Pro SQL Server 2005 Assemblies

Robin Dewson and Julian Skinner

Apress®

Pro SQL Server 2005 Assemblies

ISBN (pbk): 1-59059-566-1

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewers: Damien Foggon, Adam Machanic, Joseph Sack, Kent Tegels
Additional Material: Adam Machanic
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Laura Cheu, Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editors: Ami Knox, Nicole LeClerc, Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Molly Sharp
Proofreader: Dan Shaw
Indexer: Julie Grady
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 3

The SQL Server .NET Programming Model

In the last chapter, we saw how to create and deploy a simple SQL Server assembly. This introduced us briefly to a couple of the new .NET classes that we use to create SQL assemblies: `SqlContext` and `SqlPipe`. These classes act as extensions to the standard .NET data provider that is used to access SQL Server data from .NET code, both from outside the server and from within a SQL assembly.

In this chapter, we'll look at the following topics:

- Using the `SqlContext` and `SqlPipe` classes
- Examining other classes that we use when creating SQL assemblies
- Using the standard ADO.NET provider for SQL Server from within a SQL assembly

The .NET Data Access Namespaces

Although data access in a SQL assembly basically uses the standard ADO.NET provider for SQL Server, there are inevitably some differences from data access from an external application, and therefore there need to be extensions. After some internal debate, Microsoft eventually decided to add the new classes to a separate .NET namespace (earlier, prerelease versions of SQL Server 2005 used an entirely different data provider for SQL assemblies, which closely mirrored the ADO.NET provider but contained a few extensions). In fact, the new combination of SQL Server 2005 and .NET 2.0 sees no fewer than four new namespaces added to the already sprawling .NET APIs dealing with data access. There are now eight .NET namespaces involved in working with SQL Server (not including the `System.Data.SqlServerCe` namespace used for working with the Pocket PC edition of SQL Server). The distinctions between them aren't always totally clear, so it's worth taking a moment to review the new and existing namespaces:

- `System.Data`: This is the root namespace for .NET data access. This namespace includes the classes and interfaces that make up the core ADO.NET architecture—in particular the `DataSet`, `DataTable`, and related classes.
- `System.Data.Common`: This namespace contains classes that are used by all .NET data providers.

- `System.Data.Design`: This new namespace contains classes used to generate strongly typed datasets.
- `System.Data.Sql`: This new namespace provides two classes that are used for SQL Server–specific functionality.
- `System.Data.SqlClient`: This is the .NET data provider for SQL Server (version 7.0 and above). It contains classes for accessing SQL Server data, both from an external .NET application and now also from within a SQL Server assembly using a new in-process provider.
- `System.Data.SqlTypes`: This namespace contains .NET types that map directly onto native SQL Server types, which allow you to avoid the overhead of implicit type conversions in SQL Server assemblies.
- `System.Transactions`: This new namespace provides a new transaction framework for .NET 2.0 that is integrated with both ADO.NET and the SQL CLR. We'll look at this namespace toward the end of the chapter.
- `Microsoft.SqlServer.Server`: This namespace provides classes that extend the .NET data provider for SQL Server to access data from within a SQL Server assembly, as well as miscellaneous classes, attributes, enumerations, and interfaces for creating SQL Server assemblies.

Four of these namespaces are new with SQL Server 2005/.NET 2.0: `System.Data.Design`, `System.Data.Sql`, `System.Transactions`, and `Microsoft.SqlServer.Server`. The last of these contains the classes we'll use for building SQL Server assemblies, so it's the classes in this namespace that we'll concentrate on in this chapter.

Accessing SQL Server Data

Most of the new classes that you use to build SQL assemblies reside in the `Microsoft.SqlServer.Server` namespace. These classes are used together with the ADO.NET provider for SQL Server (in the `System.Data.SqlClient` namespace) to access SQL Server data from within a SQL assembly. Together these are sometimes referred to as the *in-process provider*, although Microsoft seems largely to have abolished this term since the provider used for SQL assemblies was merged with the `SqlClient` provider. Apart from a couple of extensions to provide extra functionality, the chief difference between accessing SQL Server data from within a SQL assembly and from outside the database is in the way the connection is made.

Establishing the Context of a SQL Assembly

When you write the code inside a SQL assembly, you may need to know about the context under which it is running. For example, you might need to know what user account the code is executing under, and if the module that the assembly implements is a trigger, you will need to know the circumstances that caused the trigger to fire. This information is available through the `SqlContext` class, which provides an abstract representation of the context under which the assembly is running. The new extensions to the SQL Server .NET data provider include classes

that give you access to this information, and the `SqlContext` class provides a number of static properties that you can use to get references to these objects, as shown in Table 3-1.

