

Keeping your cool when your stored procedures talk back

William Vaughn

Most of us execute stored procedures, but when they get in trouble, they sometimes complain, kick and cry. This article discusses how to handle those complaints without having to turn the car around and return home.

I've just endured one of the most harrowing experiences of my life—and I was a combat helicopter pilot in Vietnam. For reasons that now escape me, I drove my wife, daughter and her two kids (my grandchildren) back to Texas. We spent three days in the car listening to the three-year-old tell us what she wanted and if she didn't get it, she complained, kicked the seat and screamed at the top of her lungs. Years ago her mom behaved that way for a brief instant (when she was two), but never again as we nipped that behavior in the bud the instant it happened. I think I'm going to record the little darling and sell the tapes to the Army to use in Baghdad to encourage terrorist suspects to talk. But I digress. This article is not about three-year-old kids, but about stored procedures which can be just as troublesome.

Most of us have long since decided that stored procedures are the way to handle middle tier functionality. This is great until something goes wrong and they (like the three-year-old) begin to complain and cry. Based on the number of questions we got last night at the user group meeting (we had a great talk by Ron Talmage of Solid Quality Learning), I suspect there are those of you out there that have trouble getting your stored procedures to accurately report what's going on or when things aren't as they should be.

Stored procedures return a series of messages called resultsets. A stored procedure can return 1 to N resultsets. Each SELECT or action query generates its own resultset. Each resultset contains:

- 0 to 1 rowsets. If the stored procedure contains a SELECT it returns the rows in this structure that's extracted on the client end with a DataReader. Yes, even if you use a Fill to extract a rowset from the resultset, ADO.NET uses a DataReader to extract the rows.
- 0 to 1 "rows affected" value (a big integer). This is generated for action queries to indicate how many rows were inserted, deleted or updated by the action query. We often suppress this message by using SET NOCOUNT ON to help reduce the traffic when we're just executing action queries and choose to assume that the operations are all successful. This also simplifies executing stored procedures as it precludes the need to process resultsets individually.
- A RETURN value signed integer. By default, SQL Server sends back 0 if you don't use the RETURN statement to return some other integer. Usually, -1 indicates trouble—something went wrong in the procedure.
- 0 to N OUTPUT parameters. Another way to return information back from a stored procedure is to use (more efficient) OUTPUT parameters. Remember these are not available until the rowset is processed.
- 0 to N PRINT or RAISERROR message(s). While a stored procedure can use as many PRINT messages as you need (for status messages), you only get one RAISERROR.

The stored procedure can also report issues via the Application log file. This is accomplished through the RAISERROR statement using the WITH LOG option or through the xp_logevent function. Note that the application login account must be in the SysAdmins role to permit the WITH LOG option, but this is not a requirement for the xp_logevent function. I'm a real fan of message logging—but not always to the system logs. In the past I used my own log files, but using the system logs is okay if your DBA says so. This approach gives the developer or DBA a chance to see what's been going on—even if the application appears to be working fine. Logging also permits you to debug a problem at a distance—simply open the event log on the target server to look for messages posted by your stored procedures.

Previous articles have already discussed processing OUTPUT parameters so I won't discuss these here—just remember that they are managed through the Command Parameters collection when you set the Direction property appropriately, and OUTPUT parameters aren't available until you finish processing the rowset. They are especially efficient ways to pass back information from a stored procedure.

The RETURN statement can also be an easy way to indicate if a stored procedure has succeeded or failed. By convention, we usually set the RETURN value to -1 to indicate that the procedure failed for some reason and pass back 0 to indicate that the routine succeeded. Of course, you can set the RETURN value to any (signed)

integer you choose, so it can serve a variety of purposes—assuming you document what each return code means. Since it's common for stored procedures to call other stored procedures, the RETURN values are also used to pass back completion status from nested procedures. For example, the following TSQL executes the stored procedure "TestRaiseerror" and captures the RETURN value in a declared integer. It returns this value as a PRINT message to the client.

```
DECLARE @RetVal Int
EXEC @RetVal = TestRaiseerror 'TX', 10
PRINT CONVERT(Varchar,5,@RetVal)
```

As you know, a stored procedure is simply a TSQL program that executes on SQL Server. In many cases it contains more than just a SELECT—it also can execute logic to deal with the "extra stuff" that happens in the normal course of doing its job. Consider that not all of the things that happen are "errors"—many are simply situations that need to be dealt with. As a developer you have to (should) write code to deal with these situations either in the stored procedure, or in the code that calls the stored procedure.

