demek çok önemli verilerin kaybına neden olmak demektir. Bu tarz sorunlarla karşılaşmamak için son işlem gerçekleşinceye kadar tüm işlemler beklemelidir, eğer bir hata oluşmazsa tüm işlemler geçerli kabul edilir ve sona yansır. Bu işlemler sırasında veriler üzerindeki değişikliklerin de tutarlı olması, birbirlerini tamamlayıcı nitelik taşımaları gerekmektedir. İşte bu noktada veri tabanı uygulamaları geliştirilirken bu tip iş parçacıklarının "transaction" bloklarının içine alıp sürecin daha güvenli ilerlemesini sağlıyoruz

**SQL Server 3 farklı transaction desteği sağlar:**

**1. Harici (Explicit) Transaction:** SQL Server’in kullanıcı tarafından bir BEGIN TRAN ifadesi ile transaction’a başlatılması şeklindeki bloktur. Bir aksilik olması halinde SQL Server tarafından veya kullanıcı tarafından COMMIT ifadesi ile gerçekleşmiş olarak veya ROLLBACK ifadesi ile hiç olmamış olarak sonlandırılabilir.

**2.Dâhili (Implict) Transaction:** SQL Server’in belli ifadelerden sonra otomatik olarak transaction açmasını sağlar. Bu modda, bu belli ifadeler kullanıldıktan sonra, kullanıcı tarafından transaction’ın sonlandırılması gerekir. Bu nedenle zahmetli bir mod’dur. **Hangi ifadeler?**

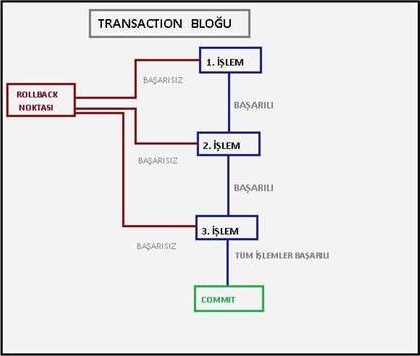
**3.Auto Commit:** Hiç bir transaction mod’u tayin edilmedi ise, SQL Server bu modda çalışır. Auto Commit modunda iken, her bir batch bir transaction bloğu olarak ele alınır. Batch içerisinde bir sorun olursa, SQL Server otomatik olarak bütün batch’i geri alır(ROLLBACK eder).

**Batch : (yığın, Query Analyzer için iki ‘go’ arasındaki ifade veya bir defada çalıştırılan bütün SQL ifadeleri)**

**Soru: Biz transaction kullanmasak ta otomatik olarak bizim yaptığımız hataları geri mi alıyor. Alıyorsa hangi hatalarda bunu yapıyor.**

Ancak biz genel olarak transaction denilince, harici transaction işlemlerini kastederiz.

## **Harici transaction işleyiş mantığı**

1.Transaction bloğu başlatılır. Böylece yapılan işlemlerin geçersiz sayılabileceği VTYS’ye bildirilmiş olur ve SQL Server Auto Commit modundan çıkıp, Explicit yani Harici moda geçer.

2.Transaction bloğu arasında yapılan her bir işlem bittiği anda başarılı olup olmadığına gerek varsa, programcı tarafından bakılıp, başarılı olmadığı anda geri alım işlemine geçilebilir(ROLLBACK). Ancak bir sorun olması halinde, SQL Server tarafından da **verilerin tutarlılığını** denetlemek üzere, **transaction** bloğunun başladığı andan itibaren bir **güç kesilmesi gibi** durum ortaya çıkarsa, değişiklikler dikkate alınmayacak şekildedir. Bu, transaction logları denilen yöntem ile yapılır.

**Transaction logları nasıl tutulur? Ldf**

Bu yöntemde, bir transaction başladıktan sonra, verileri tutan sayfalar **diskten(HDD) hafızaya(RAM)** yüklenir ve ilgili değişiklikler, önce hafızada yapılır. Ardından, değişikliklerin izdüşümü loglar diske yazdırılır, ardından veriler de güncellenir.

3.Tüm işlemler tamamlandığı anda COMMIT ile bilgiler yeni hali ile sabitlenir. Başarısız bir sonuç ise ROLLBACK ile en başa alınır ve bilgiler ilk hali ile sabitlenir.

## **Transaction ifadeleri**

Transaction yönetimi için kullanılan dört farklı ifade vardır.

* Transaction başlatılabilir (**BEGIN**),
* İşlemler geri alınabilir (**ROLLBACK**),
* Transaction bitirilebilir (**COMMIT**)
* Kayıt noktaları (**SAVE**) oluşturulabilir.

**Transaction’ı Başlatmak: BEGIN TRAN**

Transaction’ın başlangıcını belirtir. Bu kısımdan sonraki tüm işlemler transaction’ın bir parçasıdır. İşlem sırasında oluşabilecek olası sorunlarda geri alma ya da transaction’ın sonlandırılması gerçekleştirilebilir.

|  |
| --- |
| BEGIN TRAN[SACTION] [transaction\_ismi | @transaction\_degiskeni] |

**Transaction’ı Tamamlamak: COMMIT TRAN**

Transaction’ın tamamlandığını ve gerçekleştirilen transaction işlemlerinin kalıcı olarak veri tabanına yansıtılması için kullanılır. Transaction tarafından etkilenen tüm değişiklikler, işlemlerin tamamı gerçekleşmese bile, bu işlemden sonra kalıcı hale gelir.

