ISBN: 0321382188

Chapter 4
In-Process Data Access

Whether you access data from the client, middle-tier, or server, when you're using SQL Server and .NET you use the `SqlClient` data provider. Your data access code is similar regardless of its location, but the .NET 2.0 version of `SqlClient` contains code to encapsulate differences when you're programming inside SQL Server and optimize the in-server programming environment.

Draft Chapter

Programming With SqlClient

Accessing data from outside SQL Server entails connecting to SQL Server through a network library and a client library. When you use .NET with SQL Server, you use System.Data.dll as the client library and the ADO.NET programming model. ADO.NET is a provider-based model, similar in concept to ODBC, OLE DB, and JDBC. The model uses a common API (or a set of classes) to encapsulate data access; each database product has its own provider. ADO.NET providers are known as data providers and the data provider for SQL Server is `SqlClient`. The latest release of `SqlClient`, installed with .NET 2.0 includes new client-side functionality to take advantage of new features in SQL Server 2005. In addition, `SqlClient` contains extensions to allow ADO.NET code to be used inside the database itself. Though T-SQL is usually preferred when a stored procedure, user-defined function, or trigger accesses database data, you can also use ADO.NET when writing procedural code in .NET. The programming

ISBN: 0321382188

model when using `SqlClient` in .NET stored procedures is similar to client-side code, but in-database access is optimized because no network libraries are needed. Let's start by writing some simple client database code then convert it to run on the server.

Simple data access code is very similar regardless of the programming model used. To summarize, using ADO.NET and `SqlClient` as an example:

1. Connect to database by instantiating a `SqlConnection` class and calling its `Open` method.
2. Create an instance of a `SqlCommand` class. This instance contains a SQL statement or procedure name as its `CommandText property`. The `SqlCommand` is associated with the `SqlConnection`.
3. Execute the `SqlCommand` and return either a set of columns and rows called `SqlDataReader` or possibly only a count of rows affected by the statement
4. Use the `SqlDataReader` to read the results and Close it when finished.
5. Dispose of the `SqlCommand` and `SqlConnection` to free the associated memory, network, and server resources.

Draft Chapter

The ADO.NET code to accomplish inserting a row into a SQL Server table would look like this.

```
// Code to insert data from client
//   See chapter 14 for an implementation of
//   the GetConnectionStringFromConfigFile method.
string connStr = GetConnectionStringFromConfigFile();
SqlConnection conn = new SqlConnection(connStr);
conn.Open();
SqlCommand cmd = conn.CreateCommand();
```

```
cmd.CommandText = "insert into test values ('testdata')";
int rows_affected = cmd.ExecuteNonQuery();
cmd.Dispose();
conn.Dispose();
```

Listing 4-1

Inserting a row using SqlClient from the client

The previous ADO.NET code ignored the fact that an exception might cause the execution of `cmd.Dispose` or `conn.Dispose` to be skipped. The preferred and simple way to prevent this from happening is to use the "using" syntax in C#. One or object instance declarations are followed by a block of code. The `Dispose` method is called automatically at the end of the code block. We'll be using the "using" construct a lot in the code in this book. Rewritten using this syntax, the code above would look like this:

```
//code to insert data from client
string connStr = GetConnectionStringFromConfigFile();
using (SqlConnection conn = new SqlConnection(connStr))
using (SqlCommand cmd =
  new SqlCommand("insert into test values ('testdata')", conn))
{
  conn.Open();
  int rows_affected = cmd.ExecuteNonQuery();
}  // Dispose called on cmd and conn here
```

Listing 4-2

Inserting a row using SqlClient from the client, C# using construct

Draft Chapter

Other classes in a typical ADO.NET data provider include a transaction class (`SqlTransaction`) to tie Connections and Commands to a database transaction, a parameter collection (`SqlParameterCollection`) of parameters (`SqlParameter`) to use with parameterized SQL queries or stored procedures, and specialized Exception and Error classes (`SqlException, SqlErrorCollection, SqlError`) to represent processing errors. `SqlClient` includes all of the typical classes; Figure 4-1 shows the object model.

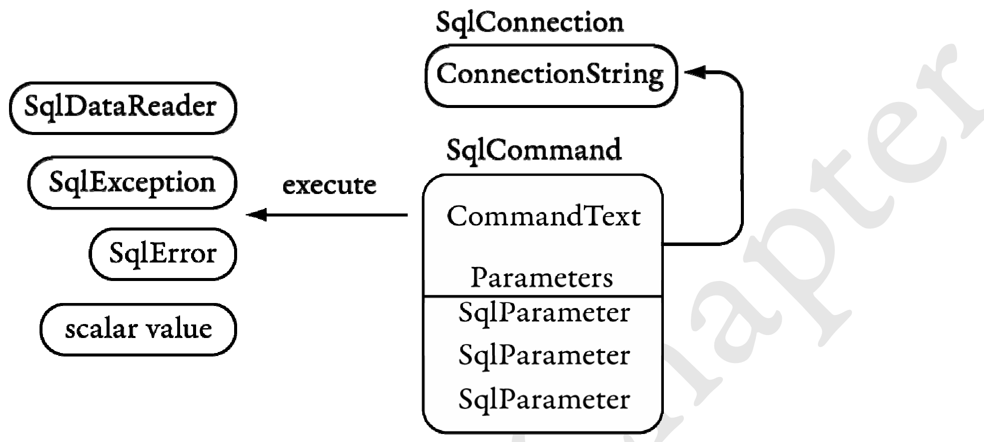

Figure 4-1

The SqlClient Provider Object Model (Major classes only)

Draft Chapter

The same basic model is used inside the server to access data in .NET stored procedures. It's familiar to ADO.NET programmers and using it inside the server makes it easy for programmers to use their existing skills to write procedures. The big difference is that, when you're writing a .NET procedure, you're already inside the database. No explicit connection is needed. Although there is no network connection to the database, there is a `SqlConnection` instance. The difference is in the connection string. Outside the database, the connection string should be read from a configuration file and contains items like the SQL Server instance to connect to (server keyword), the SQL Server login (either user id and password keyword or "`integrated security=true`"), and the initial database (database keyword). The connection string that indicates to `SqlClient` that we're already inside the database and the provider should just use

the existing database context contains only the keyword "context connection=true". When you specify "context connection=true", no other connection string keyword can be used. Here's the same code as above, but executing inside a .NET stored procedure.

```
//code to insert data in a stored procedure
public static void InsertRowOfTestData()
{
  string connStr = "context connection=true";
  using (SqlConnection conn = new SqlConnection(connStr))
  using (SqlCommand cmd =
    new SqlCommand("insert into test values ('testdata')", conn))
  {
    conn.Open();
    int rows_affected = cmd.ExecuteNonQuery();
  }
}
```

Listing 4-3

Inserting a row using SqlClient in a SQLCLR stored procedure

Draft Chapter

Note that this code is provided as a stored procedure only to explain how to access data on the server. The code is not only faster as a T-SQL stored procedure, but SQL Server will check the SQL statements for syntactic correctness at CREATE PROCEDURE time. This is not the case with the .NET stored procedure above. When you execute SQL statements by using SqlCommand, it's the equivalent of using sp_executesql (a system-supplied store procedure for dynamic string execution of commands) inside of a T-SQL stored procedure. There is the same potential for SQL injection as with sp_executesql, so don't execute commands whose CommandText property is calculated by using input parameters passed in by the procedure user.

ISBN: 0321382188

This code is so similar to the previous client-side code that, if we knew whether the code was executing in a stored procedure on the server or on the client, we could use the same code, changing only the connection string. But there are a few constructs that only exist if you are writing server-side code. Enter the `SqlContext` class.

Context—The SqlContext Class

The `SqlContext` class is one of the new classes that is only available if you're running inside the server. When a procedure or function is executed, it is executed as part of the user's connection. Whether that user connection comes from ODBC, ADO.NET, or T-SQL doesn't really matter. You are in a connection, that has specific properties, environment variables, and so on, and you are executing within that connection; you are in the context of the user's connection.

A command is executed within the context of the connection, but it also has an execution context, which consists of data related to the command. The same goes for triggers, which are executed within a trigger context.

Prior to SQL Server 2005, the closest we came to being able to write code in another language that executed within the process space of SQL Server was to write extended stored procedures. An extended stored procedure is a C or C++ DLL that has been cataloged in SQL Server and therefore can be executed in the same manner as a "normal" SQL Server stored procedure. The extended stored procedure is executed in process with SQL Server and on the same Windows thread[1] as the user's connection.