Much of this extra code constitutes what many of us call "business rules". That is, it's written and tuned to address specific business situations. For example, if a stored procedure places an order for widgets and the operation reduces the "in-stock" count of widgets below a specified level, the stored procedure can automatically report this condition to the calling code which, in turn, can issue a reorder request—without the "human" getting involved. Clearly, this situation is not an "error", so it should not kill the process, application or stored procedure if it occurs. Unfortunately, earlier versions of the Microsoft data access interfaces (DAO, RDO and ADO) often failed when developers used a low-severity RAISERROR function to report these situations. ADO.NET works better in this respect as it understands that low-severity (less than 11) RAISERROR requests are not "fatal", but simply informational.

The RAISERROR function has been discussed before so I won't get into much detail here, but consider that it returns several pieces of information back to the calling program—or your code. These include:

- A formatted message string and/or number.
- A *severity* integer value (0-18).
- A *state* integer value (1-127).

These values can be captured by your program and used to determine what response should be made to the "issue" indicated. I'm not a fan of parsing strings to find out what happened so it's a good idea to come up with a set of numbers for your custom exceptions triggered by RAISERROR. You should "register" these messages with the server (it only has to be done once) so your entire team can use the same number and the associated message. You're permitted to create messages with numbers over 49,999. The following TSQL statement creates (replaces) message number 50010 with the severity set to 10, and a fixed message.

```
EXECUTE sp_addmessage 50010, 10, 'State not found',@replace='Replacement message'
```

Once the message is registered (this is typically done by the DBA), you can use RAISERROR to invoke it as shown in the following TSQL code. In this case, I invoke message number 50010 but pass in different *severity* and *state* values.

```
RAISERROR (50010, @Severity, 1)
```

Keep in mind that the RAISERROR function does *not* trip your Try/Catch logic unless the severity is 11 or higher. The message (and just the message) is passed to the InfoMessage event—assuming you coded it. Your stored procedure can set a severity anywhere from 1-25, but only members of the *sysadmin* role can invoke RAISERROR with severities of 19-25. Note that any severity 20-25 is considered fatal. In this case the client connection is terminated (closed) after receiving the message and the error is logged in the error and application logs. There is some evidence that closing the connection from the server end might also trash the connection in the pool, but that's fodder for another article.

Passing back "informational" messages

If you simply want to pass back an informational message using RAISERROR, you should use a Severity of 10 or less. In this case your code needs to implement the InfoMessage event to capture the message returned. This same event is used to process TSQL PRINT statement messages as well. These PRINT statements can be used to indicate status of a procedure, but remember that they won't necessarily be sent in real time—they might not appear at the client until the stored procedure is completed. Shown below is an example of a simple InfoMessage event implementation:

```

Private Sub cn_InfoMessage(ByVal sender As Object, ByVal e As _
System.Data.SqlClient.SqlInfoMessageEventArgs) Handles cn.InfoMessage
' Trapped on Sev 10 or less. Returns PRINT or RAISERROR messages
  lblMessages.Text &= "Infomessage from Cn:" & e.ToString _
    & vbCrLf & "Source:" & e.Source
End Sub

```

Of course, this requires you to first declare the Connection variable using the WithEvents construct as shown below:

```
Dim WithEvents cn As SqlConnection
```

The sample stored procedure

The sample application included with this article captures a few parameters and submits a query that executes a test stored procedure. The input parameters set the severity of the RAISERROR so you can see how ADO.NET behaves differently as the severity increases. Let's walk through the code--starting with the stored procedure. The stored procedure accepts two input parameters: one to set the state to use in the SELECT and another to let us choose the severity of the exception raised when the state code is invalid.

```

ALTER PROCEDURE dbo.TestRaiserror

(@StateWanted Char(2) = 'WA',
 @Severity tinyint = 0
)

AS
/*
RAISERROR ( { msg_id | msg_str } { , severity , state }
[ , argument [ ,...n ] ] )
[ WITH option [ ,...n ] ]
Create or replace current system message with this one...
*/

