Microsoft SQL Server 9.0 Technical Articles

Managed Data Access Inside SQL Server with ADO.NET and SQLCLR

Pablo Castro
Microsoft Corporation

April 2005

Applies to:
- Microsoft SQL Server 2005
- Microsoft .NET Framework 2.0
- ADO.NET

Summary: Using the new SQLCLR feature, managed code can use ADO.NET when running inside SQL Server 2005. Learn about SQLCLR via basic scenarios of in-process data access, SQLCLR constructs, and their interactions. (26 printed pages)

Contents

Introduction
Part I: The Basics
- Why Do Data Access Inside SQLCLR?
- Getting Started with ADO.NET Inside SQLCLR
- The Context Connection
- Using ADO.NET in Different SQLCLR Objects
- When Not to Use SQLCLR + ADO.NET

Part II: Advanced Topics
- More on Connections
- Transactions

Conclusion
- Acknowledgements

Introduction

This white paper discusses how managed code can use ADO.NET when running inside SQL Server 2005 using the new SQLCLR feature.

In Part I, I describe the basic scenarios in which in-process data access might be required, and the difference between local and remote connections. Most of the different SQLCLR constructs are covered, such as stored procedures and functions, as well as interesting aspects of the interaction between the data access infrastructure and those constructs.

Part II further details in-process connections and restrictions that apply to ADO.NET when running inside SQLCLR. Finally, there is a detailed discussion on the transactions semantics of data access code inside SQLCLR, and how can it interact implicitly and explicitly with the transactions API.

Part I: The Basics

Why Do Data Access Inside SQLCLR?

SQL Server 2005 is highly integrated with the .NET Framework, enabling the creation of stored procedures, functions, user-defined types, and user-defined aggregates using your favorite .NET programming language. All of these constructs can take advantage of large portions of the .NET Framework infrastructure, the base class library, and third-party managed libraries.

In many cases the functionality of these pieces of managed code will be very computation-oriented. Things such as string parsing or scientific math are quite common in the applications that our early adopters are creating using SQLCLR.

However, you can do only so much using only number crunching and string manipulation algorithms in isolation. At some point you'll have to obtain your input or return results. If that information is relatively small and granular you can use input and output parameters or return values, but if you're handling a large volume of

information, then in-memory structures won't be an appropriate representation/transfer mechanism; a database might be a better fit in those scenarios. If you choose to store the information in a database, then SQLCLR and a data-access infrastructure are the tools you'll need.

There are a number of scenarios where you'll need database access when running inside SQLCLR. One is the scenario I just mentioned, where you have to perform some computation over a potentially large set of data. The other is integration across systems, where you need to talk to different servers to obtain an intermediate answer in order to proceed with a database-related operation.

> **Note** For an introduction and more details about SQLCLR in general, see Using CLR Integration in SQL Server 2005 [http://technet.microsoft.com/en-us/library/ms345136(printer).aspx] .

Now, if you're writing managed code and you want to do data access, what you need is ADO.NET.

Getting Started with ADO.NET Inside SQLCLR

The good news is that ADO.NET "just works" inside SQLCLR, so in order to get started you can leverage all your existing knowledge of ADO.NET.

To illustrate this take a look at the code snippet below. It would work fine in a client-side application, Web application, or a middle-tier component; it turns out that it will also work just fine inside SQLCLR.

Copy Code

C#

```
// Connection strings shouldn't be hardcoded for production code
using(SqlConnection conn = new SqlConnection(
  "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")) {
  conn.Open();

  SqlCommand cmd = new SqlCommand(
                    "SELECT Name, GroupName FROM HumanResources.Department", conn);

  SqlDataReader r = cmd.ExecuteReader();
  while(r.Read()) {
    // Consume the data from the reader and perform some computation with it
  }
}
```

Visual Basic .NET

```
Dim cmd as SqlCommand
Dim r as SqlDataReader

' Connection strings shouldn't be hardcoded for production code
Using conn As New SqlConnection( _
     "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")
  conn.Open()

  cmd = New SqlCommand("SELECT Name, GroupName FROM HumanResources.Department", conn)

  r = cmd.ExecuteReader()
  Do While r.Read()
    ' Consume the data from the reader and perform some computation with it
  Loop
End Using
```

This sample uses the **System.Data.SqlClient** provider to connect to SQL Server. Note that if this code runs inside SQLCLR, it would be connecting from the SQL Server that hosts it to another SQL Server. You can also connect to different data sources. For example, you can use the **System.Data.OracleClient** provider to connect to an Oracle server directly from inside SQL Server.

For the most part, there are no major differences using ADO.NET from within SQLCLR. However, there is one scenario that needs a little bit more attention: what if you want to connect to the same server your code is running in to retrieve/alter data? See The Context Connection section to see how ADO.NET addresses that.

Before delving into further detail, I'd like to go through the basic steps to run code inside SQLCLR. If you already have experience creating SQLCLR stored procedures, you'll probably want to skip this section.

Creating a managed stored procedure that uses ADO.NET from Visual Studio

Visual Studio 2005 includes great integration with SQL Server 2005 and makes it really easy to create and deploy SQLCLR projects. Let's create a new managed stored procedure that uses ADO.NET using Visual Studio 2005.

