How to implement error handling in SQL Server

June 15, 2018 by Bojan Petrovic

Error handling overview

Error handling in SQL Server gives us control over the Transact-SQL code. For example, when things go wrong, we get a chance to do something about it and possibly make it right again. SQL Server error handling can be as simple as just logging that something happened, or it could be us trying to fix an error. It can even be translating the error in SQL language because we all know how technical SQL Server error messages could get making no sense and hard to understand. Luckily, we have a chance to translate those messages into something more meaningful to pass on to the users, developers, etc.

In this article, we'll take a closer look at the TRY... CATCH statement: the syntax, how it looks, how it works and what can be done when an error occurs. Furthermore, the method will be explained in a SQL Server case using a group of T-SQL statements/blocks, which is basically SQL Server way of handling errors. This is a very simple yet structured way of doing it and once you get the hang of it, it can be quite helpful in many cases.

On top of that, there is a RAISERROR function that can be used to generate our own custom error messages which is a great way to translate confusing error messages into something a little bit more

```
END TRY
BEGIN CATCH
     --code to run if an error occurs
--is generated in try
END CATCH
```

Anything between the BEGIN TRY and END TRY is the code that we want to monitor for an error. So, if an error would have happened inside this TRY statement, the control would have immediately get transferred to the CATCH statement and then it would have started executing code line by line.

Now, inside the CATCH statement, we can try to fix the error, report the error or even log the error, so we know when it happened, who did it by logging the username, all the useful stuff. We even have access to some special data only available inside the CATCH statement:

- ERROR_NUMBER – Returns the internal number of the error
- ERROR_STATE – Returns the information about the source
- ERROR_SEVERITY – Returns the information about anything from informational errors to errors user of DBA can fix, etc.
- ERROR_LINE – Returns the line number at which an error happened on
- ERROR_PROCEDURE – Returns the name of the stored procedure or function
- ERROR_MESSAGE – Returns the most essential information and that is the message text of the error

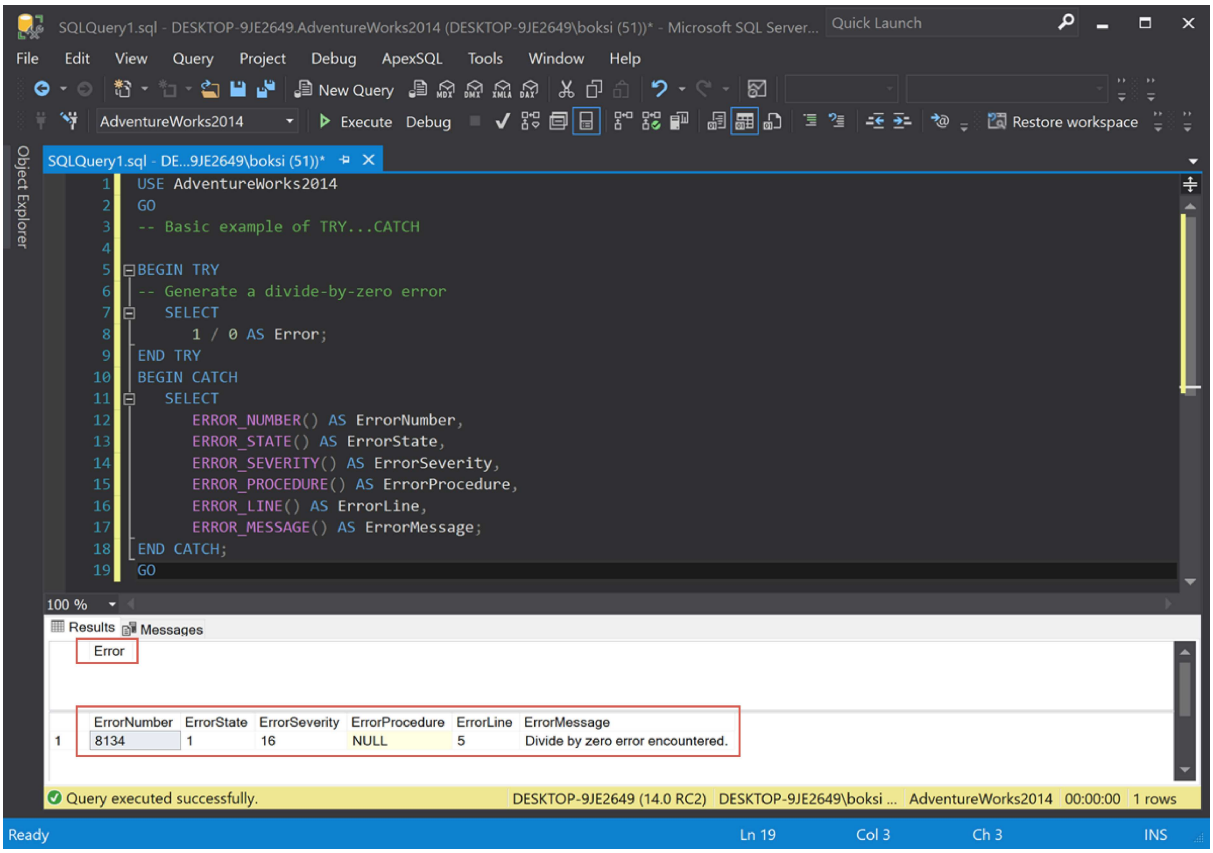
That's all that is needed when it comes to SQL Server error handling. Everything can be done with a simple TRY and CATCH statement and the only part when it can be tricky is when we're dealing with transactions. Why? Because if there's a BEGIN TRANSACTION, it always must end with a COMMIT or ROLLBACK transaction. The problem is if an error occurs after we begin but before we commit or rollback. In this particular case, there is a special function that can be used in the CATCH statement that allows checking whether a transaction is in a committable state or not, which then allows us to make a decision to rollback or to commit it.