Table 3-1. *Properties of the SqlContext Class*

Property	Return Type	Description
`IsAvailable`	`bool`	Indicates whether you can access the context connection or not. This will return `false` if the code is running outside SQL Server and `true` if it's running in a SQL assembly. You can use this property if you write code that is run both in a SQL assembly and externally.
`Pipe`	`Microsoft.SqlServer.Server.SqlPipe`	Returns a `SqlPipe` object that you can use to send messages to the SQL Server output stream.
`TriggerContext`	`Microsoft.SqlServer.Server.SqlTriggerContext`	If the assembly is a trigger, the returned object contains information about the event that fired the trigger and the columns updated.
`WindowsIdentity`	`System.Security.Principal.WindowsIdentity`	Returns a `WindowsIdentity` object that you can use to impersonate the Windows user of the current Windows login. We'll look at role-based security in Chapter 10.

For example, to get a `SqlPipe` object that you can use to send messages back to SQL Server, you would use the following:

```
SqlPipe pipe = SqlContext.Pipe;
```

Creating the Context Connection

Just as when you access SQL Server from external .NET code, you start by creating a `System.Data.SqlClient.SqlConnection` object. To connect via the in-process provider, you use the connection string `"context connection = true"`:

```
using (SqlConnection cn = new SqlConnection("context connection = true"))
{
   cn.Open();
   // Code that accesses SQL Server data
}
```

As in this example, it's good practice to enclose the code where you use the connection within a `using` statement to ensure that the connection is closed and disposed of once you're finished with it. You don't need to pass any user information into the connection because the connection exists in the current security context—the user must already be authenticated to call the object (CLR stored procedure, UDF, etc.) that the assembly implements.

One thing to note is that you can have only one context connection open within an assembly at a given time. In most circumstances this isn't a problem since, as you saw in Chapter 1, you shouldn't include threading code in a SQL assembly, so you'll be performing tasks sequentially. However, it can create problems if you're iterating through a data reader and want to perform some secondary data access for each row. You can't perform a second query while the data reader remains open. In most cases, of course, this is exactly the kind of thing T-SQL is designed to do with a single query, but you'll see an example of this when we examine Service Broker in Chapter 11. We get around it there by iterating through the data a second time, but if you really do need to open a second connection, it's possible to open one in the normal way, using the same connection string as you would use from external code. There's obviously a performance hit from this, as the `SqlClient` provider is designed for running outside the SQL Server process. You could potentially also use this technique if you need to open a second connection to SQL Server using a different security context from the current one, although in most cases impersonation using the new `EXECUTE AS` clause in the `CREATE PROCEDURE` statement would be a better option.

Making Requests Against SQL Server

Once you have a reference to the context connection, you'll need to instantiate and execute a `SqlCommand` object, exactly as you would in standard ADO.NET. This code is really just ADO.NET code, but since it's of paramount importance in creating SQL assemblies, we'll run through the basics very quickly.

Retrieving Data

The command needs to be initialized with the context connection and the SQL statement you want to execute. Once this is done, you can call one of the `Execute` methods to execute the command. For example, to execute a simple `SELECT` statement, getting the data into a `SqlDataReader`, you'd use code like this:

```
using (SqlConnection cn = new SqlConnection("context connection = true"))
{
   cn.Open();
   SqlCommand cmd = new SqlCommand("SELECT * FROM Person.Address", cn);
   SqlDataReader reader = cmd.ExecuteReader();
```

Once you have the reader, you can use it to iterate one row at a time through a resultset of data returned by a query. `SqlDataReader`s are always forward-only and read-only, so you can't go back to a row that you've already read, and you can't modify any of the data (we'll look at a way to get around these problems shortly).

When the reader is first opened, it points to the beginning of the data (BOF), so you need to move to the first row before any data can be accessed. You do this by calling the `Read` method, which moves the reader to the next row and returns `false` if there are no more rows or `true` if there is still more data to be read. To iterate through the whole of the data, you therefore need to call `Read` until it returns `false`:

```
   while (rdr.Read())
   {
      // Access current row here
```

```
    }
    rdr.Close();
}
```

Once you have the current row, there are a number of ways you can access the values in particular columns:

```
rdr.GetValue(int index);
rdr[int index];
rdr[string column_name];
rdr.Get<.NET type>(int index);
rdr.Get<SQL type>(int index);
```

Here, `index` is the number of the column from which you want to retrieve the data. The first three of these return a generic `object`, so the value needs to be cast to the specific type. The actual lines of code for an `int` would be as follows:

```
int i = (int)rdr.GetValue(0);
int i = (int)rdr[0];
int i = (int)rdr["AddressId"];
int i = rdr.GetInt32(0);
int i = rdr.GetSqlInt32(0);
```

Using an ordinal index rather than the column name is significantly faster; in a simple performance test repeated for string and integer columns, we found using the column names to be in the region of 20 percent slower than the other methods, as it needs to look up the column from its name. In some circumstances this may be a price worth paying, as using the column name guarantees you're accessing the right column, while using an ordinal index means you need to know the order in which the columns are retrieved, and if the SQL statement changes, this may break your code. This is, of course, particularly true if you use `SELECT *`, so it's worth stressing that this is usually a bad practice. In most circumstances, the best approach is to specify the column names in the `SELECT` statement and then use the ordinal indexer to access the column in a data reader, or (even better) use a constant or enumeration value to replace the ordinal. While this approach could also break if you change the T-SQL statement, it will be much easier to fix.