COMMIT işleminden sonra gerçekleşen değişikliklerin geri alınması için, bu işlemleri geri alacak yeni bir transaction oluşturulmalıdır. Örneğin; bir transaction ile nümerik bir sütun üzerinde 10 birim azaltma işlemi yapıldı ise, bu işlemi geri almak için aynı sütun üzerinde 10 birim artırma işlemi yapacak yeni bir transaction oluşturulmalıdır.

|  |
| --- |
| COMMIT TRAN[SACTION] [transaction\_ismi | @transaction\_degiskeni] |

**Transaction’ı Geri Almak: ROLLBACK TRAN**

Transaction’ın gerçekleştirdiği tüm işlemleri geri almak için kullanılır. Yani, yapılan tüm işlemler transaction’ın başlangıcındaki haline geri döner. Verilerdeki değişikliklerin anında kalıcı olarak veri tabanına yansıtılmadığını belirtmiştik. ROLLBACK ile gerçekleştirilen tüm işlemler geriye alınarak transaction sonucunun tutarlılığı garanti edilir.

ROLLBACK işlemi, oluşturduğunuz transaction mimarisine bağlı olarak, kayıt noktalarına (save-points) geri dönüş için de kullanılabilir.

|  |
| --- |
| ROLLBACK TRAN[SACTION] [transaction\_ismi | kayit\_noktasi\_ismi  | @transaction\_degiskeni | @kayit\_noktasi\_degiskeni] |

**Sabitleme Noktaları: SAVE TRAN**

ROLLBACK işlemi transaction ’da en başa dönmek için kullanılır. Bazen de belirli bir noktaya kadar gerçekleşen işlemlerin geçerli kalması istenebilir. Bu işlemlerden sonra gerçekleşecek işlemler için ROLLBACK’e ihtiyaç duyulabilir. Sabitleme noktaları oluşturulması, transaction içerisinde en başa dönmek yerine, belirlenen bir işlem noktasına dönmek için kullanılır

|  |
| --- |
| SAVE TRAN[SACTION] [ kayit\_noktasi\_ismi | @kayit\_noktasi\_degiskeni ] |

**Örnek:**

Banka hesabı örneği

--banka hesabı için tablo oluştur.

CREATE TABLE BankaHesabi(

HesapNo CHAR(20) NOT NULL PRIMARY KEY,

Adi VARCHAR(55),

Soyadi VARCHAR(55),

Sube INTEGER,

Bakiye FLOAT

)

--hesapları gir

INSERT INTO BankaHesabi values('1','Kadriye','Doğutaş', 001, 10000)

INSERT INTO BankaHesabi values('2','Mehmet','Akay', 003, 5000)

INSERT INTO BankaHesabi values('3','Adem','Erbilen', 002, 5440)

INSERT INTO BankaHesabi values('4','Sultan','Susmaz', 003, 18900)

INSERT INTO BankaHesabi values('5','Zeynep','Eğilmez', 002, 3000)

INSERT INTO BankaHesabi values('6','Gamze ','Durmaz', 001, 6440)

--banka hesabındaki paranın miktarını değiştirip işlemleri kaydedicez. Sonrasında hesabı siliyoruz.

--Amacımız kayıt noktasına geri döndüğümüzde hesabı geri getirebiliyormuyuz.

--transactionu başlat

BEGIN TRANSACTION

--1 numaralı hesaptaki bakiyeyi 50000 yap.

UPDATE BankaHesabi

SET bakiye = 50000

WHERE hesapNo='1';

--son durumu kaydet

SAVE TRANSACTION svp\_kaydet

--Hesabı kapat- Sil yani. Buradaki amacımız kayıt noktasına geri döndüğümüzde hesap geri gelecekmi?

DELETE FROM BankaHesabi

WHERE HesapNo='1';

-- kayıt noktasına geri dön ve listele

ROLLBACK TRAN svp\_kaydet;

SELECT \* FROM BankaHesabi;

--2. örnek

BEGIN TRANSACTION

--Varolan bakiyeden 2.000 lirayı 1 numaralı hesaptan 2 numarali hesaba aktar.

UPDATE BankaHesabi

SET bakiye=bakiye - 2000

WHERE hesapNo=1

--hata gerçekleşmediyse @@error'un değeri 0'a eşittir. Her hangi bir hata gerçekleştiyse yani 0'a eşit değilse parayı geri aktar.

IF @@ERROR<>0

ROLLBACK

ELSE

BEGIN

UPDATE BankaHesabi

SET bakiye=bakiye + 2000

WHERE hesapNo=2

IF @@ERROR<>0

ROLLBACK

ENDIF

-- her şey tamamlandığında işlemleri onayla

COMMIT