Note, however, that if you need to do any kind of database access, even within the database to which the user is connected, from the extended stored procedure you still need to connect to the

[1] Strictly speaking thread or fiber, depending on the setting in the server. See chapter 2 for information about fiber mode.

Draft Chapter

database explicitly either through ODBC, OLE DB, or even DBLib, exactly as you would do from a client, as Figure 4-2 illustrates. Furthermore, once you have created the connection from the procedure, you may want to share a common transaction lock space with the client. Because you now have a separate connection, you need to explicitly ensure that you share the transaction lock space by using the `srv_getbindtoken` call and the stored procedure `sp_bindsession`.

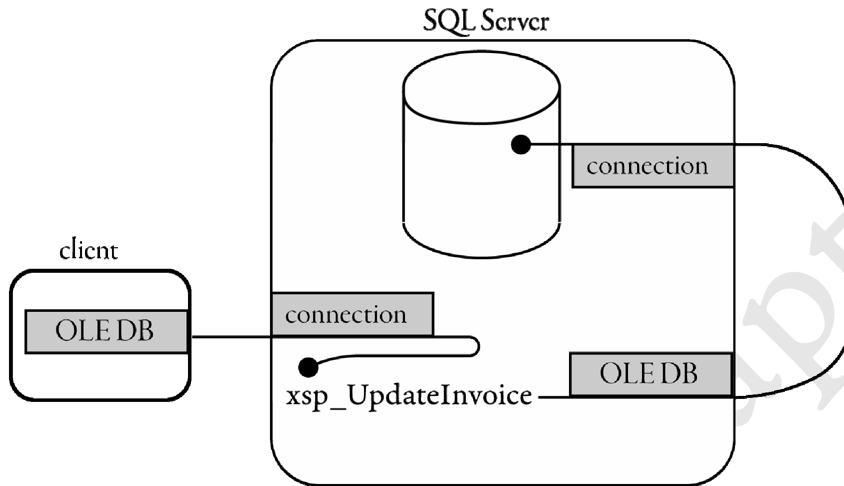

Figure 4-2

Connections from Extended Stored Procedures

In SQL Server 2005 when you use .NET to write procedures, functions, and triggers, the `SqlContext` is available. The original program can now be rewritten so that the same code works either or the client/middle-tier or in the server if it's called as part of a stored procedure using the `SqlContext` static `IsAvailable` property.

Draft Chapter

```
// other using statements elided for clarity
using System.Data.SqlClient;
using Microsoft.SqlServer.Server; // for SqlContext

public static void InsertRowOfTestData2()
{
```

ISBN: 0321382188

```
  string connStr;
  if (SqlContext.IsAvailable)
    connStr = "context connection=true";
  else
    connStr = GetConnectionStringFromConfigFile();
  // the rest of the code is identical
  using (SqlConnection conn = new SqlConnection(connStr))
  using (SqlCommand cmd =
    new SqlCommand("insert into test values ('testdata')", conn))
  {
    conn.Open();
    // The value of i is the number of rows affected
    int i = cmd.ExecuteNonQuery();
  }
}
```

Listing 4-4

Using IsAvailable to determine if the code is running on the server

Draft Chapter

You can see the `SqlContext` as a helper class; there are static read-only properties that allow you the access the class that encapsulate functionality that exists only on the server. These properties are shown in Table 4-1.

Table 4-1

SqlContext Static Properties

Property	Return Value
`IsAvailable`	`Boolean`
`WindowsIdentity`	`System.Security.Principal.WindowsIdentity`
`Pipe`	`Microsoft.SqlServer.Server.SqlPipe`

TriggerContext	Microsoft.SqlServer.Server.SqlTriggerContext

Using `SqlContext` is the only way to get an instance of the classes in Table 4-1—you cannot create them by using a constructor (`New` in Visual Basic.NET). You can create the other classes that are part of the `SqlClient` provider in the same way that you normally would create them if used from and ADO.NET client. Some of the classes and methods in `SqlClient` act a little differently if you use them on the server, however.

SQLCLR stored procedures can do data access by default, but this is not the case with a SQLCLR user-defined function. As was discussed in the previous chapter, unless `DataAccessKind` or `SystemDataAccessKind` is set to `DataAccessKind.Read/SystemDataAccessKind.Read`, any attempt to do data access using the `SqlClient` provider will fail. However, even if `DataAccessKind` is set to `DataAccessKind.None` (the default), `SqlContext.IsAvailable` returns true. You can still use the property as shown later in the chapter, so the `SqlContext's WindowsIdentity` must be available. `SqlContext.IsAvailable` is an indication of whether or not you're running in the server, rather than whether data access is permitted.

By now you may be wondering: If some of the managed classes are calling into SQL Server, does that mean that the internals of SQL Server are managed as well, and if not, are interoperability calls between managed and native code space happening? The answers are no and yes: No, the internals of SQL Server are not managed—Microsoft did not rewrite the whole of SQL Server in managed code. And yes, interoperability calls happen. The managed classes are making PInvoke calls against the executable of SQL Server, `sqlservr.exe`, as shown in Figure 4-3, which exposes a couple of dozen methods for the CLR to call into.

Draft Chapter

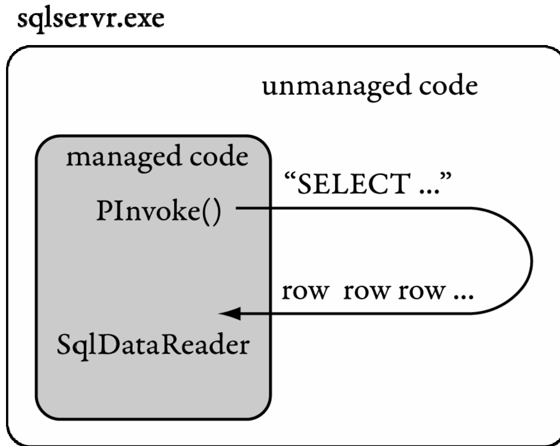

Figure 4-3

Interop between .NET and SQL Server code in-process

When you read this about interop, you may become concerned about performance. Theoretically, a performance hit is possible, but because SQL Server hosts the CLR (as discussed in Chapter 2) and the `SqlClient` provider runs in process with SQL Server, the hit is minimal. In the last sentence, notice that we said *theoretically:* Remember that when you execute CLR code, you will run machine-compiled code, which is not the case when you run T-SQL. Therefore, for *some* code executing in the CLR, the end result may be a performance improvement, compared with pure T-SQL code.

Now, that we have discussed the `SqlContext` class, let's see how we go about using it.

Draft Chapter

Connections

As already mentioned, when you are at server side and a client executes, you are part of that client's connection context, which in SQL Server 2005 is exposed by using a special connection string. `SqlConnection` object exposes the public methods, properties, and events listed in Table 4-2. (Note that the table doesn't show members inherited from `System.Object`.)

ISBN: 0321382188

Table 4-2

Public Members of SqlConnection

Name	Return Value/Type	Member Type
Constructor		Constructor
Constructor(String)		Constructor
BeginTransaction()	SqlTransaction	Method
BeginTransaction(IsolationLevel)	SqlTransaction	Method
BeginTransaction(IsolationLevel, String)	SqlTransaction	Method
BeginTransaction(String)	SqlTransaction	Method
ChangeDatabase(String)	void	Method
ChangePassword(String, String)	void	Static Method
ClearAllPools	void	Static Method
Close()	void	Method
CreateCommand()	SqlCommand	Method
EnlistDistributedTransaction(ITransaction)	void	Method
EnlistTransaction(Transaction)	void	Method
GetSchema()	DataTable	Method
GetSchema(String)	DataTable	Method
GetSchema(String, String[])	DataTable	Method
Open()	void	Method
ResetStatistics	void	Method
RetrieveStatistics	Hashtable	Method

Draft Chapter

ISBN: 0321382188

ConnectionString	String	Property
ConnectionTimeout	Int32	Property
Database	String	Property
DataSource	String	Property
FireInfoMessageOnUserErrors	Boolean	Property
PacketSize	Int32	Property
ServerVersion	String	Property
State	String	Property
StatisticsEnabled	Boolean	Property
WorkStationId	String	Property
InfoMessage	SqlInfoMessageEventHandler	Event

You can only create one `SqlConnection` at a time with the special "context connection=true" string. Attempting to create a second `SqlConnection` instance will fail, but you can create an "internal" `SqlConnection` and another external `SqlConnection` back to the same instance using an ordinary connection string. Opening this additional `SqlConnection` will start a distributed transaction, however[2], because you have multiple SPIDs (SQL Server sessions) possibly attempting to update the same data. There is no way to "knit" the two sessions together through the ADO.NET API into a single local transaction, however, as you can in an extended stored procedure with `sp_bindtoken`. You can call the `SqlConnection's` Close() method and reopen it again if you'd like although it's unlikely that you ever actually need to do this. Keeping the `SqlConnection` open doesn't use any additional resources after you originally refer to it in code.