Let's head over to SQL Server Management Studio (SSMS) and start with basics of how to handle SQL Server errors. The AdventureWorks 2014 sample database is used throughout the article. The script below is as simple as it gets:

```
USE AdventureWorks2014
GO
```

```
    ERROR_PROCEDURE() AS ErrorProcedure,
    ERROR_LINE() AS ErrorLine,
    ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

This is an example of how it looks and how it works. The only thing we're doing in the BEGIN TRY is dividing 1 by 0, which, of course, will cause an error. So, as soon as that block of code is hit, it's going to transfer control into the CATCH block and then it's going to select all of the properties using the built-in functions that we mentioned earlier. If we execute the script from above, this is what we get:



We got two result grids because of two SELECT statements: the first one is 1 divided by 0, which causes the error and the second one is the transferred control that actually gave us some results. From left to right, we got ErrorNumber, ErrorState, ErrorSeverity; there is no procedure in this case (NULL), ErrorLine, and ErrorMessage.

Now, let's do something a little more meaningful. It's a clever idea to track these errors. Things that

```
-- Table to record errors

CREATE TABLE DB_Errors
         (ErrorID         INT IDENTITY(1, 1),
          UserName        VARCHAR(100),
          ErrorNumber     INT,
          ErrorState      INT,
          ErrorSeverity   INT,
          ErrorLine       INT,
          ErrorProcedure VARCHAR(MAX),
          ErrorMessage    VARCHAR(MAX),
          ErrorDateTime   DATETIME)
GO
```

Here we have a simple identity column, followed by username, so we know who generated the error and the rest is simply the exact information from the built-in functions we listed earlier.

Now, let's modify a custom stored procedure from the database and put an error handler in there:

```
ALTER PROCEDURE dbo.AddSale @employeeid INT,
                   @productid  INT,
                   @quantity   SMALLINT,
                   @saleid     UNIQUEIDENTIFIER OUTPUT
AS
SET @saleid = NEWID()
  BEGIN TRY
    INSERT INTO Sales.Sales
         SELECT
           @saleid,
           @productid,
           @employeeid,
           @quantity
  END TRY
  BEGIN CATCH
    INSERT INTO dbo.DB_Errors
    VALUES
  (SUSER_SNAME(),
   ERROR_NUMBER(),
   ERROR_STATE(),
   ERROR_SEVERITY(),
   ERROR_LINE(),
   ERROR_PROCEDURE(),
   ERROR_MESSAGE(),
   GETDATE());
  END CATCH
GO
```

Altering this stored procedure simply wraps error handling in this case around the only statement in-

sults grid will be populated differently:

This time, we got two indicators in the results grid:

0 rows affected – this line indicated that nothing actually went into the Sales table

1 row affected – this line indicates that something went into our newly created logging table

So, what we can do here is look at the errors table and see what happened. A simple Select statement will do the job:

Here we have all the information we set previously to be logged, only this time we also got the procedure field filled out and of course the SQL Server "friendly" technical message that we have a violation:

Violation of PRIMARY KEY constraint 'PK_Sales_1'. Cannot insert duplicate key in object' Sales.Sales'. The duplicate key value is (20).

How this was a very artificial example, but the point is that in the real world, passing an invalid date is very common. For example, passing an employee ID that doesn't exist in a case when we have a foreign key set up between the Sales table and the Employee table, meaning the Employee must exist in order to create a new record in the Sales table. This use case will cause a foreign key constraint violation.

The general idea behind this is not to get the error fizzled out. We at least want to report to an individual that something went wrong and then also log it under the hood. In the real world, if there was an application relying on a stored procedure, developers would probably have SQL Server error handling coded somewhere as well because they would have known when an error occurred. This is also where it would be a clever idea to raise an error back to the user/application. This can be done by adding the RAISERROR function so we can throw our own version of the error.

For example, if we know that entering an employee ID that doesn't exist is more likely to occur, then

with them. We can use the same procedure as before, only this time let's wrap a transaction around the Insert statement:

```
ALTER PROCEDURE dbo.AddSale @employeeid INT,
                    @productid  INT,
                    @quantity   SMALLINT,
                    @saleid     UNIQUEIDENTIFIER OUTPUT
AS
SET @saleid = NEWID()
  BEGIN TRY
    BEGIN TRANSACTION
    INSERT INTO Sales.Sales
         SELECT
           @saleid,
           @productid,
           @employeeid,
           @quantity
    COMMIT TRANSACTION
  END TRY
  BEGIN CATCH
    INSERT INTO dbo.DB_Errors
    VALUES
  (SUSER_SNAME(),
   ERROR_NUMBER(),
   ERROR_STATE(),
   ERROR_SEVERITY(),
   ERROR_LINE(),
   ERROR_PROCEDURE(),
   ERROR_MESSAGE(),
   GETDATE());

-- Transaction uncommittable
    IF (XACT_STATE()) = -1
      ROLLBACK TRANSACTION

-- Transaction committable
    IF (XACT_STATE()) = 1
      COMMIT TRANSACTION
  END CATCH
GO
```

So, if everything executes successfully inside the Begin transaction, it will insert a record into Sales, and then it will commit it. But if something goes wrong before the commit takes place and it transfers control down to our Catch – the question is: How do we know if we commit or rollback the whole thing?

If the error isn't serious, and it is in the committable state, we can still commit the transaction. But if something went wrong and is in an uncommittable state, then we can roll back the transaction. This

0 – there are no pending transactions

The only catch here is to remember to actually do this inside the catch statement because you don't want to start transactions and then not commit or roll them back:

How, if we execute the same stored procedure providing e.g. invalid **EmployeeID** we'll get the same errors as before generated inside out table:

The way we can tell that this wasn't inserted is by executing a simple Select query, selecting everything from the **Sales** table where **EmployeeID** is **20**:

Generating custom raise error SQL message

Let's wrap things up by looking at how we can create our own custom error messages. These are good when we know that there's a possible situation that might occur. As we mentioned earlier, it's possible that someone will pass an invalid employee ID. In this particular case, we can do a check before then and sure enough, when this happens, we can raise our own custom message like saying employee ID does not exist. This can be easily done by altering our stored procedure one more time and adding the lookup in our TRY block:

```
ALTER PROCEDURE dbo.AddSale @employeeid INT,
                  @productid  INT,
                  @quantity   SMALLINT,
                  @saleid     UNIQUEIDENTIFIER OUTPUT
AS
SET @saleid = NEWID()
  BEGIN TRY
  IF (SELECT COUNT(*) FROM HumanResources.Employee e WHERE employeeid = @employ-
eeid) = 0
      RAISEERROR ('EmployeeID does not exist.', 11, 1)

    INSERT INTO Sales.Sales
         SELECT
           @saleid,
           @productid,
           @employeeid,
```

```
    DECLARE @Message varchar(MAX) = ERROR_MESSAGE(),
          @Severity int = ERROR_SEVERITY(),
          @State smallint = ERROR_STATE()

