CLR Stored Procedures

Source	http://msdn2.microsoft.com/en-us/library/ms131094.aspx

CLR Stored Procedures

Stored procedures are routines that cannot be used in scalar expressions. Unlike scalar functions, they can return tabular results and messages to the client, invoke data definition language (DDL) and data manipulation language (DML) statements, and return output parameters. For information about the advantages of CLR integration and choosing between managed code and Transact-SQL, see Overview of CLR Integration.

Requirements for CLR Stored Procedures

In the common language runtime (CLR), stored procedures are implemented as public static methods on a class in a Microsoft .NET Framework assembly. The static method can either be declared as void, or return an integer value. If it returns an integer value, the integer returned is treated as the return code from the procedure. For example:

```
EXECUTE @return_status = procedure_name
```

The `@return_status` variable will contain the value returned by the method. If the method is declared void, the return code is 0.

If the method takes parameters, the number of parameters in the .NET Framework implementation should be the same as the number of parameters used in the Transact-SQL declaration of the stored procedure.

Parameters passed to a CLR stored procedure can be any of the native SQL Server types that have an equivalent in managed code. For the Transact-SQL syntax to create the procedure, these types should be specified with the most appropriate native SQL Server type equivalent. For more information about type conversions, see SQL Server Data Types and Their .NET Framework Equivalents.

Returning Results from CLR Stored Procedures

Information may be returned from .NET Framework stored procedures in several ways. This includes output parameters, tabular results, and messages.

OUTPUT Parameters and CLR Stored Procedures

As with Transact-SQL stored procedures, information may be returned from .NET Framework stored procedures using OUTPUT parameters. The Transact-SQL DML syntax used for creating .NET Framework stored procedures is the same as that used for creating stored procedures written in Transact-SQL. The corresponding parameter in the implementation code in the .NET Framework class should use a pass-by-reference parameter as the argument. Note that Visual Basic does not support output parameters in the same way that Visual C# does. You must specify the parameter by reference and apply the <Out()> attribute to represent an OUTPUT parameter, as in the following:

```
Imports System.Runtime.InteropServices
…
Public Shared Sub PriceSum ( <Out()> ByRef value As SqlInt32)
```

The following shows a stored procedure returning information through an OUTPUT parameter:

C#

```
using System;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public class StoredProcedures
{
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void PriceSum(out SqlInt32 value)
   {
      using(SqlConnection connection = new SqlConnection("context
connection=true"))
      {
         value = 0;
         connection.Open();
         SqlCommand command = new SqlCommand("SELECT Price FROM
Products", connection);
         SqlDataReader reader = command.ExecuteReader();
         using (reader)
         {
            while( reader.Read() )
            {
               value += reader.GetSqlInt32(0);
            }
         }
      }
   }
}
```

Visual Basic

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient
Imports System.Runtime.InteropServices

'The Partial modifier is only required on one class definition per
project.
Partial Public Class StoredProcedures
    ''' <summary>
    ''' Executes a query and iterates over the results to perform a
summation.
    ''' </summary>
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub PriceSum( <Out()> ByRef value As SqlInt32)

        Using connection As New SqlConnection("context
connection=true")
           value = 0
           Connection.Open()
           Dim command As New SqlCommand("SELECT Price FROM Products",
connection)
           Dim reader As SqlDataReader
           reader = command.ExecuteReader()
           Using reader
              While reader.Read()
                 value += reader.GetSqlInt32(0)
              End While
           End Using
        End Using
    End Sub
End Class
```

Once the assembly containing the above CLR stored procedure has been built and created on the server, the following Transact-SQL is used to create the procedure in the database, and specifies *sum* as an OUTPUT parameter.