Draft Chapter

[2] Technically, you can avoid a distributed transaction by using "enlist=false" in the connection string of the new SqlConnection. In this case the second session does not take part in the context connection's transaction.

ISBN: 0321382188

Although the same `System.Data.SqlClient.SqlConnection` class is used for both client and server code, some of the features and methods will not work inside the server.

- `ChangePassword` method
- `GetSchema` method
- Connection pooling and associated parameters and methods
- Transparent failover when database mirroring is used
- Client statistics
- `PacketSize, WorkstationID` and other client information

Commands—Making Things Happen

The `SqlClient` provider implements the `SqlCommand` class in order to execute action statements and submit queries to the database. When you already have created your connection, you can get the command object from the `CreateCommand` method on your connection, as the code in Listing 4-5 shows.

```
//get a command through CreateCommand
SqlConnection conn = new SqlConnection("context connection=true");
SqlCommand cmd = conn.CreateCommand();
```

Listing 4-5:

Create a Command from the Connection Object

Another way of getting to the command is to use one of the `SqlCommand's` constructors, which Listing 4-5 shows.

Draft Chapter

ISBN: 0321382188

```
//use constructor that takes a CommandText and Connection
string cmdStatement = "select * from authors";
SqlConnection conn = new SqlConnection("context connection=true");
SqlCommand cmd = new SqlCommand(cmdStatement, conn);
```

Listing 4-6:

Using SqlCommand's constructor

We have seen how a `SqlCommand` is created; let's now look at what we can do with the command. Table 4-3 lists the public methods, properties, and events (the table doesn't show public members inherited from `System.Object` or the extra asynchronous versions of the execute-related methods).

Table 4-3

Public Members of SqlCommand

Name	Return Value/Type	Member Type
`Constructor()`		Constructor
`Constructor(String)`		Constructor
`Constructor(String, SqlConnection)`		Constructor
`Constructor(String, SqlConnection, SqlTransaction)`		Constructor
`Cancel()`	`void`	Method
`CreateParameter()`	`SqlParameter`	Method
`Dispose()`	`void`	Method
`ExecuteNonQuery()`	`int`	Method
`ExecuteReader()`	`SqlDataReader`	Method

Draft Chapter

ExecuteReader(CommandBehavior)	SqlDataReader	Method
ExecuteScalar()	Object	Method
ExecuteXmlReader()	XmlReader	Method
Prepare()	void	Method
ResetCommandTimeout	void	Method
CommandText	String	Property
CommandTimeout	int	Property
CommandType	CommandType	Property
Connection	SqlConnection	Property
Notification	SqlNotificationRequest	Property
NotificationAutoEnlist	Boolean	Property
Parameters	SqlParameterCollection	Property
Transaction	SqlTransaction	Property
UpdatedRowSource	UpdateRowSource	Property
StatementCompleted	StatementCompletedEventHandler	Event

For those of you who are used to the `SqlClient` provider, most of the members are recognizable, but as with the connection object used inside when used inside of SQL Server, there are some differences.

- The new asynchronous execution methods are not available when running on the server.
- You can have multiple `SqlCommands` associated with the special "context connection", but cannot have multiple active `SqlDataReaders` at the same time on this connection. This functionality, known as multiple active resultsets or MARS, is only available when using the data provider from a client.
- You cannot cancel a `SqlCommand` inside a stored procedure using the

Draft Chapter

ISBN: 0321382188

SqlCommand's Cancel method.

- SqlNotificationRequest and SqlDependency do not work with commands issued inside SQL Server

When you execute parameterized queries or stored procedures, you specify the parameter values through the Parameters property of the SqlCommand class. This property can contain a SqlParameterCollection that is a collection of SqlParameter instances. The SqlParameter instance contains a description of the parameter and also the parameter value. Properties of the SqlParameter class include parameter name, data type (including precision and scale for decimal parameters), parameter length, and parameter direction. The SqlClient provider uses named parameters rather than positional parameters. Use of named parameters means the following.

- The parameter name is significant; the correct name must be specified.
- The parameter name is used as a parameter marker in parameterized SELECT statements, rather than the ODBC/OLE DB question mark parameter marker.
- The order of the parameters in the collection is not significant.
- Stored procedure parameters with default values may be omitted from the collection; if they are omitted, the default value will be used.
- Parameter direction must be specified as a value of the ParameterDirection enumeration.

This enumeration contains the values Input, Output, InputOutput, and ReturnCode. Although Chapter 3 mentioned that in T-SQL all parameters defined as OUTPUT can also be used for input, the SqlClient provider (and ADO.NET is general) is more precise. Attempting to use the wrong parameter direction will cause an error, and if you specify

Draft Chapter

ISBN: 0321382188

`ParameterDirection.Output`, input values will be ignored. If you need to pass in a value to a T-SQL procedure that declares it as `OUTPUT`, you must use `ParameterDirection.InputOutput`. An example of executing a parameterized T-SQL statement follows.

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();

// set the command text
// use names as parameter markers
cmd.CommandText =
 "insert into jobs values(@job_desc, @min_lvl, @max_lvl)";

// names must agree with markers
// length of the VarChar parameter is deduced from the input value
cmd.Parameters.Add("@job_desc", SqlDbType.VarChar);
cmd.Parameters.Add("@min_lvl", SqlDbType.TinyInt);
cmd.Parameters.Add("@max_lvl", SqlDbType.TinyInt);

// set values
cmd.Parameters[0].Value = "A new job description";
cmd.Parameters[1].Value = 10;
cmd.Parameters[2].Value = 20;

// execute the command
// should return 1 row affected
int rows_affected = cmd.ExecuteNonQuery();
```

Draft Chapter

ISBN: 0321382188

Listing 4-7:

Using A Parameterized SQL statement

Obtaining Results

Execution of SQL commands can return the following:

- A numeric return code
- A count of rows affected by the command
- A single scalar value
- One or more multirow results using SQL Server's "default (cursorless) behavior"
- A stream of XML

Some commands, such as a command that executes a stored procedure, can return more than one of these items—for example, a return code, a count of rows affected, and many multirow results. You tell the provider which of these output items you want by using the appropriate method of `SqlCommand, as shown in Table 4-4.`

Table 4-4

How to Obtain Different Result Types

Result Desired	**Mechanism to Obtain It**
Return code	Parameter with `ParameterDirection` of `ReturnCode`
Count of rows affected	Returned value from `SqlCommand.ExecuteNonQuery` Or Use `SqlCommand.ExecuteReader` and

Draft Chapter

ISBN: 0321382188

	`SqlDataReader.RecordsAffected`
Scalar value	Use `SqlCommand.ExecuteScalar`
Cursorless mode results	Use `SqlCommand.ExecuteReader`
XML stream	Use `SqlCommand.ExecuteXmlReader`

When you return data from a `SELECT` statement, it is a good idea to use the "lowest overhead" choice. Because of the amount of internal processing and the number of object allocations needed, `ExecuteScalar` may be faster than `ExecuteReader`. Of course, you need to consider the shape of the data that is returned. Using `ExecuteReader` to return a forward-only, read-only cursorless set of results is always preferred over using a server cursor. An example of when to use each results-returning method follows.

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();

// 1. this is a user-defined function
// returning a single value (authorname) as VARCHAR
cmd.CommandText = "GetFullAuthorNameById";
// required from procedure or UDF
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@id", "172-32-1176");

String fullname = (String)cmd.ExecuteScalar();
// use fullname
cmd.Parameters.Clear();

// 2. returns one row
cmd.CommandText = "GetAuthorInfoById";
```

Draft Chapter

```
// required from procedure or UDF
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@id", "172-32-1176");

SqlDataReader rdr1 = cmd.ExecuteReader();
// use fields in SqlDataReader
rdr1.Close();
cmd.Parameters.Clear();

// 3. returns multiple rows
cmd.CommandText = "select * from authors";
cmd.CommandType = CommandType.Text;

SqlDataReader rdr2 = cmd.ExecuteReader();
while (rdr2.Read())
  // process rows in SqlDataReader
  { }
rdr2.Close();
```

Draft Chapter

Listing 4-8:

Returning rows with SqlClient

`SqlDataReader` encapsulates multiple rows that can be read in a forward-only manner. You move to the next row in the set by using the `SqlDataReader's Read()` method, as shown in the preceding example. After you call `ExecuteReader`, the resultant `SqlDataReader` is positioned before the first row in the set, and an initial `Read` positions it at the first row. The `Read` method returns false when there are no more rows in the set. If more than one rowset is available, you move to the next rowset by calling `SqlDataReader's NextResult` method.

