LAB:6

Stored Procedures &Inbuilt Functions

Overview

A stored procedure is nothing more than prepared SQL code that you save so you can reuse the code over and over again. So if you think about a query that you write over and over again, instead of having to write that query each time you would save it as a stored procedure and then just call the stored procedure to execute the SQL code that you saved as part of the stored procedure.

In addition to running the same SQL code over and over again you also have the ability to pass parameters to the stored procedure, so depending on what the need is the stored procedure can act accordingly based on the parameter values that were passed.

How to create a Procedure?

Syntax:

```
CREATE PROCEDURE procedure_name
AS
Sql_expression
GO
```

Example:

```
create procedure show1
as
select * from example
go
```

How to run the procedure?

Exec procedure_name

OR

Procedure_name

When creating a stored procedure you can either use CREATE PROCEDURE or CREATE PROC. After the stored procedure name you need to use the keyword "AS" and then the rest is just the regular SQL code that you would normally execute.

One thing to note is that you cannot use the keyword "GO" in the stored procedure. Once the SQL Server compiler sees "GO" it assumes it is the end of the batch.

Also, you cannot change database context within the stored procedure such as using "USE dbName" the reason for this is because this would be a separate batch and a stored procedure is a collection of only one batch of statements.

How to create a SQL Server stored procedure with parameters:

The real power of stored procedures is the ability to pass parameters and have the stored procedure handle the differing requests that are made. In this topic we will look at passing parameter values to a stored procedure.

Explanation

Just like you have the ability to use parameters with your SQL code you can also setup your stored procedures to accept one or more parameter values.

One Parameter

In this example we will query the Person.Address table from the AdventureWorks database, but instead of getting back all records we will limit it to just a particular city. This example assumes there will be an exact match on the City value that is passed.

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

To call this stored procedure we would execute it as follows:

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

We can also do the same thing, but allow the users to give us a starting point to search the data. Here we can change the "=" to a LIKE and use the "%" wildcard.

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

In both of the proceeding examples it assumes that a parameter value will always be passed. If you try to execute the procedure without passing a parameter value you will get an error message such as the following:

Msg 201, Level 16, State 4, Procedure uspGetAddress, Line 0

Procedure or function 'uspGetAddress' expects parameter '@City', which was not supplied.

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 stored 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. (Note: if the City column has NULL values this will not include these values. You will have to add additional logic for City IS NULL)

```
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...
```

Example:

```
CREATE PROCEDURE show4 @nam varchar(30),@idi int
AS
SELECT *
FROM example
WHERE name like @nam +'%' or id=@idi
GO
drop procedure show4
 show4 @nam ='s',@idi=12
```

Deleting a SQL Server stored procedure

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
```

Naming conventions for SQL Server stored Procedures

SQL Server uses object names and schema names to find a particular object that it needs to work with. This could be a table, stored procedure, function ,etc...

It is a good practice to come up with a standard naming convention for you objects including stored procedures.

Do not use sp_ as a prefix

One of the things you do not want to use as a standard is "sp_". This is a standard naming convention that is used in the master database. If you do not specify the database where the object is, SQL Server will first search the master database to see if the object exists there and then it will search the user database. So avoid using this as a naming convention.

Standardize on a Prefix

It is a good idea to come up with a standard prefix to use for your stored procedures. As mentioned above do not use "sp_", so here are some other options.

- usp_
- sp
- usp
- etc...

To be honest it does not really matter what you use. SQL Server will figure out that it is a stored procedure, but it is helpful to differentiate the objects, so it is easier to manage.

So a few examples could be:

- spInsertPerson

- uspInsertPerson
- usp_InsertPerson
- InsertPerson

Again this is totally up to you, but some standard is better than none.

Naming Stored Procedure Action

Based on the actions that you may take with a stored procedure, you may use:

- Insert
- Delete
- Update
- Select
- Get
- Validate
- etc...

So here are a few examples:

- uspInsertPerson
- uspGetPerson
- spValidatePerson
- SelectPerson
- etc...

Another option is to put the object name first and the action second, this way all of the stored procedures for an object will be together.

- uspPersonInsert
- uspPersonDelete
- uspPersonGet
- etc...

Again, this does not really matter what action words that you use, but this will be helpful to classify the behavior characteristics.

Naming Stored Procedure Object

Keep the names simple, but meaningful. As your database grows and you add more and more objects you will be glad that you created some standards.

So some of these may be:

- uspInsertPerson - insert a new person record
- uspGetAccountBalance - get the balance of an account
- uspGetOrderHistory - return list of orders

Using comments in a SQL Server stored procedure

SQL Server offers two types of comments in a stored procedure; line comments and block comments. The following examples show you how to add comments using both techniques. Comments are displayed in green in a SQL Server query window.

Line Comments

To create line comments you just use two dashes "--" in front of the code you want to comment. You can comment out one or multiple lines with this technique.

In this example the entire line is commented out.