1. **Create a SQLCLR project.** The starting point for SQLCLR in Visual Studio is a database project. You need to create a new project by selecting the **database** project category under your favorite language. Next, select the project type called **SQL Server Project**, give it a name, and let Visual Studio create the project.
2. **Set permissions to EXTERNAL_ACCESS.** Go to the properties of the project (right-click on the project node, choose **Properties**), choose the **Database** tab, and then from the **Permission Level** combo-box, choose **External**.
3. **Add a stored-procedure to the project.** Once the project is created you can right-click on the project node and select **Add** -> **New Item**. On the pop-up dialog you'll see all the different SQLCLR objects that you can create. Select **Stored Procedure**, give it a name, and let Visual Studio create it. Visual Studio will create a template stored procedure for you.
4. **Customize the Visual Studio template.** Visual Studio will generate a template for a managed stored procedure for you. Since you want to use SqlClient (the .NET data access provider for SQL Server) to connect to another SQL Server in this sample, you'll need to add at least one more **using** (C#) or **imports** (Visual Basic) statement for System.Data.SqlClient.
5. **Code the stored procedure body.** Now you need the code for the stored procedure. Let's say you want to connect to another SQL Server (remember, your code will be running inside SQL Server), obtain some information based on input data, and process the results. Note that Visual Studio generated a **SqlProcedure** attribute for the stored procedure method; it is used by the Visual Studio deployment infrastructure, so leave it in place. If you take a look at the proceeding code you'll notice that there is nothing different from old fashioned ADO.NET code that would run in the client or middle-tier. We love that part :)

Copy Code

C#

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

public partial class StoredProcedures {
  [Microsoft.SqlServer.Server.SqlProcedure()]
  public static void SampleSP() {
    // as usual, connection strings shouldn't be hardcoded for production code
    using(SqlConnection conn = new SqlConnection(
                                    "server=MyServer; database=AdventureWorks; " +
                                    "user id=MyUser; password=MyPassword")) {
      conn.Open();

      SqlCommand cmd = new SqlCommand(
                    "SELECT Name, GroupName FROM HumanResources.Department", conn);

      SqlDataReader r = cmd.ExecuteReader();
      while(r.Read()) {
        // consume the data from the reader and perform some processing
      }
    }
  }
}
```

Visual Basic .NET

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

Partial Public Class StoredProcedures
  <Microsoft.SqlServer.Server.SqlProcedure()> _
  Public Shared Sub SampleSP()
    Dim cmd As SqlCommand
    Dim r As SqlDataReader

    ' as usual, connection strings shouldn't be hardcoded for production code
```

```
        Using conn As New SqlConnection( _
         "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")
          conn.Open()

          cmd = New SqlCommand( _
                      "SELECT Name, GroupName FROM HumanResources.Department", conn)

          r = cmd.ExecuteReader()
          Do While r.Read()
            ' consume the data from the reader and perform some processing
          Loop
        End Using
      End Sub
    End Class
```

6. **Deploy your assembly.** Now you need to deploy your stored procedure in SQL Server. Visual Studio makes it trivial to deploy the assembly to SQL Server and take the appropriate steps to register each of the objects in the assembly with the server. After building the project, on the **Build** menu, choose **Deploy Solution**. Visual Studio will connect to SQL Server, drop previous versions of the assembly if needed, send the new assembly to the server and register it, and then register the stored procedure that you added to the assembly.
7. **Try it out.** You can even customize the "test.sql" file that's generated under the "Test Scripts" project folder to exercise the stored procedure you're working on so Visual Studio will execute it when you press **Ctrl**+**F5**, or just press **F5**. (Yes, F5 will start the debugger, and you can debug code inside SQLCLR—both T-SQL and CLR code—isn't that cool?)

Creating a managed stored procedure that uses ADO.NET using the SDK only

If you don't have Visual Studio 2005 handy, or you'd like to see how things work the first time before letting Visual Studio do it for you, here is how to create a SQLCLR stored procedure by hand.

First, you need the code for the stored procedure. Let's say you want to do the same as in the Visual Studio example: connect to another SQL Server, obtain some information based on input data, and process the results.

Copy Code

C#

```
using System.Data;
using System.Data.SqlClient;

public class SP {

  public static void SampleSP() {
    // as usual, connection strings shouldn't be hardcoded for production code
    using(SqlConnection conn = new SqlConnection(
                                      "server=MyServer; database=AdventureWorks; " +
                                      "user id=MyUser; password=MyPassword")) {
      conn.Open();

      SqlCommand cmd = new SqlCommand(
                    "SELECT Name, GroupName FROM HumanResources.Department", conn);

      SqlDataReader r = cmd.ExecuteReader();
      while(r.Read()) {
        // consume the data from the reader and perform some processing
      }
    }
  }
}
```

Visual Basic .NET

```
Imports System.Data
Imports System.Data.SqlClient

Public Class SP
```

```
  Public Shared Sub SampleSP()
    Dim cmd As SqlCommand
    Dim r As SqlDataReader

    ' as usual, connection strings shouldn't be hardcoded for production code
    Using conn As New SqlConnection( _
     "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")
      conn.Open()

      cmd = New SqlCommand( _
                  "SELECT Name, GroupName FROM HumanResources.Department", conn)

      r = cmd.ExecuteReader()
      Do While r.Read()
        ' consume the data from the reader and perform some processing
      Loop
    End Using
  End Sub
End Class
```

Again, nothing different from old fashioned ADO.NET :)

Now you need to compile your code to produce a DLL assembly containing the stored procedure. The following command will do it (assuming that you called your file myprocs.cs/myprocs.vb and the .NET Framework 2.0 is in your path):

Copy Code

C#

```
csc /t:library myprocs.cs
```

VB

```
vbc /t:library myprocs.vb
```

This will compile your code and produce a new DLL called myprocs.dll. You need to register it with the server. Let's say you put myprocs.dll in c:\temp, here are the SQL statements required to install the stored procedure in SQL Server from that path. You can run this either from SQL Server Management Studio or from the **sqlcmd** command-line utility:

Copy Code

```
-- Register the assembly
CREATE ASSEMBLY myprocs FROM 'c:\temp\myprocs.dll'
WITH PERMISSION_SET=EXTERNAL_ACCESS
GO
-- Register the stored-procedure
CREATE PROCEDURE SampleSP AS EXTERNAL NAME myprocs.SP.SampleSP
```

The EXTERNAL_ACCESS permission set is required because the code is accessing an external resource, in this case another SQL Server. The default permission set (SAFE) does not allow external access.