```

At this point I add a new message to handle the situation where no publishers use the state code provided.

```
EXECUTE sp_addmessage 50010, 10, 'State not found',@replace='Replacement message'
```

Next, I search the Publishers table for a match on the state code passed in. If it's not present, I use RAISERROR to signal this condition. I also illustrate use of the *xp_logevent* function to record the exception to the application log (on the SQL Server). I added a PRINT statement to illustrate how to pass back informational messages during the course of the procedure. While these seem useful, they slow down processing as they generate extra TDS packets and increase the volume of data moving over the wire—just use them judiciously.

```

DECLARE @Cnt AS INTEGER
SELECT @cnt = count(*) FROM publishers WHERE state = @StateWanted
PRINT 'Initial validation SELECT just executed--return informational message'
IF @cnt < 1
  BEGIN
    IF @Severity > 18
      BEGIN
        RAISERROR (50010, @Severity, 1) WITH LOG /* For SysAdmins */
        EXEC xp_logevent 50010,'Example message--Severity > 18','INFORMATIONAL'
      END
    ELSE
      RAISERROR (50010, @Severity, 1 )
      PRINT 'This is another way to pass back an error message string'
      RETURN -1 /* Indicate that there was a problem */
  END
ELSE
  BEGIN
    SELECT PubID, PubName, Company_Name, Address, City, State, Zip, Telephone, Fax,
Comments
    FROM Publishers
    WHERE (State = @StateWanted)
    PRINT 'Rowset SELECT just executed--return another informational message'
    RETURN @@Rowcount
  END

```

I also included a commented block of code in the SP to debug the stored procedure. Ron's talk featured this technique and I think it's a great idea to help develop the stored procedure and get it working without having to write ADO.NET code. Even Visual Studio .NET lets you select a few lines of TSQL code, right click and choose "Run Selection" to test the stored procedure—or any TSQL code. The results appear in the "Database Output" window.

```

/*
  Declare @RetVal Int
  EXEC @RetVal = TestRaiserror 'TX', 10
  Print convert(Varchar,5,@RetVal)
  GO
  Declare @RetVal Int
  EXEC @RetVal = TestRaiserror 'XX', 19
  Print @ convert(Varchar,5,@RetVal)
*/

```

The sample application

I wrote a simple application to illustrate the concepts. The initial form is shown in Figure 1.

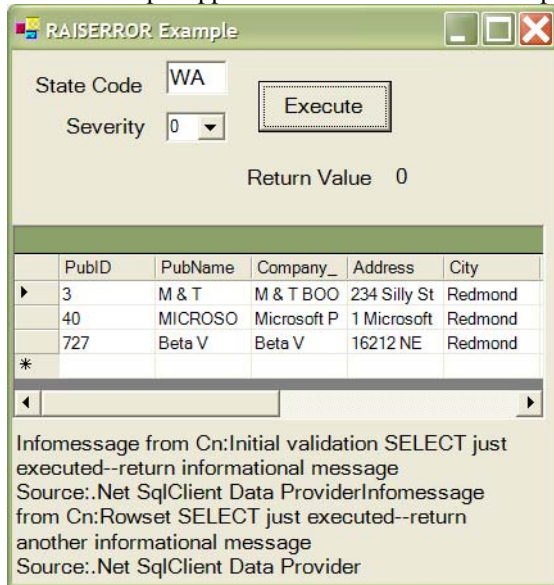


Figure 1: Test application after a successful query.