```
-- this procedure gets a list of addresses based
-- on the city value that is passed
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

This next example shows you how to put the comment on the same line.

```
-- this procedure gets a list of addresses based on the city value that is passed
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City -- the @City parameter value will narrow the search criteria
GO
```

Block Comments

To create block comments the block is started with "/*" and ends with "*/". Anything within that block will be a comment section.

```
/*
-this procedure gets a list of addresses based
 on the city value that is passed
-this procedure is used by the HR system
*/
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City
GO
```

Combining Line and Block Comments

You can also use both types of comments within a stored procedure.

```
/*
-this procedure gets a list of addresses based
 on the city value that is passed
-this procedure is used by the HR system
*/
```

```
CREATE PROCEDURE uspGetAddress @City nvarchar(30)
AS
SELECT *
FROM AdventureWorks.Person.Address
WHERE City = @City -- the @City parameter value will narrow the search criteria
GO
```

Returning stored procedure parameter values to a calling stored procedure

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

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.

Sql date time functions:

- DateTime Function in SQL Server

  - GETDATE()
  - DATEPART()
  - DATEDIFF()
  - DATENAME()
  - DAY()
  - MONTH()
  - YEAR()

GETDATE() is very common used method which returns exact date time from the system. It does not accept any parameter. Just call it like simple function.

Example:

```
SELECT getdate()
```

DATEADD()

DATEADD() is used to add or subtract datetime. Its return a new datetime based on the added or subtracted interval.

```
General Syntax

DATEADD(datepart, number, date)
```

datepart is the parameter that specifies on which part of the date to return a new value. Number parameter is used to increment datepart.

Example:

```
SELECT getdate();

SELECT DATEADD(day, 5,getdate() ) AS NewTime
```

Output:

2015-08-12 09:18:18.080

2015-08-17 09:18:18.080

DATEPART()

DATEPART() is used when we need a part of date or time from a datetime variable. We can use DATEPART() method only with select command.

Syntax

```
DATEPART(datepart, date)

Example:
SELECT DATEPART(year, GETDATE()) AS 'Year'
SELECT DATEPART(hour, GETDATE()) AS 'Hour'
```

DATEDIFF()

DATEDIFF() is very common function to find out the difference between two DateTime elements.

Syntax

```
DATEDIFF(datepart, startdate, enddate)

Example:
SELECT DATEDIFF(day, @Date1, @Date2) AS DifferenceOfDay
```

DATENAME()

DATENAME() is very common and most useful function to find out the date name from the datetime value.

```
-- Get Today
SELECT DATENAME(dw, getdate()) AS 'Today Is'
-- Get Mont name
SELECT DATENAME(month, getdate()) AS 'Month'
```

DAY()

`DAY()` is used to get the day from any date time object.