If you make changes to your stored procedure later on, you can refresh the assembly in SQL Server without dropping and recreating everything, assuming that you didn't change the public interface (e.g., changed the type/number of parameters). In the scenario presented here, after recompiling the DLL, you can simply execute:

Copy Code

```
-- Refresh assembly from the file-system
ALTER ASSEMBLY myprocs FROM 'c:\temp\myprocs.dll'
```

The Context Connection

One data-access scenario that you can expect to be relatively common is that you'll want to access the same server where your CLR stored procedure or function is executing.

One option for that is to create a regular connection using SqlClient, specify a connection string that points to the local server, and open it.

Now you have a connection. However, this is a separate connection; this implies that you'll have to specify

credentials for logging in—it will be a different database session, it may have different SET options, it will be in a separate transaction, it won't see your temporary tables, etc.

In the end, if your stored procedure or function code is running inside SQLCLR, it is because someone connected to this SQL Server and executed some SQL statement to invoke it. You'll probably want that connection, along with its transaction, SET options, and so on. It turns out that you can get to it; it is called the *context connection*.

The context connection lets you execute SQL statements in the same context that your code was invoked in the first place. In order to obtain the context connection you simply need to use the new **context connection** connection string keyword, as in the example below:

Copy Code

C#

```
using(SqlConnection c = new SqlConnection("context connection=true")) {
    c.Open();
    // do something with the connection
}
```

Visual Basic .NET

```
Using c as new SqlConnection("context connection=true")
    c.Open()
    ' do something with the connection
End Using
```

In order to see whether your code is actually running in the same connection as the caller, you can do the following experiment: use a **SqlConnection** object and compare the SPID (the SQL Server session identifier) as seen from the caller and from within the connection. The code for the procedure looks like this:

Copy Code

C#

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

public partial class StoredProcedures {
    [Microsoft.SqlServer.Server.SqlProcedure()]
    public static void SampleSP(string connstring, out int spid) {
        using (SqlConnection conn = new SqlConnection(connstring)) {
            conn.Open();

            SqlCommand cmd = new SqlCommand("SELECT @@SPID", conn);
            spid = (int)cmd.ExecuteScalar();
        }
    }
}
```

Visual Basic .NET

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

Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub SampleSP(ByVal connstring As String, ByRef spid As Integer)
        Using conn As New SqlConnection(connstring)
            conn.Open()

            Dim cmd As New SqlCommand("SELECT @@SPID", conn)
            spid = CType(cmd.ExecuteScalar(), Integer)
        End Using
    End Sub
End Class
```

After compiling and deploying this in SQL Server, you can try it out:

Copy Code

```
-- Print the SPID as seen by this connection
PRINT @@SPID

-- Now call the stored proc and see what SPID we get for a regular connection
DECLARE @id INT
EXEC SampleSP 'server=.;user id=MyUser; password=MyPassword', @id OUTPUT
PRINT @id

-- Call the stored proc again, but now use the context connection
EXEC SampleSP 'context connection=true', @id OUTPUT
PRINT @id
```

You'll see that the first and last SPID will match, because they're effectively the same connection. The second SPID is different because a second connection (which is a completely new connection) to the server was established.

Using ADO.NET in Different SQLCLR Objects

The "context" object

As you'll see in the following sections that cover different SQLCLR objects, each one of them will execute in a given server "context." The context represents the environment where the SQLCLR code was activated, and allows code running inside SQLCLR to access appropriate run-time information based on what kind of SQLCLR object it is.

The top-level object that surfaces the context is the **SqlContext** class that's defined in the **Microsoft.SqlServer.Server** namespace.

Another object that's available most of the time is the **pipe** object, which represents the connection to the client. For example, in T-SQL you can use the PRINT statement to send a message back to the client (if the client is SqlClient, it will show up as a **SqlConnection.InfoMessage** event). You can do the same in SQLCLR by using the **SqlPipe** object:

Copy Code

```
C#
SqlContext.Pipe.Send("Hello, World! (from SQLCLR)");

Visual Basic .NET
SqlContext.Pipe.Send("Hello, World! (from SQLCLR)")
```

Stored procedures

All of the samples I used above were based on stored procedures. Stored procedures can be used to obtain and change data both on the local server and in remote data sources.

Stored procedures can also send results to the client, just like T-SQL stored procedures do. For example, in T-SQL you can have a stored procedure that does this:

Copy Code

```
CREATE PROCEDURE GetVendorsMinRating(@rating INT)
AS
SELECT VendorID, AccountNumber, Name FROM Purchasing.Vendor
WHERE CreditRating <= @rating
```

A client running this stored procedure will see result sets coming back to the client (i.e., you would use **ExecuteReader** and get a **SqlDataReader** back if you were using ADO.NET in the client as well).