ISBN: 0321382188

While you are positioned on a row, the `IDataRecord` interface can be used to read data. You can use loosely typed ordinals or names to read the data in single columns. Using ordinals or names is a syntactic shortcut to using `IDataRecord.GetValue(n)`. This returns the value as a .NET `System.Object`, which must be cast to the correct type.

If you know the data type of the value, you can use more strongly typed column accessors. Both SQL Server providers have two kinds of strongly typed accessors. `IDataReader.GetDecimal(n)` is an example; this returns the value of the first column of the current row as a .NET `System.Decimal` data type. If you want full SQL Server type fidelity it is better to use `SqlDataReader`'s SQL Server–specific accessors such as `IDataReader.GetSqlDecimal(n)`; these return instances of structures from the `System.Data.SqlTypes` namespace. These types are isomorphic with SQL Server data types; examples of their use and reasons they are preferable to the .NET data types when uses inside the server were covered in Chapter 3. An example of using each type follows.

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "select * from authors";
cmd.CommandType = CommandType.Text;

SqlDataReader rdr = cmd.ExecuteReader();
while (rdr.Read() == true)
{
  string s;
  // 1. Use ordinals or names
  //    explicit casting, if you know the right type
  s = (string)rdr[0];
  s = (string)rdr["au_id"];
```

Draft Chapter

ISBN: 0321382188

```
  // 2. Use GetValue (must cast)
  s = (string)rdr.GetValue(0);

  // 3. Strong typed accessors
  s = rdr.GetString(0);

  // 4. Accessors for SqlTypes
  SqlString s2 = rdr.GetSqlString(0);
}
```

Listing 4-9:

Getting column values from a SqlDataReader

Although you can process results obtained inside .NET procedural code, you can also pass these items back to the client. This is accomplished through the `SqlPipe` class, which is described later in the chapter. Note that each of the classes returns rows, which must be processed sequentially; these results cannot be updated in place.

Transactions

Multiple SQL operations within a stored procedure or user-defined function can be executed individually or composed within a single transaction. Composing multi-statement procedural code inside a transaction ensures that a set of operations has "ACID" properties. ACID is an acronym for the following:

- Atomicity—Either all the operations in a transaction will succeed or none of them will.

Draft Chapter

ISBN: 0321382188

- Consistency— The transaction transforms the database from one consistent state to another.
- Isolation—Each transaction has its own view of the database state.
- Durability—These behaviors are guaranteed even if the database or host operating system fails—for example, because of a power failure.

You can use transactions in two general ways within the `SqlClient` managed provider—by starting a transaction by using the `SqlConnection's BeginTransaction` method, or by using declarative transactions using `System.Transaction.TransactionScope`. The TransactionScope is part of a new library in .NET 2.0, the System.Transactions library. A simple example of each method follows.

```
// Example 1: start transaction using the API
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlTransaction tx = conn.BeginTransaction();
// do some work
tx.Commit();
conn.Dispose();
```

```
// Example 2: start transaction using Transaction Scope
using System.Data.SqlClient;
using System.Transactions;

using (TransactionScope ts = new TransactionScope())
{
  SqlConnection conn = new SqlConnection("context connection=true");
  // connection auto-enlisted in transaction on Open()
  conn.Open();
```

Draft Chapter

ISBN: 0321382188

```
  // transactional commands here
  conn.Close();
  ts.Complete();
} // transaction commits when TransactionScope.Dispose called implicitly
```

Listing 4-10:

SqlClient can use two different coding styles for transactions

If you've done any ADO.NET coding before, you've probably run into the `BeginTransaction` method. This method encapsulates issuing a `BEGIN TRANACTION` statement in T-SQL. The `TransactionScope` requires a bit more explanation.

The `System.Transactions` library is meant to provide a representation of the concept of a transaction in the managed world. It is also a lightweight way to access MSDTC, the distributed transaction coordinator. It can be used as a replacement for the automatic transaction functionality in COM+ exposed by the `System.EnterpriseServices` library, but it does not require the components that use it to be registered in the COM+ catalog. `System.EnterpriseServices` cannot be used in .NET procedural code that runs in SQL Server. To use automatic transactions with `System.Transactions`, simply instantiate a `TransactionScope` object with a using statement and any connections that are opened inside the using block will automatically be enlisted in the transaction. The transaction will be committed or rolled back when you exit the using block and the `TransactionScope`'s `Dispose` method is called. Notice that the default behavior when `Dispose` is called is to roll back the transaction. To commit the transaction you need to call the `TransactionScope`'s `Complete` method.

Draft Chapter

ISBN: 0321382188

In SQL Server 2005, using the `TransactionScope` starts a local, not a distributed transaction. This is the behavior whether `TransactionScope` is used with client-side code or SQLCLR procedures unless there is already a transaction started when the SQLCLR procedure is invoked. This phenomenon is illustrated below.

```
-- Calling a SQLCLR procedure that uses TransactionScope

EXECUTE MySQLCLRProcThatUsesTransactionScope -- local transaction
GO

BEGIN TRANSACTION
-- other T-SQL statements
EXECUTE MySQLCLRProcThatUsesTransactionScope -- distributed transaction
COMMIT
GO
```

The transaction actually begins when `Open` is called on the `SqlConnection`, not when the `TransactionScope` instance is created. If more than one `SqlConnection` is opened inside a `TransactionScope` both connections are enlisted in a distributed transaction when the second connection is opened. The transaction on the first connection actually changes from a local transaction to a distributed transaction. Recall that you can only have a single instance of the "context connection", so opening a second connection really means opening a connection using `SqlClient` and a network library. Most often you'll be doing this specifically to start a distributed transaction with another database. Because of the network traffic involved and the nature of the two-phase commit protocol used by distributed transactions, a distributed transaction will be much higher overhead than a local transaction.

Draft Chapter

Using BeginTransaction and TransactionScope work identically in the simple case. But some database programmers like to make each procedure useable and transactional stand-alone or when called when a transaction already exists. To accomplish this, you would put transaction code in each procedure. When one procedure with transaction code calls another procedure with transaction code, this is called composing transactions. SQL Server supports nesting of transactions and named savepoints but not autonomous (true nested) transactions. So, using a T-SQL procedure "X" as an example,

```
CREATE PROCEDURE X
AS
BEGIN TRAN
-- work here
COMMIT
```

calling it standalone (`EXECUTE X`) means the work is in a transaction. Calling it from procedure "Y":

```
CREATE PROCEDURE Y
AS
BEGIN TRANSACTION
-- other work here
EXECUTE X
COMMIT
```

doesn't start an autonomous transaction (a second transaction with a different scope), the `BEGIN TRANSACTION` in X merely increases a T-SQL variable `@@TRANCOUNT` by 1. Two error messages are produced when you roll back in procedure X while it's being called by procedure Y.

```
Msg 266, Level 16, State 2, Procedure Y, Line 0
```

Draft Chapter

```
Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK
TRANSACTION statement is missing. Previous count = 1, current count = 0.
Msg 3902, Level 16, State 1, Procedure X, Line 5
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

I'd like to emulate this behavior in SQLCLR, i.e. have a procedure that acts like X, and can be used standalone or composed. I can do something akin to T-SQL (and get the interesting rollback behavior with a slightly different error number) using the `BeginTransaction` method on the context `SqlConnection`. Using a TransactionScope has a different behavior, however. If I have a SQLCLR proc that looks like this (condensed version):

```
public static void X {
  using (TransactionScope ts = new TransactionScope())
  using (
    SqlConnection conn = new SqlConnection("Context connection=true"))
  {
  conn.Open();
  ts.Complete();
  }
}
```

If SQLCLR X is used standalone, all is well and the `TransactionScope` code gets a local transaction. If SQLCLR X is called from procedure Y (above) then `SqlConnection's Open` starts a distributed transaction. Apparently it HAS to be this way, at least for now, because of how `TransactionScope` works. Local transactions don't expose the events that TransactionScope needs to compose transactions.