Example:

```
SELECT DAY(getdate()) AS 'DAY'
```

Output :

```
DAY
-----------
12
```

MONTH()

```
SELECT MONTH(getdate()) AS 'Month'
```

YEAR()

```
SELECT YEAR(getdate()) AS 'Year'
```

String Functions

- ASCII()
- CHAR()
- LEFT()
- RIGHT()
- LTRIM()
- RTRIM()
- REPLACE()
- QUOTNAME()
- REVERSE()
- CHARINDEX()
- PATINDEX()
- LEN()
- STUFF()
- SUBSTRING()
- LOWER/UPPER()

ASCII()

Returns the ASCII code value of the leftmost character of a character expression.

Syntax

```
ASCII ( character_expression )
```

CHAR()

Converts an **int** ASCII code to a character.

Syntax
```
CHAR ( integer_expression )
```

Arguments: integer_expression: Is an integer from 0 through 255. NULL is returned if the integer expression is not in this range.

LEFT()

Returns the left most characters of a string.

Syntax
```
LEFT(string, length)
```

string
Specifies the string from which to obtain the left-most characters.

length
Specifies the number of characters to obtain.

```
Example:
SELECT LEFT('kathmandu',5)
 Output:
Kathm
```

RIGHT()

Returns the right most characters of a string.

Syntax
```
RIGHT(string, length)
```

string
Specifies the string from which to obtain the left-most characters.

length
Specifies the number of characters to obtain.

LTRIM()

Returns a character expression after it removes leading blanks.

Example :

```
SELECT LTRIM('   Md. Ashok')
```
Output :

Md. Ashok

RTRIM()

Returns a character string after truncating all trailing blanks.

Example :

```
SELECT RTRIM('Md. barsha    ')
```

Output:

Md. Barsha

REVERSE()

Returns a character expression in reverse order.

Example :

```
SELECT REVERSE('nepal')
```

CHARINDEX

CharIndex returns the first occurance of a string or characters within another string. The Format of CharIndex is given Below:

CHARINDEX (*expression1* , *expression2* [, *start_location*])

Here *expression1* is the string of characters to be found within *expression2*. So if you want to search *ij* within the word *Abhijit*, we will use *ij* as *expression1* and *Abhijit* as *expression2*. start_location is an optional integer argument which identifies the position from where the string will be searched. Now let us look into some examples :

Hide Copy Code

```
SELECT CHARINDEX('SQL', 'Microsoft SQL Server')
```

OUTPUT:

```
11
```

So it will start from 1 and go on searching until it finds the total string element searched, and returns its first position. The Result will be 0 if the searched string is not found.

We can also mention the Start_Location of the string to be searched.

EXAMPLE:

```
SELECT CHARINDEX('SQL', 'Microsoft SQL server has a great SQL Engine',12)
```

So in the above example we can have the Output as 34 as we specified the StartLocation as 12, which is greater than initial SQL position(11).

PATINDEX

As a contrast `PatIndex` is used to search a pattern within an expression. The Difference between `CharIndex` and `PatIndex` is the later allows WildCard Characters.

```
PATINDEX ('%pattern%' , expression)
```

Here the first argument takes a pattern with wildcard characters like `'%' (meaning any string) or '_' (meaning any character).`

For Example
PATINDEX('%BC%','ABCD')

Output:

```
2
```

LEN

Len is a function which returns the length of a string. This is the most common and simplest function that everyone use. Len Function excludes trailing blank spaces.

```
SELECT LEN('ABHISHEK IS WRITING THIS')
```

This will output 24, it is same when we write `LEN('ABHISHEK IS WRITING THIS ')` as LEN doesnt take trailing spaces in count.

SUBSTRING

`Substring` returns the part of the string from a given characterexpression. The general syntax of Substring is as follows :

```
SUBSTRING(expression, start, length)
```

Here the function gets the string from start to length. Let us take an example below:

```
SELECT OUT = SUBSTRING('abcdefgh', 2, 3)
```

The output will be "bcd".
Note : substring also works on ntext, VARCHAR, CHAR etc.

LOWER / UPPER

Another simple but handy function is Lower / UPPER. The will just change case of a string expression. For Example,

```
SELECT UPPER('this is Lower TEXT')
```