Managed stored procedures can return result sets, too. For stored procedures that are dominated by set-oriented statements such as the example above, using T-SQL is always a better choice. However, if you have a stored procedure that does a lot of computation-intensive work or uses a managed library and then returns some results, it may make sense to use SQLCLR. Here is the same procedure rewritten in SQLCLR:

Copy Code

```
C#
```

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

public partial class StoredProcedures {
    [Microsoft.SqlServer.Server.SqlProcedure()]
    public static void SampleSP(int rating) {
        using (SqlConnection conn = new SqlConnection("context connection=true")) {
            conn.Open();

            SqlCommand cmd = new SqlCommand(
                "SELECT VendorID, AccountNumber, Name FROM Purchasing.Vendor " +
                "WHERE CreditRating <= @rating", conn);
            cmd.Parameters.AddWithValue("@rating", rating);

            // execute the command and send the results directly to the client
            SqlContext.Pipe.ExecuteAndSend(cmd);
        }
    }
}
```

Visual Basic .NET

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Transactions
Imports Microsoft.SqlServer.Server

Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub SampleSP(ByVal rating As Integer)
        Dim cmd As SqlCommand

        ' connect to the context connection
        Using conn As New SqlConnection("context connection=true")
            conn.Open()

            cmd = New SqlCommand( _
                "SELECT VendorID, AccountNumber, Name FROM Purchasing.Vendor " & _
                "WHERE CreditRating <= @rating", conn)
            cmd.Parameters.AddWithValue("@rating", rating)

            ' execute the command and send the results directly to the client
            SqlContext.Pipe.ExecuteAndSend(cmd)
        End Using
    End Sub
End Class
```

The example above shows how to send the results from a SQL query back to the client. However, it's very likely that you'll also have stored procedures that produce their own data (e.g., by performing some computation locally or by invoking a Web service) and you'll want to return that data to the client as a result set. That's also possible using SQLCLR. Here is a trivial example:

Copy Code

C#

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

public partial class StoredProcedures {
    [Microsoft.SqlServer.Server.SqlProcedure()]
    public static void SampleSP() {
        // simply produce a 10-row result-set with 2 columns, an int and a string
```

```
        // first, create the record and specify the metadata for the results
        SqlDataRecord rec = new SqlDataRecord(
            new SqlMetaData("col1", SqlDbType.NVarChar, 100),
            new SqlMetaData("col2", SqlDbType.Int));

        // start a new result-set
        SqlContext.Pipe.SendResultsStart(rec);

        // send rows
        for(int i = 0; i < 10; i++) {
            // set values for each column for this row
            // This data would presumably come from a more "interesting" computation
            rec.SetString(0, "row " + i.ToString());
            rec.SetInt32(1, i);
            SqlContext.Pipe.SendResultsRow(rec);
        }

        // complete the result-set
        SqlContext.Pipe.SendResultsEnd();
    }
}
```

Visual Basic .NET

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Transactions
Imports Microsoft.SqlServer.Server

Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub SampleSP()
        ' simply produce a 10-row result-set with 2 columns, an int and a string

        ' first, create the record and specify the metadata for the results
        Dim rec As New SqlDataRecord( _
            New SqlMetaData("col1", SqlDbType.NVarChar, 100), _
            New SqlMetaData("col2", SqlDbType.Int))

        ' start a new result-set
        SqlContext.Pipe.SendResultsStart(rec)

        ' send rows
        Dim i As Integer
        For i = 0 To 9
            ' set values for each column for this row
            ' This data would presumably come from a more "interesting" computation
            rec.SetString(0, "row " & i.ToString())
            rec.SetInt32(1, i)
            SqlContext.Pipe.SendResultsRow(rec)
        Next

        ' complete the result-set
        SqlContext.Pipe.SendResultsEnd()
    End Sub
End Class
```

User-defined functions

User-defined scalar functions already existed in previous versions of SQL Server. In SQL Server 2005 scalar functions can also be created using managed code, in addition to the already existing option of using T-SQL. In both cases the function is expected to return a single scalar value.

SQL Server assumes that functions do not cause side effects. That is, functions should not change the state of the database (no data or metadata changes). For T-SQL functions, this is actually enforced by the server, and a run-time error would be generated if a side-effecting operation (e.g. executing an UPDATE statement) were

attempted.

The same restrictions (no side effects) apply to managed functions. However, there is less enforcement of this restriction. If you use the context connection and try to execute a side-effecting T-SQL statement through it (e.g., an UPDATE statement) you'll get a **SqlException** from ADO.NET. However, we cannot detect it when you perform side-effecting operations through regular (non-context) connections. In general it is better play safe and not do side-effecting operations from functions unless you have a very clear understanding of the implications.

Also, functions cannot return result sets to the client as stored procedures do.

Table-valued user-defined functions

T-SQL table-valued functions (or TVFs) existed in previous versions of SQL Server. In SQL Server 2005 we support creating TVFs using managed code. We call table-valued functions created using managed code "streaming table-valued functions," or streaming TVFs for short.

- They are "table-valued" because they return a relation (a result set) instead of a scalar. That means that they can, for example, be used in the FROM part of a SELECT statement.
- They are "streaming" because after an initialization step, the server will call into your object to obtain rows, so you can produce them based on server demand, instead of having to create all the result in memory first and then return the whole thing to the database.

Here is a very simple example of a function that takes a single string with a comma-separated list of words and returns a single-column result-set with a row for each word.