We found no significant difference between the other methods, although there was perhaps a very slight advantage to using the typed methods and a slight advantage to using `GetSqlInt32` instead of `GetInt32`. However, these differences were within the margin of error and may not apply to the released product. Moreover, we found no corresponding advantage to calling `GetSqlString` over `GetString`. While our test simply dumped the data once it had retrieved it, your code will probably need to use the data after accessing it! Therefore, it makes sense to choose the method that returns the data type you need, rather than converting a `SqlString` to a `System.String` or vice versa.

Finally, you can also read all the data from the current row into an object array by calling the `GetValues` method. This takes as a parameter the object array you want to populate and returns the number of elements in the array:

```
object[] rowData = new object[rdr.FieldCount];
int i = rdr.GetValues(rowData);
```

Note that you need to initialize the array with the correct number of elements before passing it into the `GetValues` method; you can do this by calling the `SqlDataReader`'s `FieldCount` property to ascertain how many columns are in the row. If the array is initialized with the wrong size, an exception will be thrown when `GetValues` is called. Default values (such as zero or an empty string) will be returned for any null columns, so these *will* be included in the array.

Handling Parameters

When you create a `SqlCommand` object, you'll very often need to execute the command it represents with different criteria. For example, you may need to retrieve multiple rows based on a selection of IDs. Therefore, you need a way to pass in parameters to your commands.

To allow this, the `SqlCommand` object has a `Parameters` collection, which contains a `SqlParameter` object for each parameter the command takes. You can indicate that a command requires parameters by using SQL-style variables in the command text, for example:

```
SqlCommand cmd = SqlContext.GetCommand();
cmd.CommandText = @"SELECT *
                    FROM HumanResources.EmployeePayHistory
                    WHERE EmployeeID = @id";
```

You can now create a parameter named `@id`, add it to the `Parameters` collection of the `SqlCommand` object, and set its value before executing the command, so that only the correct row will be returned from your query. To do this, you can use one of three methods of the `SqlParametersCollection` class: `Add`, `AddRange`, or `AddWithValue`. `Add` has several overloads that vary somewhat in how they're used. One overload lets you add a preconfigured `SqlParameter` object:

```
SqlParameter param = new SqlParameter("@id", SqlDbType.Int);
cmd.Parameters.Add(param);
```

Another overload takes the name of the parameter and its type (as one of the `SqlDbType` enumeration values):

```
SqlParameter param = cmd.Parameters.Add("@id", SqlDbType.Int);
```

The `AddWithValue` method provides a quick way of adding a parameter to the collection at the same time as the parameter itself:

```
SqlParameter param = cmd.Parameters.AddWithValue("@id", 17);
```

Finally, you can use the `AddRange` method to add an array of `SqlParameter` objects in a single method call:

```
SqlParameter param1 = new SqlParameter("@id", 23);
SqlParameter param2 = new SqlParameter("@name", "Bob");
cmd.Parameters.AddRange(new SqlParameter[] { param1, param2 });
```

Which of these methods you choose depends on whether the same command is going to be issued more than once. If it is, it makes sense to call `Add` to configure the parameters first, prepare the command, and then set the values for each call. If the command is going to be executed only once, it makes sense to add the parameter and value in one go. And, of course, it makes no sense to supply a value for an output parameter or return value.

Representing Row Metadata

In the next section, we'll look at how SQL Server allows us to work with individual rows of data to construct a resultset on the fly and return it to the user. To do this, we need a way of representing the metadata for each row, to which end we now have the `SqlMetaData` class. A `SqlMetaData` instance contains the metadata for a single column in a resultset, so we can represent the metadata for a row using an array of `SqlMetaData` objects. `SqlMetaData` objects can also be used to define parameters passed into stored procedures or UDFs.

The `SqlMetaData` class has 14 public properties, as shown in Table 3-2.

Table 3-2. *Public Properties of the SqlMetaData Class*

Property	Data Type	Description
`CompareOptions`	`System.Data.SqlTypes.SqlCompareOptions`	Indicates how data in the column/parameter is to be compared to other data (e.g., whether whitespace is to be ignored)
`DbType`	`System.Data.DbType`	Returns the type of the column/parameter as a `DbType` enum value
`LocaleID`	`long`	Returns the ID for the locale used for data in the column/parameter
`Max`	`long`	Static property that returns the maximum length for `text`, `ntext`, and `image` data
`MaxLength`	`long`	Returns the maximum length for the column/parameter represented by this `SqlMetaData` instance
`Name`	`string`	Returns the name of the column/parameter
`Precision`	`byte`	Returns the precision for the column/parameter
`Scale`	`byte`	Returns the scale for the column/parameter
`SqlDbType`	`System.Data.SqlDbType`	Returns the type of the column/parameter as a `SqlDbType` enum value
`Type`	`System.Type`	Returns the type of the column/parameter as a `System.Type` instance
`TypeName`	`string`	For a user-defined type column/parameter, returns the three-part name of the type
`XmlSchemaCollectionDatabase`	`string`	Returns the name of the database that contains the schema collection for an XML type

Continued

Table 3-2. *Continued*

Property	Data Type	Description
XmlSchemaCollectionName	string	Returns the name of the schema collection for an XML type
XmlSchemaCollectionOwningSchema	string	For an XML column/parameter, returns the name of the SQL Server relational schema to which the schema collection for the column belongs

The SqlMetaData class also has two public methods, which are listed in Table 3-3.