```
CREATE PROCEDURE PriceSum (@sum int OUTPUT)
AS EXTERNAL NAME TestStoredProc.StoredProcedures.PriceSum
```

Note that *sum* is declared as an int SQL Server data type, and that the *value* parameter defined in the CLR stored procedure is specified as a SqlInt32 CLR data type. When a calling program executes the CLR stored procedure, SQL Server automatically converts the SqlInt32 CLR data type to an int SQL Server data type. For more information about which CLR data types can and cannot be converted, see SQL Server Data Types and Their .NET Framework Equivalents.

Returning Tabular Results and Messages

Returning tabular results and messages to the client is done through the SqlPipe object, which is obtained by using the Pipe property of the SqlContext class. The SqlPipe object has a Send method. By calling the Send method, you can transmit data through the pipe to the calling application.

These are several overloads of the SqlPipe.Send method, including one that sends a SqlDataReader and another that simply sends a text string.

Returning Messages

Use SqlPipe.Send(string) to send messages to the client application. The text of the message is limited to 8000 characters. If the message exceeds 8000 characters, it will be truncated.

Returning Tabular Results

To send the results of a query directly to the client, use one of the overloads of the Execute method on the SqlPipe object. This is the most efficient way to return results to the client, since the data is transferred to the network buffers without being copied into managed memory. For example:

[C#]

```
using System;
using System.Data;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public class StoredProcedures
{
   /// <summary>
   /// Execute a command and send the results to the client directly.
   /// </summary>
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void ExecuteToClient()
   {
   using(SqlConnection connection = new SqlConnection("context
connection=true"))
   {
      connection.Open();
      SqlCommand command = new SqlCommand("select @@version",
connection);
      SqlContext.Pipe.ExecuteAndSend(command);
      }
   }
}
```

[Visual Basic]

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

'The Partial modifier is only required on one class definition per
project.
Partial Public Class StoredProcedures
    ''' <summary>
    ''' Execute a command and send the results to the client directly.
    ''' </summary>
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub ExecuteToClient()
        Using connection As New SqlConnection("context
connection=true")
            connection.Open()
            Dim command As New SqlCommand("SELECT @@VERSION",
connection)
            SqlContext.Pipe.ExecuteAndSend(command)
        End Using
    End Sub
End Class
```

To send the results of a query that was executed previously through the in-process provider (or to pre-process the data using a custom implementation of SqlDataReader), use the overload of the Send method that takes a SqlDataReader. This method is slightly slower than the direct method described previously, but it offers greater flexibility to manipulate the data before it is sent to the client.

C#

```
using System;
using System.Data;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
public class StoredProcedures
{
   /// <summary>
   /// Execute a command and send the resulting reader to the client
   /// </summary>
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void SendReaderToClient()
   {
      using(SqlConnection connection = new SqlConnection("context
connection=true"))
      {
         connection.Open();
         SqlCommand command = new SqlCommand("select @@version",
connection);
         SqlDataReader r = command.ExecuteReader();
         SqlContext.Pipe.Send(r);
      }
   }
}
```

[Visual Basic]

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

'The Partial modifier is only required on one class definition per
project.
```

```
Partial Public Class StoredProcedures
    ''' <summary>
    ''' Execute a command and send the results to the client directly.
    ''' </summary>
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub SendReaderToClient()
        Using connection As New SqlConnection("context
connection=true")
            connection.Open()
            Dim command As New SqlCommand("SELECT @@VERSION",
connection)
            Dim reader As SqlDataReader
            reader = command.ExecuteReader()
            SqlContext.Pipe.Send(reader)
        End Using
    End Sub
End Class
```

To create a dynamic result set, populate it and send it to the client, you can create records from the current connection and send them using SqlPipe.Send.

C#

```
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;

public class StoredProcedures
{
   /// <summary>
   /// Create a result set on the fly and send it to the client.
   /// </summary>
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void SendTransientResultSet()
   {
      // Create a record object that represents an individual row,
including it's metadata.
      SqlDataRecord record = new SqlDataRecord(new
SqlMetaData("stringcol", SqlDbType.NVarChar, 128));