Copy Code

C#

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

public partial class Functions {
    [Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName="FillRow")]
    // if you're using VS then add the following property setter to
    // the attribute above: TableDefinition="s NVARCHAR(4000)"
    public static IEnumerable ParseString(string str) {
        // Split() returns an array, which in turn
        // implements IEnumerable, so we're done :)
        return str.Split(',');
    }

    public static void FillRow(object row, out string str) {
        // "crack" the row into its parts. this case is trivial
        // because the row is only made of a single string
        str = (string)row;
    }
}
```

Visual Basic .NET

```
Imports System.Collections
Imports System.Data
Imports System.Data.SqlClient
Imports System.Transactions
Imports System.Runtime.InteropServices
Imports Microsoft.SqlServer.Server

Partial Public Class Functions
    <Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName:="FillRow")> _
    ' if you're using VS then add the following property setter to
    ' the attribute above: TableDefinition:="s NVARCHAR(4000)"
    Public Shared Function ParseString(ByVal str As String) As IEnumerable
        ' Split() returns an array, which in turn
        ' implements IEnumerable, so we're done :)
        Return Split(str, ",")
```

```
    End Function

    Public Shared Sub FillRow(ByVal row As Object, <Out()> ByRef str As String)
        ' "crack" the row into its parts. this case is trivial
        ' because the row is only made of a single string
        str = CType(row, String)
    End Sub
End Class
```

If you're using Visual Studio, simply deploy the assembly with the TVF. If you're doing this by hand, execute the following to register the TVF (assuming you already registered the assembly):

Copy Code

```
-- register the managed TVF
CREATE FUNCTION ParseString(@str NVARCHAR(4000))
RETURNS TABLE (s NVARCHAR(4000))
AS EXTERNAL NAME myprocs.Functions.ParseString
```

Once registered, you can give it a try by executing this T-SQL statement:

Copy Code

```
-- Use the TVF in a SELECT statement, throwing-in an "order by" just because
SELECT s FROM dbo.ParseString('a,b,c') ORDER BY s DESC
```

Now, what does this have to do with data access? Well, it turns out there are a couple of restrictions to keep in mind when using ADO.NET from a TVF:

- TVFs are still functions, so the side-effect restrictions also apply to them.
- You can use the context connection in the initialization method (e.g., **ParseString** in the example above), but not in the method that fills rows (the method pointed to by the **FillRowMethodName** attribute property).
- You can use ADO.NET with regular (non-context) connections in both initialization and fill-row methods. Note that performing queries or other long-running operations in the fill-row method can seriously impact the performance of the SELECT statement that uses the TVF.

Triggers

Creating triggers is in many aspects very similar to creating stored procedures. You can use ADO.NET to do data access from a trigger just like you would from a stored procedure.

For the triggers case, however, you'll typically have a couple of extra requirements:

- You'll want to "see" the changes that caused the trigger to fire. In a T-SQL trigger you'd typically do this by using the INSERTED and DELETED tables. For a managed trigger the same still applies: as long as you use the context connection, you can reference the INSERTED and DELETED tables from your SQL statements that you execute in the trigger using a SqlCommand object.
- You'll want to be able to tell which columns changed. You can use the **IsUpdatedColumn()** method of the **SqlTriggerContext** class to check whether a given column has changed. An instance of SqlTriggerContext is available off of the SqlContext class when the code is running inside a trigger; you can access it using the **SqlContext.TriggerContext** property.

Another common practice is to use a trigger to validate the input data, and if it doesn't pass the validation criteria, then abort the operation. You can also do this from managed code by simply using this statement:

Copy Code

C#