Table 3-3. *Public Methods of the SqlMetaData Class*

Method	Return Type	Parameter(s)	Description
Adjust	*<DataType>*	*<DataType>*	This method validates the value passed in as a parameter against the metadata represented by this instance and adjusts it if it isn't valid. There are many overloads for specific .NET and SQL data types, and the return type will be the same as the type of the parameter.
InferFromValue	SqlMetaData	object, string	This static method creates a new SqlMetaData instance based on the value of the object and the name passed in as parameters.

There are a number of ways to create a new SqlMetaData object, depending on the data type it represents. The easiest way is to call the InferFromValue method:

```
SqlMetaData md = SqlMetaData.InferFromValue(somevariable, "Column Name");
```

Alternatively, you can create a new instance using the constructor. The SqlMetaData constructor has seven overloads:

- The simplest takes the name as a string and the type as a SqlDbType enumeration value:

  ```
  SqlMetaData md = new SqlMetaData("Column Name", SqlDbType.Int32);
  ```

- For string and binary types, you can also specify the maximum length:

  ```
  SqlMetaData md = new SqlMetaData("Column Name", SqlDbType.NVarChar, 256);
  ```

- Additionally, you can include the locale ID and the compare options for string types:

  ```
  SqlMetaData md = new SqlMetaData("Column Name",
                                   SqlDbType.NVarChar, 256, 1033,
                                   SqlCompareOptions.IgnoreCase);
  ```

- For `Decimal` types, you need to specify the precision and scale:

```
SqlMetaData md = new SqlMetaData("Column Name", SqlDbType.Decimal, 8, 4);
```

- If the type for this `SqlMetaData` instance is a user-defined type (UDT), you need to provide the type as a `System.Type` instance:

```
SqlMetaData md = new SqlMetaData("Column Name", SqlDbType.UDT,
                                 myUdt.GetType());
```

  where `myUdt` is an instance of the UDT. In this case, the second parameter must be set to `SqlDbType.UDT`.

- If the instance represents a strongly typed XML column or parameter, you need to supply the name of the database that contains the schema collection, the name of the owning schema, and the name of the schema collection:

```
SqlMetaData md = new SqlMetaData("Column Name", SqlDbType.Xml,
                                 "AdventureWorks", "HumanResources",
                                 "HRResumeSchemaCollection");
```

- The final constructor takes the name of the column, its type as a `SqlType` value, the maximum allowed length, its precision and scale, its locale ID, its compare options, and the type of a UDT as a `System.Type`. This constructor is presumably intended for cases where a UDT column requires string and/or floating-point options to be set.

You'll see the `SqlMetaData` class in action when we look at creating rows on the fly later in the chapter.

Working with Single Rows

As well as classes for communicating with SQL Server and for building specific types of CLR modules, the `Microsoft.SqlServer.Server` namespace also includes a new class for working with an individual row in a resultset. The `SqlDataRecord` class is basically very similar to a data reader that can only contain one row, but with one important difference: a `SqlDataRecord` can be updated as well as read.

Overview of SqlDataRecord

The public properties of the `SqlDataRecord` class are shown in Table 3-4.

Table 3-4. *Public Properties of the SqlDataRecord Class*

Property	Type	Description
`FieldCount`	`int`	Returns the number of columns in the row.
`Item`	`object`	Returns the value of the specified column in the row. This property takes an `int` or `string` parameter and is accessed as the indexer in C#.

The `SqlDataRecord` class also has a number of public methods, as shown in Table 3-5.

