THE LECTURE 7

SQL SERVER STORED PROCEDURES

STORED PROCEDURES

- Different options for creating SQL Server stored procedures
- Creating a simple stored procedure
- Create a SQL Server stored procedure with parameters
- Default Parameter Values
- Multiple Parameters
- Using TRY CATCH in SQL Server stored procedures
- Naming conventions for SQL Server stored procedures
- Reducing amount of network data for SQL Server stored procedures

STORED PROCEDURES

- In SQL Server, many administrative and informational activities can be performed by using system stored procedures.
- System stored procedures are prefixed by sp_, so it is not advisable to use sp_ for any of the stored procedures that we create, unless they form a part of our SQL Server installation.
- Stored procedures can be:
  - system / sp_ help …; sp_helptext …./
  - local
  - temporary
  - remote
  - extended

STORED PROCEDURES

- Stored procedures, user-defined functions, and prepared statements

- A stored procedure is a collection of SQL statements that can be called via a CALL statement.

- • A user-defined function is also a collection of SQL statements, but it can be called and used like any Built-in function

- • A prepared statement is a query that is stored on the server and that can be executed in the future

STORED PROCEDURES

- STORED PROCEDURES MUST BE DECLARED BEFORE THEY CAN BE CALLED.
- THE DECLARATION CAN INCLUDE PARAMETERS.
- IF PARAMETERS ARE CHANGED INSIDE THE PROCEDURE, THEIR MODIFIED VALUES ARE ACCESSIBLE AFTER THE CALL.

Creating a simple stored procedure

To create a stored procedure to do this the code would look like this:

```
CREATE PROCEDURE uspGetAddress
AS
SELECT * FROM
AdventureWorks.Person.Address
GO
```

To call the procedure to return the contents from the table specified, the code would be:

```
EXEC uspGetAddress

--or just simply

uspGetAddress
```

HOW TO CREATE A SQL SERVER STORED PROCEDURE WITH PARAMETERS

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT * FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

```
EXEC uspGetAddress @City = 'New York'
```

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT * FROM AdventureWorks.Person.Address
WHERE City LIKE @City + '%'
GO
```

DEFAULT PARAMETER VALUES

- In most cases it is always a good practice to pass in all parameter values, but sometimes it is not possible. So in this example we use the NULL option to allow you to not pass in a parameter value.
- If we create and run this stored procedure as is it will not return any data, because it is looking for any City values that equal NULL.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL
AS
SELECT *FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

We could change this procedure and use the ISNULL function to get around this.
So if a value is passed it will use the value to narrow the result set and if a value is not passed it will return all records.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL
AS
SELECT *FROM AdventureWorks.Person.Address
WHERE City = ISNULL(@City,City)
GO
```

MULTIPLE PARAMETERS

- Setting up multiple parameters is very easy to do. You just need to list each parameter and the data type separated by a comma as shown below.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30) = NULL,
@AddressLine1 nvarchar(60) = NULL
AS
SELECT *FROM AdventureWorks.Person.Address
WHERE City = ISNULL(@City,City)
AND AddressLine1 LIKE '%' + ISNULL(@AddressLine1 ,AddressLine1) + '%‘
GO
```

To execute this you could do any of the following:

```
EXEC uspGetAddress @City = 'Calgary‘
--or
EXEC uspGetAddress @City = 'Calgary', @AddressLine1 = 'A‘
--or
EXEC uspGetAddress @AddressLine1 = 'Acardia‘
-- etc...
```

RETURNING STORED PROCEDURE PARAMETER VALUES TO A CALLING STORED PROCEDURE

- **Overview**
  In a previous topic we discussed how to pass parameters into a stored procedure, but another option is to pass parameter values back out from a stored procedure.

  One option for this may be that you call another stored procedure that does not return any data, but returns parameter values to be used by the calling stored procedure.

- **Explanation**
  Setting up output paramters for a stored procedure is basically the same as setting up input parameters, the only difference is that you use the OUTPUT clause after the parameter name to specify that it should return a value.

  The output clause can be specified by either using the keyword "OUTPUT" or just "OUT".

SIMPLE OUTPUT

```
CREATE PROCEDURE uspGetAddressCount @City nvarchar(30),
@AddressCount int OUTPUT
AS
SELECT @AddressCount = count(*)
FROM AdventureWorks.Person.Address
WHERE City = @City
```

Or it can be done this way:

```
CREATE PROCEDURE uspGetAddressCount @City nvarchar(30),
@AddressCount int OUT
AS
SELECT @AddressCount = count(*)
FROM AdventureWorks.Person.Address
WHERE City = @City
```

To call this stored procedure we would execute it as follows. First we are going to declare a variable, execute the stored procedure and then select the returned valued.

```
DECLARE @AddressCount int
EXEC uspGetAddressCount @City = 'Calgary', @AddressCount =
@AddressCount
OUTPUT
SELECT @AddressCount
```

This can also be done as follows, where the stored procedure parameter names are not passed.