```
System.Transactions.Transaction.Current.Rollback();
```

Visual Basic .NET

```
System.Transactions.Transaction.Current.Rollback()
```

Wow, what happened there? It is simple thanks to the tight integration of SQLCLR with the .NET Framework. See the Part II: Advanced Topics section, Transactions, for more information.

When Not to Use SQLCLR + ADO.NET

Don't just wrap SQL

If you have a stored procedure that only executes a query, then it's always better to write it in T-SQL. Writing it in SQLCLR will take more development time (you have to write T-SQL code for the query and managed code for the procedure) and it will be slower at run-time.

Whenever you use SQLCLR to simply wrap a relatively straightforward piece of T-SQL code, you'll get worse performance and extra maintenance cost. SQLCLR is better when there is actual work other than set-oriented operations to be done in the stored procedure or function.

> **Note** The samples in this article are always parts of stored procedures or functions, and are never complete real-world components of production-level databases. I only include the minimum code necessary to exercise the ADO.NET API I am describing. That's why many of the samples contain only data-access code. In practice, if your stored procedure or function only has data-access code, then you should double-check and see if you could write it in T-SQL.

Avoid procedural row processing if set-oriented operations can do it

Set-oriented operations can be very powerful. Sometimes it can be tricky to get them right, but once they are there, the database engine has a lot of opportunities to understand what you want to do based on the SQL statement that you provide, and it can perform deep, sophisticated optimizations on your behalf.

So in general it's a good thing to process rows using set-oriented statements such as UPDATE, INSERT and DELETE.

Good examples of this are:

- Avoid row-by-row scans and updates. If at all possible, it's much better to try to write a more sophisticated UPDATE statement.
- Avoid custom aggregation of values by explicitly opening a SqlDataReader and iterating over the values. Either use the built-in aggregation functions (SUM, AVG, MIN, MAX, etc.) or create user-defined aggregates.

There are of course some scenarios where row-by-row processing using procedural logic makes sense. It's mostly a matter of making sure that you don't end up doing row-by-row processing for something that could be expressed in a single SQL statement.

Part II: Advanced Topics

More on Connections

Choosing between regular and context connections

If you're connecting to a remote server, you'll always be using regular connections. On the other hand, if you need to connect to the same server you're running a function or stored procedure on, in most cases you'll want to use the context connection. As I mentioned above, there are several reasons for this, such as running in the same transaction space, and not having to reauthenticate.

Additionally, using the context connection will typically result in better performance and less resource utilization. The context connection is an in-process–only connection, so it can talk to the server "directly", meaning that it doesn't need to go through the network protocol and transport layer to send SQL statements and receive results. It doesn't need to go through the authentication process either.

There are some cases where you may need to open a separate regular connection to the same server. For example, there are certain restrictions in using the context connection described in the Restrictions for the context connection section.

What do you mean by connect "directly" to the server?

I mentioned before that the context connection could connect "directly" to the server and bypass the network protocol and transport layers. Figure 1 represents the primary components of the SqlClient managed provider, as well as how the different components interact with each other when using a regular connection, and when using the context connection.

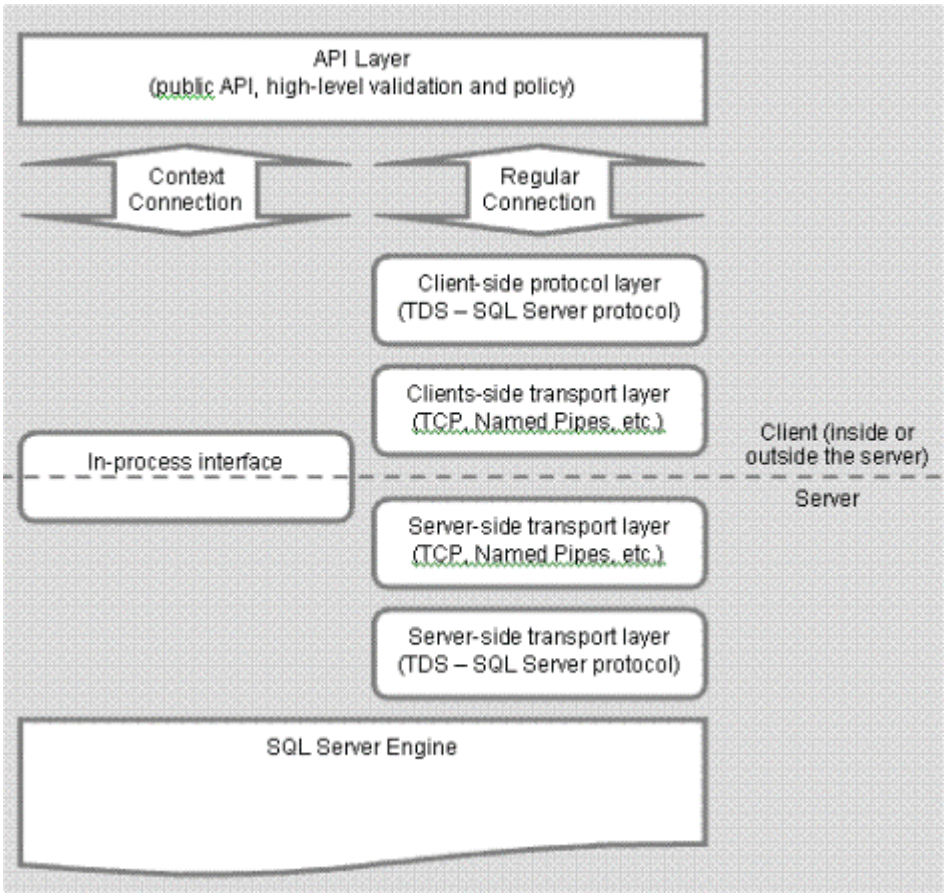

Figure 1. Connection processes

As you can see, the context connection follows a shorter code path and involves fewer components. Because of that, you can expect the context connection to get to the server and back faster than a regular connection. Query execution time will be the same, of course, because that work needs to be done regardless of how the SQL statement reaches the server.

Restrictions for the context connection

Here are the restrictions that apply to the context connection that you'll need to take into account when using it in your application:

- You can have only one context connection opened at a given time for a given connection.
    - Of course, if you have multiple statements running concurrently in separate connections, each one of them can get their own context connection. The restriction doesn't affect concurrent requests from different connections; it only affects a given request on a given connection.
- MARS (Multiple Active Result Sets) is not supported in the context connection.
- The **SqlBulkCopy** class will not operate on a context connection.
- We do not support update batching in the context connection.
- **SqlNotificationRequest** cannot be used with commands that will execute against the context connection.
- We don't support canceling commands that are running against the context connection. **SqlCommand.Cancel()** will silently ignore the request.
- No other connection string keywords can be used when you use **context connection=true**.

Some of these restrictions are by design and are the result of the semantics of the context connection. Others are actually implementation decisions that we made for this release and we may decide to relax those restrictions in a future release based on customer feedback.

Restrictions on regular connections inside SQLCLR

For those cases where you decide to use regular connections instead of the context connection, there are a few limitations to keep in mind.

Pretty much all the functionality of ADO.NET is available inside SQLCLR; however, there are a few specific features that either do not apply, or we decided not to support, in this release. Specifically, asynchronous command execution and the **SqlDependecy** object and related infrastructure are not supported.

Credentials for connections

You probably noticed that all the samples I've used so far use SQL authentication (user id and password) instead of integrated authentication. You may be wondering why I do that if we always strongly suggest using integrated authentication.

It turns out that it's not that straightforward to use inside SQLCLR. There are a couple of considerations that need to be kept in mind before using integrated authentication.

First of all, no client impersonation happens by default. This means that when SQL Server invokes your CLR code, it will be running under the account of the SQL Server service. If you use integrated authentication, the "identity" that your connections will have will be that of the service, not the one from the connecting client. In some scenarios this is actually intended and it will work fine. In many other scenarios this won't work. For example, if your SQL Server runs as "local system", then you won't be able to login to remote servers using integrated authentication.

> **Note** Skip these next two paragraphs if you want to avoid a headache :)

In some cases you may want to impersonate the caller by using the **SqlContext.WindowsIdentity** property instead of running as the service account. For those cases we expose a **WindowsIdentity** instance that represents the identity of the client that invoked the calling code. This is only available when the client used integrated authentication in the first place (otherwise we don't know the Windows identity of the client). Once you obtained the WindowsIdentity instance you can call **Impersonate** to change the security token of the thread, and then open ADO.NET connections on behalf of the client.

It gets more complicated. Even if you obtained the instance, by default you cannot propagate that instance to another computer; Windows security infrastructure restricts that by default. There is a mechanism called "delegation" that enables propagation of Windows identities across multiple trusted computers. You can learn more about delegation in the TechNet article, Kerberos Protocol Transition and Constrained Delegation [http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/constdel.mspx] .

Transactions

Let's say you have a managed stored procedure called SampleSP that has the following code:

Copy Code