Table 3-5. *Public Methods of the SqlDataRecord Class*

Method	Return Type	Parameter(s)	Description
Get<*DataType*>	<*DataType*>	int	Returns the value of the specified column as the given data type
GetDataTypeName	string	int	Returns the name of the data type of the specified column
GetFieldType	System.Type	int	Returns the data type of the specified column as a System.Type
GetName	string	int	Returns the name of the specified column
Get<*SqlType*>	<*SqlType*>	int	Returns the value of the specified column as a SQL native type
GetSqlFieldType	System.Type	int	Returns the SQL type of the specified column as a System.Type
GetSqlMetaData	Microsoft.SqlServer.Server.SqlMetaData	int	Returns a SqlMetaData object describing the specified column
GetSqlValue	object	int	Returns the value of the specified column as a SQL native type cast as object
GetSqlValues	int	object[]	Populates the supplied object array with the values in the row (as native SQL types) and returns the number of elements in the array
GetValue	object	int	Returns the value of the specified column as a standard .NET type cast as object
GetValues	int	object[]	Populates the supplied object array with the values in the row (as standard .NET types) and returns the number of elements in the array
IsDbNull	bool	int	Indicates whether or not the value in the specified column is null
Set<*DataType*>	void	int, <*DataType*>	Sets the value of the column specified in the first parameter to the value of the second parameter, which must be castable to the same type as the type indicated by the method name
Set<*SqlType*>	void	int, <*SqlType*>	Same as the previous set of methods, but uses native SQL types instead of standard .NET types
SetValue	void	int, object	Sets the value of the column specified in the first parameter to the value of the second parameter, which is passed in as an object
SetValues	int	object[]	Sets the values of the row to those contained in the supplied object array and returns the number of column values set

As Table 3-5 implies, reading a field from a row works in exactly the same way as reading a value from a SqlDataReader: you can use an integer or string indexer, or you can call one of the Get methods. There are three ways to set a value for a column:

```
row.SetValue(int index, object value);
row.Set<.NET type>(int index, value);
row.Set<SQL type>(int index, value);
```

for example:

```
// Set string value using SetValue
row.SetValue(0, "10 Downing Street");

// Set string value using SetString
row.SetString(1, "Westminster");

// Set string value using SetSqlString
SqlString city = new SqlString("London");
row.SetSqlString(2, city);
```

Because all these methods take an integer index for the column rather than a string name, there's little difference in performance, although the typed methods performed slightly faster than SetValue. Again, your choice to use a method that takes a .NET type or one that takes a SQL type will depend on context: there's no point converting a value to a SqlString if you already have it as a System.String (our previous example where we created a new SqlString from a string literal was obviously contrived).

There's also a SetValues method that parallels GetValues:

```
string[] rowData = new string[] { "10 Downing Street",
                                  "Westminster",
                                  "London" };
// Set all three columns in one go
row.SetValues(rowData);
```

Of course, this relies on the elements in the array being in the same order as the columns in the SqlDataRecord, and on each being of the appropriate type. In this case, we've made it easy for ourselves, because all the columns are string columns, so we can define the array as of type string[]; normally we'd need the array to be of type object[], with elements of different types.

Creating Rows on the Fly

The SqlDataRecord class allows you to create new rows programmatically and send them as a resultset to SQL Server. This is where the SqlMetaData class comes in. The SqlDataRecord's constructor takes an array of SqlMetaData objects, each element of which represents a single column in the row.

The following code creates a SqlDataRow containing two columns, of type int and nvarchar, and sets their values to 0 and "SQL Server 2005 Assemblies", respectively:

```
SqlMetaData idMetadata = new SqlMetaData("id", SqlDbType.Int);
SqlMetaData titleMetadata = new SqlMetaData("title",
          SqlDbType.NVarChar, 255);
SqlMetaData[] rowMetadata = new SqlMetaData[] { idMetadata,
          titleMetadata };
SqlDataRecord record = new SqlDataRecord(rowMetadata);
record.SetInt32(0, 0);
record.SetString(1, "SQL Server 2005 Assemblies");
```

First you create two SqlMetaData objects to represent id and title columns of type int and nvarchar, respectively (notice that you need to specify a maximum length for columns of type nvarchar or varchar). You then create an array called rowMetadata consisting of these two objects. Once you have this, you can use it to create a new SqlDataRecord object. Finally, you set values for the row data.

Communicating with the Caller

Once you have the data from SQL Server and manipulated it, you need to return the results to the caller, or at least send the user a message saying that your work is done. You do this using a SqlPipe object, which provides a direct channel to SQL Server's output stream. Any messages or query results sent to the pipe will be displayed directly to the user in Query Analyzer or sent to the calling application.

Overview of SqlPipe

The SqlPipe class has just one property, as shown in Table 3-6.

Table 3-6. *Public Property of the SqlPipe Class*

Property	Type	Description
SendingResults	bool	Indicates whether the SqlPipe is currently being used to send rows back to the caller

The SqlPipe class also has five methods (excluding inherited methods), which are listed in Table 3-7.