If you WANT a distributed transaction composed with your outer transaction (your `SqlConnection` is calling to another instance for example), USE `TransactionScope`, if you

ISBN: 0321382188

DON'T want one, use `SqlConnection`'s `BeginTransaction`. It won't act any different from T-SQL (except you do get a different error number) if you roll back inside an "inner" transaction. But you get a nesting local transaction with `BeginTransaction`.

Here's an example of using a distributed transaction with the `TransactionScope`:

```
public static void DoDistributed() {
  string ConnStr =
    "server=server2;integrated security=sspi;database=pubs";
  using (TransactionScope ts = new TransactionScope())
  using (SqlConnection conn1 =
         new SqlConnection("Context connection=true"))
  using (SqlConnection conn2 =
         new SqlConnection(ConnStr))
  {
  conn1.Open();
  conn2.Open();
  // do work on connection 1
  // do work on connection 2
  // ask to commit the distributed transaction
  ts.Complete();
  }
}
```

Listing 4-11:

A distributed transaction using TransactionScope

Draft Chapter

ISBN: 0321382188

TransactionScope Exotica

You can use options of the `TransactionScope` class to compose multiple transactions in interesting ways. For example, you can start multiple transactions (but not on the context connection) by using a different `TransactionScopeOption`. For example, the following code will begin a local transaction using the context connection and then begin an autonomous transaction using a connection to the same server.

```
public static void DoPseudoAutonomous() {
  string ConnStr =
    "server=sameserver;integrated security=sspi;database=samedb";
  using (TransactionScope ts1 = new TransactionScope())
  using (SqlConnection conn1 =
          new SqlConnection("context connection=true"))
  {
  conn1.Open();
  // do work on connection 1, then
  {
      using (TransactionScope ts2 =
              new TransactionScope(TransactionScopeOption.RequiresNew))
      using (SqlConnection conn2 = new SqlConnection(ConnStr))
      {
         conn2.Open();
         // do work on connection 2
         ts2.Complete();
      }
  // ask to commit transaction1
  ts1.Complete();
  }
}
```

Draft Chapter

ISBN: 0321382188

Listing 4-12:

Producing the equivalent of a autonomous transaction

This code works because it uses a second connection to the same server to start a second transaction. This second connection is separate from the first one, not an autonomous transaction on the same connection. The end result is the same as you would get from an autonomous transaction; you just need two connections (the context connection and a second connection) to accomplish it.

Attempting to use any `TransactionScopeOption` other than the default `TransactionRequired` fails if there already is an existing transaction (as we saw before, when BEGIN TRANSACTION was called in T-SQL before EXECUTE on the SQLCLR procedure) and you attempt to use to context connection. You'll get a message saying "no autonomous transaction".

```
-- Calling a SQLCLR procedure that uses TransactionScope
-- with an option other than TransactionRequired

EXECUTE DoPseudoAutonomous -- works
GO


BEGIN TRANSACTION
-- other T-SQL statements
EXECUTE DoPseudoAutonomous -- fails, "no autonomous transaction"
COMMIT
GO
```

Draft Chapter

ISBN: 0321382188

Listing 4-13:

Attempting to use autonomous transactions on a single connection fails

This is because SQL Server doesn't support autonomous transactions on a single connection.

Best Practices

With all of these options and different behaviors, what's the best and easier thing to do to ensure that your local transactions always work correctly in SQLCLR procedures? At this point, because SQL Server 2005 doesn't support autonomous transactions on the same connection, `SqlConnection`'s `BeginTransaction` method is the best choice for local transactions. In addition, you need to use the `Transaction.Current` static properties in `System.Transactions.dll` to determine if a transaction already exists, that is, if the caller already started a transaction. Here's a strategy that works well whether or not you compose transactions.

Draft Chapter

```
// Works whether caller has transaction or not
public static int ComposeTx()
{
  int returnCode = 0;
  // determine if we have transaction
  bool noCallerTx = (Transaction.Current == null);
  SqlTransaction tx = null;

  SqlConnection conn = new SqlConnection("context connection=true");
  conn.Open();

  if (noCallerTx)
    tx = conn.BeginTransaction();
```

```
  try {
    // do the procedure's work here
    SqlCommand workcmd = new SqlCommand(
      "INSERT jobs VALUES('New job', 10, 10)", conn);
    if (tx != null)
      workcmd.Transaction = tx;
    int rowsAffected = workcmd.ExecuteNonQuery();

    if (noCallerTx)
      tx.Commit();
  }

  catch (Exception ex) {
    if (noCallerTx) {
      tx.Rollback();
      // raise error - covered later in chapter
    }
    else {
      // signal an error to the caller with return code
      returnCode = 50010;
    }
  }
  conn.Dispose();
  return returnCode;
}
```

Draft Chapter

ISBN: 0321382188

Listing 4-14:

A generalized strategy for nesting transactions

For distributed transactions, as well as "pseudo-autonomous" transactions described earlier, you must use `TransactionScope` or a separate second connection back to the server using `SqlClient`. If you don't mind the behavior that nesting transactions with `TransactionScope` forces a distributed transaction, and the extra overhead caused by MSDTC, you can use `TransactionScope` all the time. Finally, if you know your procedure won't be called with an existing transaction, you can use either `BeginTransaction` or `TransactionScope`. Refraining from nesting transactions inside nested procedures may be a good strategy until this gets sorted out.

Pipe

In the section on results, we mentioned that you had a choice of processing results in your procedural code as part of its logic or returning the results to the caller. Consuming `SqlDataReaders` or the stream of XML in procedural code makes them unavailable to the caller; you cannot process a cursorless mode result more than once. The code for in-process consumption of a `SqlDataReader` is identical to `SqlClient`; you call `Read()` until no more rows remain. To pass a resultset back to the client, you need to use a special class, `SqlPipe`.

The `SqlPipe` class represents a channel back to the client, this is a TDS output stream if the TDS protocol is used for client communication. You obtain a `SqlPipe` by using the static `SqlContext.Pipe property`. Rowsets, single rows, and messages can be written to the pipe. Although you can get a `SqlDataReader` and return it to the client through the `SqlPipe`, this is less efficient than just using a new special method for the `SqlPipe` class: `ExecuteAndSend`.

Draft Chapter

This method executes a `SqlCommand` and points it directly to the `SqlPipe`: An example is illustrated below.

```
public static void getAuthorsByState(SqlString state)
{
  SqlConnection conn = new SqlConnection("context connection=true");
  conn.Open();
  SqlCommand cmd = conn.CreateCommand();
  cmd.CommandText = "select * from authors where state = @state";
  cmd.Parameters.Add("@state", SqlDbType.VarChar);
  cmd.Parameters[0].Value = state;
  SqlPipe pipe = SqlContext.Pipe;
  pipe.ExecuteAndSend(cmd);
}
```

Listing 4-15:

Using SqlPipe to return rows to the client

In addition to returning an entire set of results through the pipe, `SqlPipe`'s `Send` method lets you send an instance of the `SqlDataRecord` class. You can also batch the send operations however you'd like. An interesting feature of using `SqlPipe` is that the result is streamed to the caller immediately as fast as you are able to send it, taking into consideration that the client stack may do row buffering. This may improve performance at the client because you can process rows as fast as they are sent out the pipe. Note that you can combine executing a command and sending the results back through `SqlPipe` in a single operation with the `ExecuteAndSend` convenience method, using a `SqlCommand` as a method input parameter.

Draft Chapter

ISBN: 0321382188

SqlPipe also contains methods for sending scalar values as messages and affects how errors are exposed. We'll talk about error handling practices next. The entire set of methods exposed by SqlPipe is shown in Table 4-8.

Table 4-8

Methods of the SqlPipe Class

Method	What It Does
ExecuteAndSend(SqlCommand)	Execute command, return results through SqlPipe
Send(String)	Send a message as a string
Send(SqlDataReader)	Send results through SqlDataReader
Send(SqlDataRecord)	Send results through SqlDataRecord
SendResultsStart(SqlDataRecord)	Start sending results
SendResultsRow(SqlDataRecord)	Send a single row after calling SendResultsStart
SendResultsEnd()	Indicate finished sending rows

Draft Chapter

There is also a boolean property on the SqlPipe class, IsSendingResults, that enables you to find out if the SqlPipe is busy. Because multiple active resultsets are not supported when you're inside SQL Server, attempting to execute another method that uses the pipe while it's busy will procedure an error. The only exception to this rule is the SendResultsStart, SendResultsRow, and SendResultsEnd are used together to send results one row at a time.