```
C#
// as usual, connection strings shouldn't be hardcoded for production code
using(SqlConnection conn = new SqlConnection(
  "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")) {
  conn.Open();

  // insert a hardcoded row for this sample
  SqlCommand cmd = new SqlCommand("INSERT INTO HumanResources.Department " +
     "(Name, GroupName) VALUES ('Databases', 'IT'); SELECT SCOPE_IDENTITY()", conn);
  outputId = (int)cmd.ExecuteScalar();
}

Visual Basic .NET
Dim cmd as SqlCommand

' as usual, connection strings shouldn't be hardcoded for production code
Using conn As New SqlConnection( _
     "server=MyServer; database=AdventureWorks; user id=MyUser; password=MyPassword")
  conn.Open()

  ' insert a hardcoded row for this sample
  cmd = New SqlCommand("INSERT INTO HumanResources.Department " _ &
      "(Name, GroupName) VALUES ('Databases', 'IT'); SELECT SCOPE_IDENTITY()", conn)
  outputId = CType(cmd.ExecuteScalar(), Integer)
End Using
```

What happens if you do this in T-SQL?

Copy Code

```
BEGIN TRAN
```

```
DECLARE @id INT
-- create a new department and get its ID
EXEC SampleSP @id OUTPUT
-- move employees from department 1 to the new department
UPDATE Employees SET DepartmentID = @id WHERE DepartmentID = @id
-- now undo the entire operation
ROLLBACK
```

Since you did a BEGIN TRAN first, it's clear that the ROLLBACK statement will undo the work done by the UPDATE from T-SQL. But the stored procedure created a new ADO.NET connection to another server and made a change there, what about that change? Nothing to worry about—we'll detect that the code established ADO.NET connections to remote servers, and by default we'll transparently take any existing transaction with the connection and have all servers your code connects to participate in a distributed transaction. This even works for non-SQL Server connections!

How do we do this? We have GREAT integration with **System.Transactions**.

System.Transactions + ADO.NET + SQLCLR

System.Transactions is a new namespace that's part of the 2.0 release of the .NET Framework. It contains a new transactions framework that will greatly extend and simplify the use of local and distributed transactions in managed applications.

For an introduction to System.Transactions and ADO.NET, see the MSDN Magazine article, Data Points: ADO.NET and System.Transactions [http://msdn.microsoft.com/msdnmag/issues/05/02/DataPoints/default.aspx] , and the MSDN TV episode, Introducing System.Transactions in .NET Framework 2.0 [http://msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20050203NETMC/manifest.xml] .

ADO.NET and SQLCLR are tightly integrated with System.Transactions to provide a unified transactions API across the .NET Framework.

Transaction promotion

After reading about all the magic around distributed transactions for procedures, you may be thinking about the huge overhead this implies. It turns out that it's not bad at all.

When you invoke managed stored procedures within a database transaction, we flow the transaction context down into the CLR code.

As I mentioned before, the context connection is literally the same connection, so the same transaction applies and no extra overhead is involved.

On the other hand, if you're opening a connection to a remote server, that's clearly not the same connection. When you open an ADO.NET connection, we automatically detect that there is a database transaction that came with the context and "promote" the database transaction into a distributed transaction; then we enlist the connection to the remote server into that distributed transaction so everything is now coordinated. And this extra cost is only paid if you use it; otherwise it's only the cost of a regular database transaction. Cool stuff, huh?

> **Note** A similar transaction promotion feature is also available with ADO.NET and System.Transactions when used from the client and middle-tier scenarios. Consult the documentation on MSDN for further details.

Accessing the current transaction

At this point you may be wondering, "How do they know that there is a transaction active in the SQLCLR code to automatically enlist ADO.NET connections?" It turns out that integration goes deeper.

Outside of SQL Server, the System.Transactions framework exposes the concept of a "current transaction," which is available through **System.Transaction.Current**. We basically did the same thing inside the server.

If a transaction was active at the point where SQLCLR code is entered, then the transaction will be surfaced to the SQLCLR API through the **System.Transactions.Transaction** class. Specifically, Transaction.Current will be non-null.

In most cases you don't need to access the transaction explicitly. For database connections, ADO.NET will check Transaction.Current automatically during connection **Open()** and it will enlist the connection in that transaction transparently (unless you add **enlist=false** to the connection string).

There are a few scenarios where you might want to use the transaction object directly:

- If you want to abort the external transaction from within your stored procedure or function. In this case, you can simply call **Transaction.Current.Rollback()**.
- If you want to enlist a resource that doesn't do automatic enlistment, or for some reason wasn't enlisted during initialization.
- You may want to enlist yourself in the transaction, perhaps to be involved in the voting process or just to be notified when voting happens.

Note that although I used a very explicit example where I do a BEGIN TRAN, there other scenarios where your SQLCLR code can be invoked inside a transaction and Transaction.Current will be non-null. For example, if you invoke a managed user-defined function within an UPDATE statement, it will happen within a transaction even if one wasn't explicitly started.

Using System.Transactions explicitly

If you have a block of code that needs to execute within a transaction even if the caller didn't start one, you can use the System.Transactions API. This is, again, the same code you'd use in the client or middle-tier to manage a transaction. For example:

Copy Code

C#

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

public partial class StoredProcedures {
  [Microsoft.SqlServer.Server.SqlProcedure()]
  public static void SampleSP() {

    // start a transaction block
    using(TransactionScope tx = new TransactionScope()) {

      // connect to the context connection
      using(SqlConnection conn = new SqlConnection("context connection=true")) {
        conn.Open();

        // do some changes to the local database
      }

      // connect to the remote database
      using(SqlConnection conn = new SqlConnection(
                                 "server=MyServer; database=AdventureWorks;" +
                                 "user id=MyUser; password=MyPassword")) {
        conn.Open();

        // do some changes to the remote database
      }

      // mark the transaction as complete
      tx.Complete();
    }
  }
}
```