```
DECLARE @AddressCount int
EXEC uspGetAddressCount 'Calgary', @AddressCount OUTPUT
SELECT @AddressCount
```

MODIFYING AN EXISTING SQL SERVER STORED PROCEDURE

- **Overview**
  When you first create your stored procedures it may work as planned, but how to do you modify an existing stored procedure.
- In this topic we look at the ALTER PROCEDURE command and it is used.

- **Explanation**
  Modifying or ALTERing a stored procedure is pretty simple. Once a stored procedure has been created it is stored within one of the system tables in the database that is was created in.
- When you modify a stored procedure the entry that was originally made in the system table is replaced by this new code. Also, SQL Server will recompile the stored procedure the next time it is run, so your users are using the new logic.
- The command to modify an existing stored procedure is ALTER PROCEDURE or ALTER PROC.

MODIFYING AN EXISTING STORED PROCEDURE

- Let's say we have the following existing stored procedure: This allows us to do an exact match on the City.

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT * FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

Let's say we want to change this to do a LIKE instead of an equals.
To change the stored procedure and save the updated code you would use the ALTER PROCEDURE command as follows.

```
ALTER PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT * FROM AdventureWorks.Person.Address
WHERE City LIKE @City + '%‘
GO
```

Now the next time that the stored procedure is called by an end user it will use this new logic.

DELETING A SQL SERVER STORED PROCEDURE

- **Overview**
  In addition to creating stored procedures there is also the need to delete stored procedures. This topic shows you how you can delete stored procedures that are no longer needed.
- **Explanation**
  The syntax is very straightforward to drop a stored procedure, here are some examples.

Dropping Single Stored Procedure
To drop a single stored procedure you use the DROP PROCEDURE or DROP PROC command as follows.

```
DROP PROCEDURE uspGetAddress
GO
-- or
DROP PROC uspGetAddress
GO
-- or DROP PROC dbo.uspGetAddress -- also specify the schema
```

DROPPING MULTIPLE STORED PROCEDURES

- To drop multiple stored procedures with one command you specify each procedure separated by a comma as shown below.

```
DROP PROCEDURE uspGetAddress, uspInsertAddress, uspDeleteAddress
GO
-- or
DROP PROC uspGetAddress, uspInsertAddress, uspDeleteAddress
GO
```

USING TRY CATCH IN SQL SERVER STORED PROCEDURES

- **Overview**
  A great new option that was added in SQL Server 2005 was the ability to use the TRY..CATCH paradigm that exists in other development languages. Doing error handling in SQL Server has not always been the easiest thing, so this option definitely makes it much easier to code for and handle errors.
- **Explanation**
  If you are not familiar with the TRY...CATCH paradigm it is basically two blocks of code with your stored procedures that lets you execute some code, this is the Try section and if there are errors they are handled in the Catch section.

Using TRY CATCH in SQL Server stored procedures

- Let's take a look at an example of how this can be done. As you can see we are using a basic SELECT statement that is contained within the TRY section, but for some reason if this fails it will run the code in the CATCH section and return the error information.

```
CREATE PROCEDURE uspTryCatchTest
AS
BEGIN TRY
    SELECT 1/0
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber
    ,ERROR_SEVERITY() AS ErrorSeverity
    ,ERROR_STATE() AS ErrorState
    ,ERROR_PROCEDURE() AS ErrorProcedure
    ,ERROR_LINE() AS ErrorLine
    ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

REDUCING AMOUNT OF NETWORK DATA FOR SQL SERVER STORED PROCEDURES

- **Overview**
  There are many tricks that can be used when you write T-SQL code. One of these is to reduce the amount of network data for each statement that occurs within your stored procedures. Every time a SQL statement is executed it returns the number of rows that were affected. By using "SET NOCOUNT ON" within your stored procedure you can shut off these messages and reduce some of the traffic.

- **Explanation**
  As mentioned above there is not really any reason to return messages about what is occuring within SQL Server when you run a stored procedure. If you are running things from a query window, this may be useful, but most end users that run stored procedures through an application would never see these messages.

- You can still use @@ROWCOUNT to get the number of rows impacted by a SQL statement, so turning SET NOCOUNT ON will not change that behavior.
- **Not using SET NOCOUNT ON**
- Here is an example without using SET NOCOUNT ON:

```
-- not using SET NOCOUNT ON
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT * FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

The messages that are returned would be similar to this:

```
(23 row(s) affected)
```

USING SET NOCOUNT ON

- This example uses the SET NOCOUNT ON as shown below. It is a good practice to put this at the beginning of the stored procedure.

```
-- using SET NOCOUNT ON
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SET NOCOUNT ON
SELECT * FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

The messages that are returned would be similar to this:

```
Command(s) completed successfully.
```

USING SET NOCOUNT ON AND @@ROWCOUNT

- This example uses SET NOCOUNT ON, but will still return the number of rows impacted by the previous statement.
- This just shows that this still works.

```
-- not using SET NOCOUNT ON
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SET NOCOUNT ON
SELECT * FROM AdventureWorks.Person.Address
WHERE City = @CityPRINT @@ROWCOUNT
GO
```

The messages that are returned would be similar to this:

```
23
```

SET NOCOUNT OFF

If you wanted to turn this behavior off, you would just use the command "SET NOCOUNT OFF".