SqlPipe is only available for use inside a SQLCLR stored procedure. Attempting to get the SqlContext.Pipe value inside a user-defined function returns a null instance. This is because sending rowsets is not permitted in a user-defined function. Within a stored procedure however, you can not only send rowsets through the SqlPipe by executing a command that returns a rowset; you can synthesize your own. Synthesizing rowsets involves the use of two server-specific classes we haven't seen before, SqlDataRecord and SqlMetaData.

ISBN: 0321382188

Creating and Sending new Rowsets

We've already seen that you can execute commands that return rowsets and send these to the client. You might want to execute a command that returns a rowset and then augment or change the rowset before sending it on. Or you might get data that is not in the database, like an RSS feed or other Web Service, and choose to expose that data as a set of columns and rows. This is similar to the functionality that you can expose using table-valued functions, but without the capability to perform SQL using a where clause on the result. Your rowset will appear as one of the outputs of the stored procedure just as though it came from SQL Server's data.

You'd accomplish creating and sending a rowset by using the following steps:

- Create an array of `SqlMetaData` instances that describes the data in each column.
- Create an instance of `SqlDataRecord`. You must associate the array of `SqlMetaData` instances with the `SqlDataRecord`.
- Populate the values in the `SqlDataRecord` using either weakly or strongly typed setter methods. `
- Call `SqlPipe's SendResultsStart` method to send the first row. This sends the metadata back to the client.
- Populate the values in the `SqlDataRecord` using either weakly or strongly typed setter methods.

Use `SqlPipe's SendResultsRow` method to send the data.

Use `SqlPipe's SendResultsEnd` method to indicate that the rowset is complete.

First, we'll talk about using the SqlMetaData class. `SqlMetaData` is a class that is used to completely describe a single column of data. It can be used with `SqlDataRecord` instances.

Draft Chapter

ISBN: 0321382188

`SqlMetaData` instances encapsulate the information in the extended metadata from new format extended TDS "describe packets" used by SQL Server 2005 as well as working with earlier versions of TDS. Listing 4-16 lists the properties exposed by the `SqlMetaData` class.

```
class SqlMetaData {

  //datatype info
  public SqlDbType SqlDbType;  // SqlDbType enum value
  public DbType DbType;       // DbType enum value
  public Type Type;           // .NET data type
  public string TypeName;     // .NET type name
  public string UdtTypeName; // SQL Server 3-part type name

  //metadata info
  public bool IsPartialLength;
  public long LocaleId;
  public long Max;
  public long MaxLength;
  public byte Precision;
  public byte Scale;
  public string Name;         // column name
  public SqlCompareOptions CompareOptions;

  // XML schema info for XML data type
  public string XmlSchemaCollectionDatabase;
  public string XmlSchemaCollectionName;
  public string XmlSchemaCollectionOwningSchema;
};
```

Draft Chapter

Listing 4-16:

Fields of the SqlMetaData class

Let's use `SqlDataRecord`, `SqlMetaData`, and `SqlPipe` to create and send rows from a simple synthesized rowset. The `Thread.Sleep()` calls are inserted so that you can observe pipelining in action with this type of rowset. Rows are sent as soon as the SQL engine has spare cycles to send them.

```
// other using statements elided for clarity
using System.Threading;
using Microsoft.SqlServer.Server;
public static void Pipeline()
{
  SqlPipe p = SqlContext.Pipe;

  // a single column in each row
  SqlMetaData[] m = new SqlMetaData[1]
     {new SqlMetaData("colname", SqlDbType.NVarChar, 5) };
  SqlDataRecord rec = new SqlDataRecord(m);
  rec.SetSqlString(0, "Hello");

  p.SendResultsStart(rec);
  for (int i=0;i<10000;i++)
      p.SendResultsRow(rec);

  Thread.Sleep(10000);
  for (int i = 0; i < 10000; i++)
      p.SendResultsRow(rec);

  Thread.Sleep(10000);
```

Draft Chapter

ISBN: 0321382188

```
  for (int i = 0; i < 10000; i++)
      p.SendResultsRow(rec);

  Thread.Sleep(10000);
  p.SendResultsEnd();
}
```

Listing 4-17:

Synthesizing a rowset using SqlDataRecord and SqlMetaData

Using the WindowsIdentity

Procedural code executes SQL in the context of the caller by default. In SQL Server 2005, you can also specify alternate execution contexts by using the new "`WITH EXECUTE AS`" clause. We'll talk more about this in the chapter on security. When you're using an assembly created with the `EXTERNAL_ACCESS` or `UNSAFE` permission sets, however, you're allowing to access resources outside of SQL Server. These resources can include other database instances, the file system, and external Web Services. Some of these resources may require authentication using a Windows security principal. If you're running the stored procedure as a SQL Server user associated with a Windows login, you can choose to impersonate the logged in user. If you don't choose to impersonate the logged in user (or if you're running the procedure while logged in as a SQL login), all external access takes place under the identity of the Windows principal running the SQL Server process. Under the "principal of least privilege" this should most likely be a user with only the privileges required to run SQL Server. This principal probably won't have access to the external resources that you want.

Draft Chapter

ISBN: 0321382188

You can get the Windows Identity of the user running the procedure by using the `SqlContext.WindowsIdentity` property, as shown in the example below.

```
// other using statements elided for clarity
using System.Security.Principal;
using Microsoft.SqlServer.Server
public static void GetFile(string filename)
{
  string s;
  using (WindowsIdentity id = SqlContext.WindowsIdentity)
  {
    WindowsImpersonationContext c = id.Impersonate();
    StreamReader sr = new StreamReader(filename);
    s = sr.ReadLine();
    sr.Close();
    c.Undo();
  }
    SqlContext.Pipe.Send(s);
}
```

Listing 4-18:

Using Impersonation in a SQLCLR stored procedure

Draft Chapter

Once you have the `WindowsIdentity`, you can call its Impersonate method. Be sure to save the `WindowsImpersonationContext` that is returned because you'll have to call its `Undo` method to undo the impersonation. Note that you can only do data access before calling Impersonate or after calling `Undo`. Attempting to use an impersonated context for data access will

fail. Also, bear in mind that if you're executing the stored procedure as a SQL login rather than a Windows login, the `WindowsIdentity` will be null.

Calling a Web Service from SQLCLR

You would most likely be using impersonation from within a procedure in an assembly that is cataloged as `PERMISSION_SET = EXTERNAL_ACCESS` or `UNSAFE`. This is because, when accessing external Windows resources, you'll need to impersonate the login that executed the procedure. One such case is calling a Web Service.

If you call a Web Service from a .NET function or procedure by using the low-level APIs in `System.Web`, there are no special considerations other than impersonate apply. However, if you are use the Web Service client proxy classes generated by the .NET `WSDL.exe` utility (or "Add Web Reference" from within Visual Studio) there are a few extra steps you need to perform. The Web Service proxy classes generated will attempt to use dynamic assembly generation at procedure run time to build and load the `XmlSerializer` classes that it uses to communicate with the Web Service. This is not allowed in SQLCLR, regardless of the `PERMISSION_SET` of the assembly. The way around this limitation is to build your `XmlSerializer` classes in advance, using a command line utility, SGEN.exe. Here's how you would accomplish this.