    RAISEERROR (@Message, @Severity, @State)
  END CATCH
GO
```

If this count comes back as zero, that means the employee with that ID doesn't exist. Then we can call the RAISERROR where we define a user-defined message, and furthermore our custom severity and state. So, that would be a lot easier for someone using this stored procedure to understand what the problem is rather than seeing the very technical error message that SQL throws, in this case, about the foreign key validation.

With the last changes in our store procedure, there also another RAISERROR in the Catch block. If another error occurred, rather than having it slip under, we can again call the RAISERROR and pass back exactly what happened. That's why we have declared all the variables and the results of all the functions. This way, it will not only get logged but also report back to the application or user.

And now if we execute the same code from before, it will both get logged and it will also indicate that the employee ID does not exist:

Another thing worth mentioning is that we can actually predefine this error message code, severity, and state. There is a stored procedure called sp_addmessage that is used to add our own error messages. This is useful when we need to call the message on multiple places; we can just use RAISERROR and pass the message number rather than retyping the stuff all over again. By executing the selected code from below, we then added this error into SQL Server:

This means that now rather than doing it the way we did previously, we can just call the RAISERROR and pass in the error number and here's what it looks like:

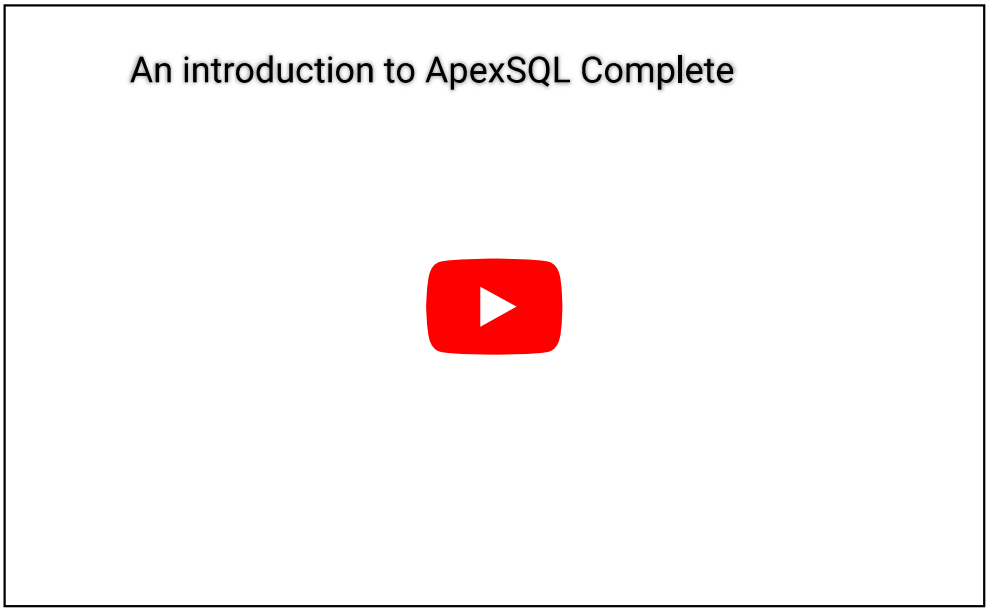
The sp_dropmessage is, of course, used to drop a specified user-defined error message. We can also view all the messages in SQL Server by executing the query from below:

References

- TRY...CATCH (Transact-SQL)
- RAISERROR (Transact-SQL)
- System Functions (Transact-SQL)

See more

To boost SQL coding productivity, check out these SQL tools for SSMS and Visual Studio including T-SQL formatting, refactoring, auto-complete, text and data search, snippets and auto-replacements, SQL code and object comparison, multi-db script comparison, object decryption and more



evangelism, and user engagement.

He has written extensively on both the SQL Shack and the ApexSQL Solution Center, on topics ranging from client technologies like 4K resolution and theming, error handling to index strategies, and performance monitoring.

Bojan works at ApexSQL in Nis, Serbia as an integral part of the team focusing on designing, developing, and testing the next generation of database tools including MySQL and SQL Server, and both stand-alone tools and integrations into Visual Studio, SSMS, and VSCode.

See more about Bojan at LinkedIn

View all posts by Bojan Petrovic

Related Posts:

1. **Static Data Masking in SSMS 18**
2. **SQL Server Replication with a table with more than 246 columns**
3. **Cómo poder implementar el manejo de errores en SQL Server**
4. **CASE statement in SQL**
5. **Convert SQL Server results into JSON**

Functions, T-SQL

643,190 Views

Brian Harrison • 2 years ago • edited

There are a couple of serious problems with this article.