      // Populate the record.
      record.SetSqlString(0, "Hello World!");

      // Send the record to the client.
      SqlContext.Pipe.Send(record);
   }
}
```

[Visual Basic]

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

'The Partial modifier is only required on one class definition per
project.
Partial Public Class StoredProcedures
    ''' <summary>
    ''' Create a result set on the fly and send it to the client.
    ''' </summary>
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub SendTransientResultSet()
```

```
        ' Create a record object that represents an individual row,
including it's metadata.
        Dim record As New SqlDataRecord(New SqlMetaData("stringcol",
SqlDbType.NVarChar, 128) )

        ' Populate the record.
        record.SetSqlString(0, "Hello World!")

        ' Send the record to the client.
        SqlContext.Pipe.Send(record)
    End Sub
End Class
```

Here is an example of sending a tabular result and a message through SqlPipe.

C#

```
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public class StoredProcedures
{
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void HelloWorld()
   {
      SqlContext.Pipe.Send("Hello world! It's now " +
System.DateTime.Now.ToString()+"\n");
      using(SqlConnection connection = new SqlConnection("context
connection=true"))
      {
         connection.Open();
         SqlCommand command = new SqlCommand("SELECT ProductNumber
FROM ProductMaster", connection);
         SqlDataReader reader = command.ExecuteReader();
```

```
            SqlContext.Pipe.Send(reader);
        }
    }
}
```

[Visual Basic]

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

'The Partial modifier is only required on one class definition per
project.
Partial Public Class StoredProcedures
    ''' <summary>
    ''' Execute a command and send the results to the client directly.
    ''' </summary>
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub HelloWorld()
        SqlContext.Pipe.Send("Hello world! It's now " &
System.DateTime.Now.ToString() & "\n")
        Using connection As New SqlConnection("context
connection=true")
            connection.Open()
            Dim command As New SqlCommand("SELECT ProductNumber FROM
ProductMaster", connection)
            Dim reader As SqlDataReader
            reader = command.ExecuteReader()
            SqlContext.Pipe.Send(reader)
        End Using
    End Sub
End Class
```

The first Send sends a message to the client, while the second sends a tabular result using SqlDataReader.

Note that these examples are for illustrative purposes only. CLR functions are more appropriate than simple Transact-SQL statements for computation-intensive applications. An almost equivalent Transact-SQL stored procedure to the previous example is:

```
CREATE PROCEDURE HelloWorld() AS
BEGIN
PRINT('Hello world!')
SELECT ProductNumber FROM ProductMaster
END
```

Note:

Messages and result sets are retrieved differently in the client application. For instance, SQL Server Management Studio result sets appear in the Results view, and messages appear in the Messages pane.

If the above Visual C# code is saved in a file MyFirstUdp.cs and compiled with:

```
csc /t:library /out:MyFirstUdp.dll MyFirstUdp.cs
```

Or, if the above Visual Basic code is saved in a file MyFirstUdp.vb and compiled with:

```
vbc /t:library /out:MyFirstUdp.dll MyFirstUdp.vb
```

Note:

Managed C++ database objects, such as stored procedures, that have been compiled with the /clr:pure Visual C++ compiler option are not supported for execution in SQL Server 2005 RTM.

The resulting assembly can be registered, and the entry point invoked, with the following DDL:

```
CREATE ASSEMBLY MyFirstUdp FROM 'C:\Programming\MyFirstUdp.dll'
CREATE PROCEDURE HelloWorld
AS EXTERNAL NAME MyFirstUdp.StoredProcedures.HelloWorld
EXEC HelloWorld
```

Note:

On a SQL Server 2005 database with a compatibility level of "80," you cannot create managed user-defined types, stored procedures, functions, aggregates, or triggers. To take advantage of these CLR integration features of SQL Server 2005, you must use the sp_dbcmptlevel (Transact-SQL) stored procedure to set the database compatibility level to "90.".

***~~~ End of Article ~~~***