I'd like to call a Web Service names "`StockService`" and have a URL, `http://stockbroker/StockService?WSDL` where I can obtain Web Service Description Language, metadata that is needed to build the client proxy. To build the proxy, issue the following command:

```
WSDL.exe http://stockbroker/StockService?WSDL
```

Draft Chapter

ISBN: 0321382188

There are additional parameters available for the WSDL.exe utility that are outside the scope of this discussion. Now that you have the proxy class (in the default case it will be named StockService.cs) you compile it into an assembly and reference it in your program. This class consists of an object that represents your Web Service call, so calling out to the Web Service is no more complex than calling a method on an object. In your SQLCLR procedure, the code to call the Web Service looks like this:

```
[Microsoft.SqlServer.Server.SqlFunction]
[return: SqlFacet(Precision = 9, Scale = 2)]
public static decimal GetStockWS(string symbol)
{
Decimal price;
using (WindowsIdentity id = SqlContext.WindowsIdentity)
  {
    WindowsImpersonationContext c = id.Impersonate();
    StockService s = new StockService();
    // use the current credentials
    s.Credentials =
      System.Net.CredentialCache.DefaultNetworkCredentials;
    price = s.GetStockPrice(symbol);
    c.Undo();
  }
return price;
}
```

Listing 4-19:

Calling a Web Service from a SQLCLR user-defined function

Draft Chapter

ISBN: 0321382188

In order for this code to work you must pregenerate the required serializer for the proxy with the SGEN.exe utility, like this:

```
SGEN.exe StockService.dll
```

The utility produces an assembly named `StockService.XmlSerializers.dll`. You must catalog this assembly to SQL Server so that your Web Service client stored procedure (GetStockWS above) can use it. But there's another SQLCLR reliability-related issue to address before you do this. Our overall build steps for the proxy so far look as shown in figure 4-4:

```
  rem from the Command line
  rem(ensure these .NET utilities are in your path)
  cd \temp
  rem produce StockService.cs, need to edit this file
① C:temp>WSDL.exe http://stockbroker/StockService?WSDL
  rem produce StockService.dll
② C:temp>CSC /target:library StockService.cs
  rem produce StockService.XmlSerializers.dll
③ C:\temp>SGEN StockService.dll

  -- From SQL Server Management Studio
④ CREATE ASSEMBLY stockproxy
  FROM 'c:\temp\StockService.dll'
  GO
⑤ CREATE ASSEMBLY stockser
  FROM 'c:\temp\StockService.XmlSerializers.dll'
  GO
  -- now, reference both of these
  -- assemblies in your SQLCLR project
```

Figure 4-4

Building a Web Service proxy and cataloging it to SQL Server

Use the WSDL (1) utility to build a proxy from the URL for the web service. It will build a C# file whose name is based on the name of the service, in this case StockService.cs. StockService.cs will include both a synchronous and asynchronous implementation of the proxy. You must remove the asynchronous one by removing it. You need keep only the constructor and

Draft Chapter

ISBN: 0321382188

the methods (2) and remove everything else (3). Build a library out of StockService.cs using the C# compiler (4). You must also build an assembly that contains the xml serialization code using the SGEN utility (5). This utility produces a dll (6) that contains the xml serialization implementation.

The only problem with this is that, when WSDL.exe produces an "automatic" proxy class two types of invocation methods are produced: a method to execute the Web Service synchronously (`GetStockPrice` in this case) and a pair of methods to execute the Web Service asynchronously (`BeginGetStockPrice` and `EndGetStockPrice`). The asynchronous methods must be edited out before compiled StockService.cs, out the resulting assembly will need to be cataloged with `PERMISSION_SET = UNSAFE`. If your DBA doesn't mind `PERMISSION_SET = UNSAFE`, this is fine, but you won't be using the methods, so let's go in and remove them. In the generated code `StockService.cs` remove or comment out:

1. The `BeginGetStockPrice` and `EndGetStockPrice` methods
2. The `GetStockPriceCompletedEventHandler` variable
3. The `GetStockPriceCompletedEventHandler` method
4. The `GetStockPriceCompletedEventHandlerEventArgs` method

as shown in Figure 4-5.

Draft Chapter

```
Visual Studio 2005 Command Prompt
C>wsdl ① http://localhost:7496/chap%205/Service.asmx
Microsoft (R) Web Services Description Language Utility
[Microsoft (R) .NET Framework, Version 2.0.50727.42]
Copyright (C) Microsoft Corporation. All rights reserved.
Writing file 'C:\serviceClient\Service.cs'.

public partial class Service … {
   private System.Threading.SendOrPostCallback
      GetStockPriceOperationCompleted;

   public Service() {…
 ② public float GetStockPrice(string Symbol) {
   …
   }
   public System.IAsyncResult BeginGetStockPrice(…
   public float EndGetStockPrice(…
   public void GetStockPriceAsync(…
   public void GetStockPriceAsync(…
 ③ private void OnGetStockPriceOperationCompleted(…
   public new void CancelAsync(…
   public delegate void
      GetStockPriceCompletedEventHandler(…
   public partial class GetStockPriceCompletedEventArgs…
}
Visual Studio 2005 Command Prompt
C>csc ④ /target:library Service.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50
Copyright (C) Microsoft Corporation 2001-2005. All rights r

C>sgen ⑤ service.dll
Microsoft (R) Xml Serialization support utility
[Microsoft (R) .NET Framework, Version 2.0.50727.42]
Copyright (C) Microsoft Corporation. All rights reserved.
Serialization Assembly Name: Service.XmlSerializers, Versic
utral, PublicKeyToken=null.
Generated serialization assembly for assembly C:\serviceCli
C:\serviceClient\Service.XmlSerializers.dll'. ⑥
```

Figure 4-5

Commenting out the Asynchronous Methods in a WSDL.exe-generated proxy

Now our proxy class assembly, `StockService.dll`, will be able to use PERMISSION_SET = SAFE. We'll return to the subject of `XmlSerializers` in chapter 5, and see how they are used by user-defined types.

ISBN: 0321382188

Exception Handling

One of the reasons .NET procedures are appealing is that .NET has real structured exception handling. SQL Server 2005 includes improvements in error handling (see `BEGIN-END TRY`, `BEGIN-END CATCH` in Chapter 7, T-SQL Enhancements), but this isn't true structured exception handling. With .NET procedures you can accomplish something that wasn't available in previous versions of SQL Server. You can choose to consume the exception and not return the error to the caller, as shown in the following example.

```
public static void EatException()
{
  try
  {
    // cause a divide-by-zero exception
    int i = 42;
    int j = 0;
    j = i / j;
  }
  catch (Exception e)
  {
    SqlContext.Pipe.Send("Ate the exception");
    SqlContext.Pipe.Send(e.Message);
  }
}

-- Run this T-SQL code to test it
-- The transaction commits and the row is inserted
-- prints:
--   (1 row(s) affected)
```

Draft Chapter

```
--   Ate the exception
--   Attempt to divide by zero
BEGIN TRANSACTION
INSERT INTO jobs VALUES('before exception', 10, 10)
EXECUTE EatException
COMMIT
```

Listing 4-20:

Result of consumer an exception in a SQLCLR procedure

You can even catch an error in .NET code and rethrow it as a user-defined exception. However, there is a catch (pun intended). Any unhandled exceptions that make their way out of your CLR procedure result in the same error at the client, whether the client is SQL Server Management Studio or a user application. However, if you are called from a try/catch block inside another .NET stored procedure, that procedure can catch your exception without causing an underlying T-SQL exception. The following code illustrates this.

```
public static void ExceptionThrower()
{
  try
  {
    int i = 42;
    int j = 0;
    j = i / j;
  }
  catch (Exception e)
  {
    SqlContext.Pipe.Send("In exception thrower");
    SqlContext.Pipe.Send(e.Message);
```

Draft Chapter

ISBN: 0321382188

```
    throw (e);
  }
}

public static void ExceptionCatcher()
{
  using (SqlConnection conn =
     new SqlConnection("context connection=true"))
  using (SqlCommand cmd = new SqlCommand("ExceptionThrower", conn))
  {
    try
    {
      cmd.CommandType = CommandType.StoredProcedure;
      conn.Open();
      cmd.ExecuteNonQuery();
      SqlContext.Pipe.Send("Shouldn't get here");
    }
    catch (SqlException e)
    {
      SqlContext.Pipe.Send("In exception catcher");
      SqlContext.Pipe.Send(e.Number + ": " + e.Message);
    }
  }
}
```

Listing 4-21:

Catching an exception and re-throwing it using SQLCLR

Draft Chapter

The results of using the `ExceptionThrower` procedure, both standalone and from the `ExceptionCatcher`, are shown in the following code.

```
-- exception thrower standalone
BEGIN TRANSACTION
INSERT INTO jobs VALUES('thrower', 10, 10)
EXECUTE ExceptionThrower
COMMIT

-- results in the messages window
(1 row(s) affected)
In exception thrower
Attempted to divide by zero.
Msg 6522, Level 16, State 1, Procedure ExceptionThrower, Line 0
A .NET Framework error occurred during execution of user defined routine
or aggregate 'ExceptionThrower':
System.DivideByZeroException: Attempted to divide by zero.
System.DivideByZeroException:
   at StoredProcedures.ExceptionThrower()

-- exception catcher calls exception thrower
BEGIN TRANSACTION
INSERT INTO jobs VALUES('catcher', 10, 10)
EXECUTE ExceptionCatcher
COMMIT

(1 row(s) affected)
In exception catcher
6522: A .NET Framework error occurred during execution of user defined
routine or aggregate 'ExceptionThrower':
```

Draft Chapter

```
System.DivideByZeroException: Attempted to divide by zero.
System.DivideByZeroException: at StoredProcedures.ExceptionThrower().
In exception thrower
Attempted to divide by zero.
```

Listing 4-22:

Results of catching and re-throwing an exception

Note that, in each case, the transaction will commit. The only thing that would cause the transaction to roll back would be if the call to the standalone `ExceptionThrower` was called from a T-SQL `TRY-CATCH` block. If the case of `ExceptionCatcher`, it catches and discards the exception raised by the `ExceptionThrower` (the error message comes from `ExceptionCatcher`'s write of `e.Message` to the `SqlPipe`. The only unusual thing is that we don't see the messages sent by the `ExceptionThrower`.