Table 3-7. *Public Methods of the SqlPipe Class*

Method	Return Type	Parameters	Description
ExecuteAndSend	void	SqlCommand	Executes the specified SqlCommand and sends the results directly back to the caller
Send	void	System.Data.SqlClient.SqlDataReader Microsoft.SqlServer.Server.SqlDataRecord string	Sends the supplied string message, SqlDataReader, or SqlDataRecord back to the caller

Method	Return Type	Parameters	Description
SendResultsEnd	void		Signals that you've finished sending rows that form part of a resultset
SendResultsRow	void	Microsoft.SqlServer.Server.SqlDataRecord	Sends the specified row to the caller as part of a resultset
SendResultsStart	void	Microsoft.SqlServer.Server.SqlDataRecord	Signals that you want to send a number of SqlDataRecords to the caller as part of a resultset, and sends the first row

We won't spend too long covering the first two of these, as they're syntactically very simple. The Send method is overloaded and can take a plain string, a SqlDataReader, or a SqlDataRecord object as parameter. For example, to send all contact details back to the caller, you could use this code:

```
using (SqlConnection cn = new SqlConnection("context connection = true")
{
   cn.Open();
   string sql = "SELECT * FROM Person.Contact";
   SqlCommand cmd = new SqlCommand(sql, cn);
   SqlDataReader reader = cmd.ExecuteReader();
   SqlPipe pipe = SqlContext.Pipe;
   pipe.Send(reader);
}
```

Alternatively, you could call the SqlPipe.ExecuteAndSend method, which will send the results of a command straight back to the caller, bypassing your CLR module entirely (and thus saving you a line of code):

```
using (SqlConnection cn = new SqlConnection("context connection = true")
{
   cn.Open();
   string sql = "SELECT * FROM Person.Contact";
   SqlCommand cmd = new SqlCommand(sql, cn);
   SqlPipe pipe = SqlContext.Pipe;
   pipe.ExecuteAndSend(cmd);
}
```

Sending a Dynamic Rowset Back to the Caller

The last three methods of the SqlPipe class allow you to send a set of SqlDataRecord objects back to the caller, one row at a time. This means you can construct rows on the fly, as done a couple of pages back, and return them to the caller as a single resultset. You can't simply send the rows using the SqlPipe.Send method, as this will send a number of separate resultsets to SQL Server, each containing one row, rather than a single resultset containing multiple rows. To send individual rows as a single resultset, you need to tell SQL Server when you send the first row that there are more rows to come in the same resultset, and you also need to tell it

when you're done and the resultset is finished. To do this, you call `SendResultsStart` to send the first row, `SendResultsRow` for all subsequent rows, and finally `SendResultsEnd` when you're finished.

Let's look at a simple example to see how this works. This example reads the details of employees who haven't had a pay rise for two years from the `HumanResources.EmployeePayHistory` table in the AdventureWorks database, including each employee's ID, the last date when that employee received a pay raise, and a computed column indicating when the employee's pay raise should be due (two years after the last one). For each row in the `SqlDataReader`, we create a new row for our dynamic resultset, checking the due date value for each row, and if it's over six months ago, we make a pay raise for that employee a priority and set it for a month's time; otherwise, we can defer it for a little, and add seven months to when it was due. There might not be a lot of logic behind this, but it does demonstrate the techniques involved!

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

namespace Apress.SqlAssemblies.Chapter03
{
   public class SendResultsTest
   {
      // Define read-only fields to refer to the fields' ordinal indexes by
      readonly static int FIELD_ID = 0;
      readonly static int FIELD_LASTCHANGE = 1;
      readonly static int FIELD_DUECHANGE = 2;

      public static void GetPayRiseDates()
      {
         // Open the context connection
         using (SqlConnection cn = new SqlConnection("context connection=true"))
         {
            cn.Open();

            // Get a new SqlDataReader with the details of the
            // employees who are due a pay raise
            string sql = @"SELECT EmployeeID,
MAX(RateChangeDate) AS LastRateChange,
DATEADD(year, 2, MAX(RateChangeDate)) AS DueRateChange
FROM HumanResources.EmployeePayHistory
GROUP BY EmployeeID
HAVING MAX(RateChangeDate) < DATEADD(year, -2, GETDATE())";
            SqlCommand cmd = new SqlCommand(sql, cn);
            SqlDataReader reader = cmd.ExecuteReader();
```

```
// Get the SqlPipe
SqlPipe pipe = SqlContext.Pipe;

// Create the SqlMetaData objects for the rowset
SqlMetaData idMeta = new SqlMetaData("Id", SqlDbType.Int);
SqlMetaData lastRaiseMeta = new SqlMetaData("Last Rate Change",
                                            SqlDbType.DateTime);
SqlMetaData dueRaiseMeta = new SqlMetaData("Due Rate Change",
                                           SqlDbType.DateTime);
SqlMetaData[] rowMetaData = new SqlMetaData[] { idMeta, lastRaiseMeta,
                                                dueRaiseMeta };

// Keep track of whether or not it's the first row
bool firstRow = true;

// Iterate through the rows, update if necessary,
// and send them back to the caller
while (reader.Read())
{
   // Create a new SqlDataRecord for each row
   SqlDataRecord row = new SqlDataRecord(rowMetaData);

   // Add the ID and Last Rate Change values to the row
   row.SetInt32(FIELD_ID, (int)reader[FIELD_ID]);
   row.SetDateTime(FIELD_LASTCHANGE,
                   (DateTime)reader[FIELD_LASTCHANGE]);

   // Store the change due date in a local variable
   DateTime dueDate = (DateTime)reader[FIELD_DUECHANGE];