Visual Basic .NET

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Transactions
Imports Microsoft.SqlServer.Server

Partial Public Class StoredProcedures
  <Microsoft.SqlServer.Server.SqlProcedure()> _
  Public Shared Sub SampleSP()

    ' start a transaction block
```

```
    Using tx As New TransactionScope()

      ' connect to the context connection
      Using conn As New SqlConnection("context connection=true")
        conn.Open()

        ' do some changes to the local database
      End Using

      ' connect to a remote server (don't hardcode the conn string in real code)
      Using conn As New SqlConnection("server=MyServer; database=AdventureWorks;" & _
                                      "user id=MyUser; password=MyPassword")
        conn.Open()

        ' do some changes to the remote database
      End Using

      ' mark the transaction as completed
      tx.Complete()
    End Using
  End Sub
End Class
```

The sample above shows the simplest way of using System.Transactions. Simply put a transaction scope around the code that needs to be transacted. Note that towards the end of the block there is a call to the **Complete** method on the scope indicating that this piece of code executed its part successfully and it's OK with committing this transaction. If you want to abort the transaction, simply don't call Complete.

The **TransactionScope** object will do the "right thing" by default. That is, if there was already a transaction active, then the scope will happen within that transaction; otherwise, it will start a new transaction. There are other overloads that let you customize this behavior.

The pattern is fairly simple: the transaction scope will either pick up an already active transaction or will start a new one. In either case, since it's in a "using" block, the compiler will introduce a call to **Dispose** at the end of the block. If the scope saw a call to Complete before reaching the end of the block, then it will vote **commit** for the transaction; on the other hand, if it didn't see a call to Complete (e.g., an exception was thrown somewhere in the middle of the block), then it will rollback the transaction automatically.

> **Note** For the SQL Server 2005 release, the TransactionScope object will always use distributed transactions when running inside SQLCLR. This means that if there wasn't a distributed transaction already, the scope will cause the transaction to promote. This will cause overhead if you only connect to the local server; in that case, SQL transactions will be lighter weight. On the other hand, in scenarios where you use several resources (e.g., connections to remote databases), the transaction will have to be promoted anyway, so there is no additional overhead.
>
> I recommend not using TransactionScope if you're going to connect only using the context connection.

Using SQL transactions in your SQLCLR code

Alternatively, you can still use regular SQL transactions, although those will handle local transactions only.

Using the existing SQL transactions API is identical to how SQL transactions work in the client/middle-tier. You can either using SQL statements (e.g., BEGIN TRAN) or call the **BeginTransaction** method on the connection object. That returns a transaction object (e.g., **SqlTransaction**) that then you can use to commit/rollback the transaction.

These transactions can be nested, in the sense that your stored procedure or function might be called within a transaction, and it would still be perfectly legal for you to call BeginTransaction. (Note that this does not mean you get "true" nested transactions; you'll get the exact same behavior that you'd get when nesting BEGIN TRAN statements in T-SQL.)

Transaction lifetime

There is a difference between transactions started in T-SQL stored procedures and the ones started in SQLCLR code (using any of the methods discussed above): SQLCLR code cannot unbalance the transaction state on entry/exit of a SQLCLR invocation. This has a couple of implications:

- You cannot start a transaction inside a SQLCLR frame and not commit it or roll it back; SQL Server will generate an error during frame exit.
- Similarly, you cannot commit or rollback an outer transaction inside SQLCLR code.
    - Any attempt to commit a transaction that you didn't start in the same procedure will cause a run-time error.
    - Any attempt to rollback a transaction that you didn't start in the same procedure will doom the transaction (preventing any other side-effecting operation from happening), but the transaction won't disappear until the SQLCLR code unwinds. Note that this case is actually legal, and it's useful when you detect an error inside your procedure and want to make sure the whole transaction is aborted.

Conclusion

SQLCLR is a great technology and it will enable lots of new scenarios. Using ADO.NET inside SQLCLR is a powerful mix that will allow you to combine heavy processing with data access to both local and remote servers, all while maintaining transactional correctness.

As with any other technology, this one has a specific application domain. Not every procedure needs to be rewritten in SQLCLR and use ADO.NET to access the database; quite the contrary, in most cases T-SQL will do a great job. However, for those cases where sophisticated logic or rich libraries are required inside SQL Server, SQLCLR and ADO.NET are there to do the job.

Acknowledgements

Thanks to Acey Bunch, Alazel Acheson, Alyssa Henry, Angel Saenz-Badillos, Chris Lee, Jian Zeng, Mary Chipman and Steve Starck for taking the time to review this document and provide helpful feedback.