A disappointing thing about errors from SQLCLR procedures is that they are always wrapped in a general error with the error number 6522. This makes it more difficult to get the actual error from a client. ADO.NET or OLE DB clients can return a stack of errors, but the actual error does not appear in the error stack. Note that SQL Server Management Studio is an ADO.NET application and so when we execute the code in SSMS, we get both errors. The 6522 contains the complete text and error number of the inner (actual) error.

From ODBC and (especially) from T-SQL callers, the picture is not as rosy, however. If you've not updated your error handling to use `TRY-CATCH` you're just looking at the value of `@@ERROR`. That value is always 6522, which makes determining what happened rather difficult. In this case, your only alternative is to expose your own error via the `SqlPipe`, as we'll show soon. An inelegant workaround is to always use `SqlPipe's ExecuteAndSend` method inside of a .NET `try` block that is followed by a dummy (do-nothing) `catch` block. In this case, providing

Draft Chapter

ISBN: 0321382188

you are also using T-SQL's new `TRY-CATCH` functionality, the outer (6522) error is stripped off is the actual error information is available through the `ERROR_NUMBER`, `ERROR_MESSAGE`, and other new T-SQL functions we'll be looking at soon. Be very careful to keep the 6522 wrapper error in mind as you plan for overall error handling strategy.

To return a custom error, the tried-and-true method is best. In fact, it is the only strategy that works regardless of the client stack you're using and whether or not you're being called by T-SQL.

Define your error to the SQL Server error catalog with `sp_addmessage`.

Use the T-SQL `RAISERROR` command in `SqlCommand.Text`.

Use `SqlPipe's ExecuteAndSend` method to return the error. Wrap the call to in a .NET try-catch block with an empty catch clause.

This method follows.

```
public static SqlInt32 getAuthorWithErrors(
  SqlString au_id, out SqlString au_info)
{
  // -- In SQL Server Management Studio add custom message to be used
  // -- when an author cannot be found
  // sp_addmessage 50005, 16,
  //  'author with ssn: %s, not found in the database'
  // go

  // build a command
  SqlConnection conn = new SqlConnection("context connection=true");
  conn.Open();
  SqlCommand cmd = conn.CreateCommand();
  // build command text
```

Draft Chapter

```
cmd.CommandText =
  "select address, city, state, zip" +
  " from authors where au_id = @au_id";

// make a parameter to hold the author id
cmd.Parameters.Add("@au_id", SqlDbType.VarChar);
// put in the value of the author id
cmd.Parameters[0].Value = au_id;

SqlDataReader rec = cmd.ExecuteReader();
// make SqlString to hold result
// note that if you do not give this
// string a value it will be null
au_info = new SqlString();

// check to see if lookup was successful
if (rec.Read() == false)
{
  rec.Close();
  // lookup was not successful, raise an error
  cmd.CommandText = "Raiserror (50005, 16, 1, '" +
    au_id.ToString() +"') with seterror";
  // use the try-catch with empty catch clause
  try   { SqlContext.Pipe.ExecuteAndSend(cmd); }
  catch { }
  // this return statement will never be executed
  return 0;
}
else
{
```

Draft Chapter

```
    // lookup was successful, set au_info to information string
    au_info = String.Format("{0} {1} {2} {3}",
      rec["address"], rec["city"], rec["state"], rec["zip"]);
    rec.Close();
  }
  // nothing to return, either success returned author info in au_info
  // or error was raised
  return 0;
}
```

Listing 4-23:

Returning a user error from a SQLCLR procedure

This procedure will return the correct custom error (and not return error 6522) when invoked from T-SQL whether T-SQL's `@@ERROR` or the new T-SQL `TRY-CATCH` construct is used to return error information.

SqlTriggerContext

SQL triggers have an execution context (represented by `SqlContext`) just as stored procedures and UDFs do. The environment settings, temporary tables in scope and so on, are available to a trigger. But triggers have additional context information; in T-SQL this information consists of logical tables for DML statements as well as information about which columns were updated, in the case of an update statement. In CLR triggers, this information is made available through the `SqlTriggerContext` class. You obtain the `SqlTriggerContext` through the `SqlContext`, as shown in the following example. `SqlTriggerContext` has a property to tell you whether the triggering action was an `INSERT`, `UPDATE`, `DELETE`, or one of the new DDL or event triggers.

Draft Chapter

ISBN: 0321382188

This is handy if your trigger handles more than one action. The `IsUpdatedColumn(n)` returns information about whether the n-th column in the table or view has been changed by the `INSERT` or `UPDATE` statement that caused the trigger to fire. Finally, because SQL Server 2005 adds DDL and Event triggers to the mix, a property that exposes the `EventData` XML structure provides detailed information.

The `INSERTED` and `DELETED` logical tables work the same way that they do in T-SQL triggers; they are visible from any `SqlCommand` that you create inside your trigger. This is shown in the following code. Because individual SQL statements (including triggers) are transactional, triggers in T-SQL use the `ROLLBACK` statement if the logic in the trigger determines that a business rule has been violated, for instance. Although inside a trigger the context transaction is visible, rolling it back produces an error just as in other CLR procedural code. The way to roll back the statement inside a trigger is to use the `System.Transactions.Transaction.Current` property. An example of this follows. Note that this example is only meant to show the coding techniques involved, using T-SQL would actually be a better choice for triggers like this.

```
// other using statements elided for clarity
using System.Data.SqlClient;
using System.Transactions;

public static void AddToObsolete()
{
  // get the trigger context so you can tell what is
  // going on
  SqlTriggerContext tc = SqlContext.TriggerContext;

  // this trigger is for deletes
```

Draft Chapter

ISBN: 0321382188

```
  if (tc.TriggerAction == TriggerAction.Delete)
  {
    // Make a command
    SqlConnection conn = new SqlConnection("context connection=true");
    conn.Open();
    SqlCommand cmd = conn.CreateCommand();
    // make a command that inserts the deleted row into
    // the obsoletejobs table
    cmd.CommandText = "insert into obsoletejobs select * from deleted";

    // move the rows
    int rowsAffected = (int)cmd.ExecuteNonQuery();

    if (rowsAffected == 0)
    {
      // something bad happened, roll back
      Transaction.Current.Rollback();
    }
  }
}
```

Chapter 4 Listing 4-24:

A SQLCLR trigger that rolls back the transaction in progress

SqlClient classes that you can't use on the server

Although the same provider is used on the client and server, some of the classes that contain ancillary functionality will not be available on the server. Most often, they are not available

Draft Chapter

because they don't make sense inside the database or couldn't be hooked up to the internal context. We've mentioned most of the "prohibited on the server" constructs in the `SqlConnection` and `SqlCommand` sections. Another class that won't work in the server is `SqlBulkCopy`, which we'll see (from the client side) in Chapter 14.

In addition, you must use the `SqlClient`-specific classes inside .NET procedural code. Chapter 14 discusses the fact that .NET data providers are now base-class based. Using the base-classes such as `DbConnection, DbCommand`, and so on inside the server is not supported. Finally, chapter 14 discusses the tracing facility in ADO.NET 2.0. This built-in tracing facility is not available inside server code.

Where Are We?

This chapter looked at using the `SqlClient` managed provider inside .NET procedural code in depth. However, in the previous chapter, we stated that T-SQL is almost always best for data-centric code. When, then, is the best time to use the provider in-process? These cases include when it’s appropriate to mix .NET-specific logic with data access code. Another case might be where structured exception handling is desired. In addition, because the programming model is similar enough (though not identical) using `SqlClient` on the client, middle-tier, and server, some existing .NET code might be moved into the server for locality-of-reference benefits.

Draft Chapter