Notice the TextBox to enter a desired state code and the drop-down ComboBox to select a Severity code. The DataGrid displays the rowset returned by the query and the lower label control shows any InfoMessage text returned. The code behind the scenes is as follows:

```

Dim WithEvents cn As SqlConnection
Dim da As SqlDataAdapter
Dim ds As New DataSet ' To hold returned rowset (if any)

```

Setup the Command used to execute the stored procedure, pass in the input parameters and capture the RETURN value parameter.

```

Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
  cn = New SqlConnection("Data Source=demoserver;integrated ... ")
  da = New SqlDataAdapter("TestRaiserror", cn)
  With da.SelectCommand
    .CommandType = CommandType.StoredProcedure
    .Parameters.Add("@StateWanted", "WA")
    .Parameters.Add("@Severity", SqlDbType.TinyInt).Value = 0
    .Parameters.Add("@ReturnValue", _
      SqlDbType.Int).Direction = ParameterDirection.ReturnValue
  End With
End Sub

```

Execute the stored procedure after having set the input parameters based on the values set in the input form.

```

Private Sub btnExecute_click(ByVal sender As System.Object, ...
  lblMessages.Text = ""
  Try

```

```

da.SelectCommand.Parameters("@StateWanted").Value = txtStateWanted.Text
da.SelectCommand.Parameters("@Severity").Value = CInt(cbSeverity.Text)

ds.Clear()      ' To clear out any previous results
da.Fill(ds, "Publishers")

```

Capture the RETURN value after the stored procedure is executed and the rowset (if any) has been processed.

```
lblRetVal.Text = da.SelectCommand.Parameters("@ReturnValue").Value.ToString
```

If a rowset was not returned, we know that something went wrong. Otherwise, we bind to the rowset.

```

If ds.Tables.Count = 1 Then
    DataGrid1.DataSource = ds.Tables(0)
Else
    MsgBox("No states with that state code")
End If

```

Trap the SQLException exception returned if the severity is greater than 10. In this case, I call a routine to display the exception on a modal dialog so we can see all of the returned SQLException arguments.

```

Catch exSQL As SQLException
    ShowSQLException(exSQL)
Catch ex As Exception
    MsgBox(ex.ToString)
End Try

```

End Sub

This is the routine used to display the SQLException details.

```

Private Sub ShowSQLException(ByVal exSQL As SQLException)
    Dim frmsQLEx As New frmsQLException
    frmsQLEx.exSQL = exSQL
    frmsQLEx.ShowDialog()
End Sub

```

The dialog (shown in Figure 2) is displayed when a state code is not found and the Severity is set to 11 or higher. Note that the SQLException object is rich with information such as the name and line number of the stored procedure that failed. No, it did not return the server name for some reason.

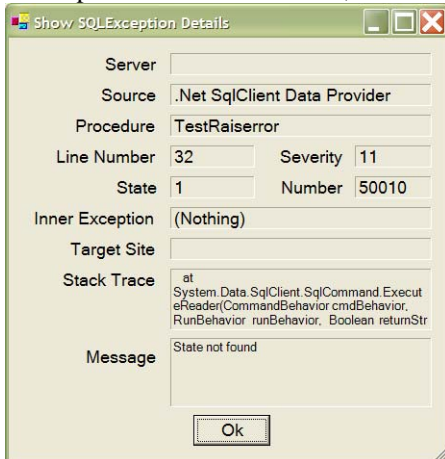


Figure 2: SQLException details dialog

The InfoMessage event traps any RAISERROR function messages (when the severity is less than 11) and any PRINT function messages. It simply displays these on the form as they arrive.

```

Private Sub cn_InfoMessage(ByVal sender As Object, ByVal e As
System.Data.SqlClient.SqlInfoMessageEventArgs) Handles cn.InfoMessage
    ' Trapped on Sev 10 or less. Returns PRINT or RAISERROR messages
    lblMessages.Text &= "Infomessage from Cn:" & e.ToString & vbCrLf
End Sub

```

Summary

Capturing the resultsets from SPs is only the first (and most elementary) step in working with stored procedures. Once you're able to handle a stored procedure that cries, kicks the back of your seat and puts silly putty in your hair while you drive, you're ready for anything. The trick is to handle these exceptions, conditions, issues (or whatever you call them) in the stored procedure whenever possible. This way you won't have to redeploy your application when the business rules change.

bio on file