**Suggested SQL Server videos**  
[Part 18 - Stored procedures](http://csharp-video-tutorials.blogspot.com/2012/08/stored-procedures-part-18.html)  
  
  
**With the introduction of Try/Catch blocks in SQL Server 2005**, error handling in sql server, is now similar to programming languages like C#, and java. Before understanding error handling using try/catch, let's step back and understand how error handling was done in SQL Server 2000, using system function **@@Error**. Sometimes, system functions that begin with two at signs **(@@)**, are called as **global variables**. They are not variables and do not have the same behaviours as variables, instead they are very similar to functions.  
  
Now let's create **tblProduct** and **tblProductSales**, that we will be using for the rest of this demo.  
  
**SQL script to create tblProduct**  
Create Table tblProduct  
(  
 ProductId int NOT NULL primary key,  
 Name nvarchar(50),  
 UnitPrice int,  
 QtyAvailable int  
)  
  
**SQL script to load data into tblProduct**  
Insert into tblProduct values(1, 'Laptops', 2340, 100)  
Insert into tblProduct values(2, 'Desktops', 3467, 50)  
  
**SQL script to create tblProductSales**  
Create Table tblProductSales  
(  
 ProductSalesId int primary key,  
 ProductId int,  
 QuantitySold int  
)   
  
   
  
Create Procedure spSellProduct  
@ProductId int,  
@QuantityToSell int  
as  
Begin  
 -- Check the stock available, for the product we want to sell  
 Declare @StockAvailable int  
 Select @StockAvailable = QtyAvailable   
 from tblProduct where ProductId = @ProductId  
   
 -- Throw an error to the calling application, if enough stock is not available  
 if(@StockAvailable < @QuantityToSell)  
   Begin  
 Raiserror('Not enough stock available',16,1)  
   End  
 -- If enough stock available  
 Else  
   Begin  
    Begin Tran  
         -- First reduce the quantity available  
 Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)  
 where ProductId = @ProductId  
   
 Declare @MaxProductSalesId int  
 -- Calculate MAX ProductSalesId   
 Select @MaxProductSalesId = Case When   
 MAX(ProductSalesId) IS NULL   
 Then 0 else MAX(ProductSalesId) end   
 from tblProductSales  
 -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation  
 Set @MaxProductSalesId = @MaxProductSalesId + 1  
 Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)  
    Commit Tran  
   End  
End  
  
1. **Stored procedure - spSellProduct**, has 2 parameters - **@ProductId** and **@QuantityToSell**. @ProductId specifies the product that we want to sell, and @QuantityToSell specifies, the quantity we would like to sell.   
  
2. Sections of the stored procedure is commented, and is self explanatory.  
  
3. In the procedure, we are using **Raiserror**() function to return an error message back to the calling application, if the stock available is less than the quantity we are trying to sell. We have to pass atleast 3 parameters to the Raiserror() function.  
**RAISERROR('Error Message', ErrorSeverity, ErrorState)**  
**Severity and State** are integers. In most cases, when you are returning custom errors, the severity level is 16, which indicates general errors that can be corrected by the user. In this case, the error can be corrected, by adjusting the **@QuantityToSell**, to be less than or equal to the stock available. ErrorState is also an integer between 1 and 255. RAISERROR only generates errors with state from 1 through 127.  
  
4. The problem with this procedure is that, the **transaction is always committed**. Even, if there is an error somewhere, between updating **tblProduct** and **tblProductSales** table. In fact, the main purpose of wrapping these 2 statments (Update tblProduct Statement & Insert into tblProductSales statement) in a transaction is to ensure that, both of the statements are treated as a single unit. For example, if we have an error when executing the second statement, then the first statement should also be rolledback.    
  
   
  
In SQL server 2000, to detect errors, we can use **@@Error** system function. @@Error returns a NON-ZERO value, if there is an error, otherwise ZERO, indicating that the previous sql statement encountered no errors. The stored procedure **spSellProductCorrected**, makes use of @@ERROR system function to detect any errors that may have occurred. If there are errors, roll back the transaction, else commit the transaction. If you comment the line (Set @MaxProductSalesId = @MaxProductSalesId + 1), and then execute the stored procedure there will be a primary key violation error, when trying to insert into **tblProductSales**. As a result of this the entire transaction will be rolled back.  
Alter Procedure spSellProductCorrected  
@ProductId int,  
@QuantityToSell int  
as  
Begin  
 -- Check the stock available, for the product we want to sell  
 Declare @StockAvailable int  
 Select @StockAvailable = QtyAvailable   
 from tblProduct where ProductId = @ProductId  
   
 -- Throw an error to the calling application, if enough stock is not available  
 if(@StockAvailable < @QuantityToSell)  
   Begin  
 Raiserror('Not enough stock available',16,1)  
   End  
 -- If enough stock available  
 Else  
   Begin  
    Begin Tran  
         -- First reduce the quantity available  
 Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)  
 where ProductId = @ProductId  
   
 Declare @MaxProductSalesId int  
 -- Calculate MAX ProductSalesId   
 Select @MaxProductSalesId = Case When   
 MAX(ProductSalesId) IS NULL   
 Then 0 else MAX(ProductSalesId) end   
 from tblProductSales  
 -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation  
 Set @MaxProductSalesId = @MaxProductSalesId + 1  
 Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)  
 if(@@ERROR <> 0)  
 Begin  
 Rollback Tran  
 Print 'Rolled Back Transaction'  
 End  
 Else  
 Begin  
 Commit Tran   
 Print 'Committed Transaction'  
 End  
   End  
End  
  
**Note**: @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.  
  
In **tblProduct** table, we already have a record with **ProductId = 2**. So the insert statement causes a primary key violation error. @@ERROR retains the error number, as we are checking for it immediately after the statement that cause the error.

Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
if(@@ERROR <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'  
  
On the other hand, when you execute the code below, you get message**'No Errors'**printed. This is because the @@ERROR is cleared and reset on each statement execution.   
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
--At this point @@ERROR will have a NON ZERO value   
Select \* from tblProduct  
--At this point @@ERROR gets reset to ZERO, because the   
--select statement successfullyexecuted  
if(@@ERROR <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'  
  
In this example, we are storing the value of @@Error function to a local variable, which is then used later.  
Declare @Error int  
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
Set @Error = @@ERROR  
Select \* from tblProduct  
if(@Error <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'