   // If it's over six months overdue, set pay raise for
   // a month's time; otherwise, put it back seven months
   if (dueDate < DateTime.Now.AddMonths(-6))
      row.SetDateTime(FIELD_DUECHANGE, DateTime.Now.AddMonths(1));
   else
      row.SetDateTime(FIELD_DUECHANGE, dueDate.AddMonths(7));

   // If it's the first row, we need to call
   // SendResultsStart; otherwise, we call SendResultsRow
   if (firstRow == true)
   {
      pipe.SendResultsStart(row);
      firstRow = false;
   }
   else
```

```
            {
               pipe.SendResultsRow(row);
            }
         }

         // Close the SqlDataReader once we've finished iterating through it
         reader.Close();

         // Call SendResultsEnd after the loop has finished, to
         // say we're done
         pipe.SendResultsEnd();
      }
   }
}
```

Let's run quickly through this code to clarify what's going on. First, we open the context connection and retrieve the details for all employees who haven't received a pay raise for two years into a `SqlDataReader`. Next, we get the `SqlPipe` object for the current context by calling `SqlContext.Pipe`. Then we create the `SqlMetaData` objects that define the name and data type for each column in our resultset, and add these to an array that contains all the metadata for each row. Because we need to call a different method for the first row we send back through the `SqlPipe`, we create a Boolean local variable with its value initialized to `true` to indicate that we're on the first row.

Now we're ready to iterate through each row in the data reader, creating a new `SqlDataRecord` object for each row and adding to it the employee ID and the date the employee last received a pay raise. Before adding the due date, we modify it as we think fit. Then we send the row back to the caller using the appropriate method of the `SqlPipe` object; if this is the first row, our `firstRow` Boolean variable will be `true`, so we know we have to call `SendResultsStart`, instead of `SendResultsRow`, which we'll use for all subsequent iterations. Once the loop has finished, we just need to close the data reader, call `SendResultsEnd` to indicate that our resultset is complete, and we're finished.

We compile this in the usual way:

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

then register it as a stored procedure with SQL Server, and execute the procedure:

```
CREATE ASSEMBLY SendResultsTest
FROM 'C:\Apress\SqlAssemblies\Chapter03\SendResultsTest.dll'
WITH PERMISSION_SET = SAFE;
GO

CREATE PROCEDURE uspGetPayRiseDates
AS
EXTERNAL NAME SendResultsTest.[Apress.SqlAssemblies.Chapter03.SendResultsTest].
GetPayRiseDates;
```

```
GO

EXEC uspGetPayRiseDates
```

The results can be seen in Figure 3-1.

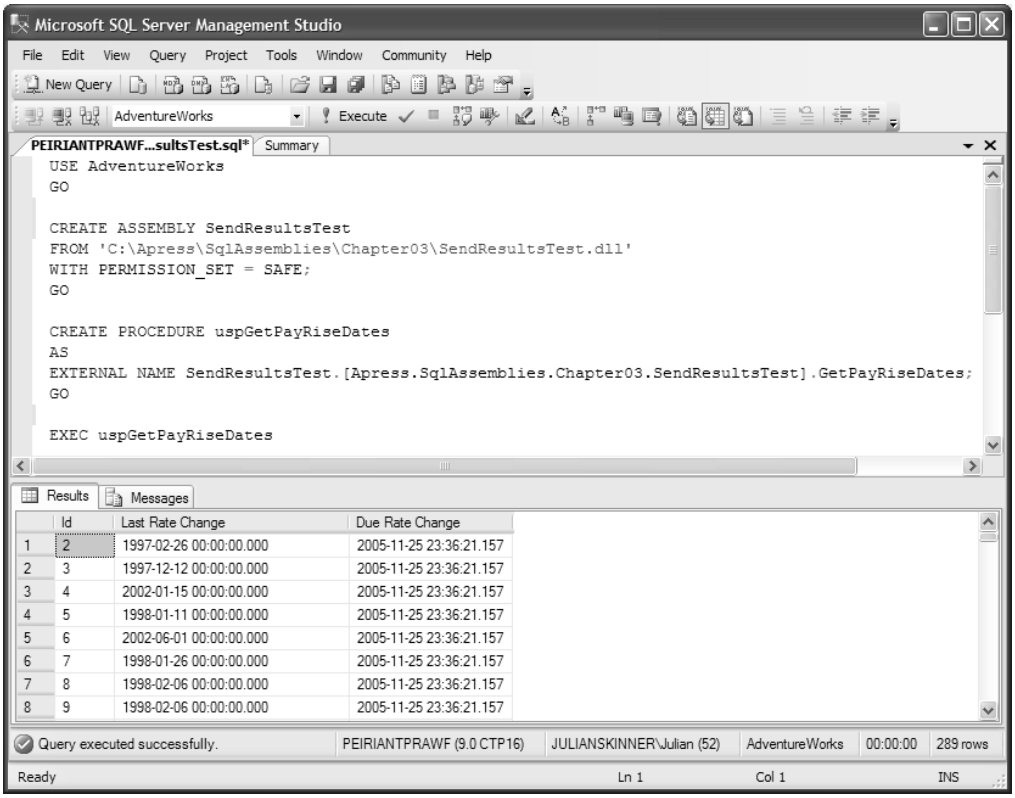

Figure 3-1. *Results of the SendResultsTest example*

CLR Triggers

The last class in the `Microsoft.SqlServer.Server` namespace is the `SqlTriggerContext` class, which provides information about the context in which a CLR trigger is running. We can gain access to this object from the `SqlContext`, similarly to the way we access the `SqlPipe` object:

```
SqlTriggerContext tc = SqlContext.TriggerContext;
```

We won't look at this in detail here, as we'll cover CLR triggers fully in Chapter 8, but for the sake of completeness, we will list its properties and methods. Table 3-8 lists its public properties.

Table 3-8. *Public Properties of the SqlTriggerContext Class*

Property	Data Type	Description
`ColumnCount`	`int`	Returns the number of columns in the table bound to the trigger.
`EventData`	`System.Data.SqlTypes.SqlXml`	A native SQL XML object that contains information about the event that caused the trigger to execute, including the time of the event, the system process ID of the connection that caused the event, and the type of event. Depending on the event type, it may also include additional information, such as the command that caused the event and the user who executed the command.
`TriggerAction`	`Microsoft.SqlServer.Server.TriggerAction`	A `TriggerAction` enumeration value that indicates the type of command that caused the trigger to fire. With the new ability to write DDL triggers, there are 76 of these values all told, so we won't list them all here. Please see Chapter 8 for more details.

Table 3-9 gives the details of the single public method of the `SqlTriggerContext`.

Table 3-9. *Public Method of the SqlTriggerContext Class*

Method	Return Type	Parameter	Description
`IsUpdatedColumn`	`bool`	`int`	Indicates whether the column with the supplied ordinal index was affected by an `INSERT` or `UPDATE` operation

Transactions

The latest version of the .NET Framework contains a whole new transaction framework in the shape of the `System.Transactions` namespace. This framework is integrated both with ADO.NET and with the SQL Server CLR. This isn't a topic specific to SQL assemblies, and the namespace is too large to cover in full, but one feature in particular needs to be highlighted: the ability to enroll a block of code implicitly in a distributed transaction using the `TransactionScope` class.

The transaction scope begins when the `TransactionScope` class is instantiated, and ends when the class is disposed of. If no errors occur, the transaction will be committed implicitly; otherwise, it will be rolled back when the scope ends. Because the scope ends when the `Dispose` method is called on the object, it's vital that this method is called in all cases. For that reason, Microsoft strongly recommends instantiating a `TransactionScope` object in a `using` statement so that `Dispose` will be called even if an exception occurs:

```
using (TransactionScope tx = new TransactionScope())
{
   // Code here is within the transaction scope
}
```

```
// Commit the transaction
tx.Complete();
```

The alternative is to place all code in the transaction scope within a `try` block, and then to call `Dispose` on it in the associated `finally` block.

The really neat trick, however, is that if you open a second connection within a transaction scope, the transaction will be promoted to a full distributed transaction, for example:

```
using (TransactionScope tx = new TransactionScope())
{
   using (SqlConnection cn1 = new SqlConnection("context connection = true"))
   {
      // This automatically enlists the connection in the transaction
      cn.Open();

      string connStr = "<connection string>"
      using (SqlConnection cn2 = new SqlConnection(connStr))
      {
         // This enlists cn2 in the transaction and thereby promotes it
         // to a distributed transaction
         cn2.Open();
      }
   }
}
// Commit the transaction
tx.Complete();
```

For this to work, Microsoft Distributed Transaction Coordinator (MSDTC) server must be running on the host machine.

Summary

Data access within a SQL assembly such as a CLR stored procedure or trigger is achieved by means of the .NET data provider for SQL Server (in the `System.Data.SqlClient` namespace), supplemented by several classes in the new `Microsoft.SqlServer.Server` namespace. In this chapter, we provided a very quick ADO.NET refresher and introduced the following new objects:

- `SqlContext`: Represents the current context and gives you access to the other objects in the provider
- `SqlMetaData`: Defines the metadata for a single column in a row of data
- `SqlDataRecord`: Provides you with access to a single row of data and allows you to create a resultset on the fly
- `SqlPipe`: Allows you to send data or messages directly to the SQL Server pipe
- `SqlTriggerContext`: Supplies information about the context in which a CLR trigger fired

We also looked very briefly at the new transaction framework in .NET 2.0 provided by the `System.Transactions` framework.

Now that we've introduced the main new classes, we can go on to present real examples of building SQL assemblies. This chapter has of necessity had more of a reference nature than the other chapters in the book, but from now on the emphasis will be firmly on practical examples. In the next chapter, you'll see the objects we presented in this chapter in action when we turn the focus to creating CLR stored procedures.