Professional SQL Server™ 2005 Programming

Chapter 14: Nothing But NET!

ISBN-10: 0-7645-8434-0
ISBN-13: 978-0-7645-8434-3

14

Nothing But NET!

In terms of where to locate a subject in my books, this subject has to be the hardest one to place I've ever had to deal with. At issue is how fundamental the addition of .NET assemblies is to SQL Server programming in the SQL Server 2005 era versus how much it is shared between the various programming topics and how much of it happens in its own world. Oh well, obviously I made a choice, and that choice was to hold the introduction of major .NET elements until we had all the major SQL Server–specific programming elements covered. So, that done, here we go.

.NET and all things related to it were just on the horizon when SQL Server 2000 came out. It's been a long wait since early 2000 (yes, even before SQL Server 2000 was out) when I first heard that T-SQL was finally going to accept code from non T-SQL languages. The story just got better and better as we heard that complex user-defined data types would be supported, and T-SQL itself would become a .NET language with associated error handling. And so it is here, and the days of the old claustrophobic TSQL are gone, and we have a wide world of possibilities available to us.

In this chapter, we're going to take a look at some of the major elements that .NET has brought to SQL Server 2005. We'll see such things utilizing .NET as:

- Creating basic assemblies — including non T-SQL based stored procedures, functions, and triggers
- Defining aggregate functions (something T-SQL user defined functions can't do)
- Complex data types
- External calls (and with it, some security considerations)

.NET is something of a wide-ranging topic that will delve into many different areas we've already touched on in this book and take them even farther, so, with that said, let's get going!

Note that several of the examples in this chapter utilize the existing Microsoft Sample set. You must install the sample scripts during SQL Server installation or download the SQL Server .NET development SDK to access these samples. In addition, there is a significant reliance on Visual Studio .NET (2005 is used in the examples).

Assemblies 101

All the new .NET functionality is surrounded by this new (to SQL Server anyway) term *assembly*. So, a reasonable question might be: "What exactly is an assembly?" An assembly is a DLL that has been created using managed code (what .NET language does not matter). The assembly may have been built using Visual Studio .NET or some other development environment, but the .NET Framework SDK also provides a command-line compiler for those of you who do not have Visual Studio available.

Not all custom attributes or .NET Framework APIs are legal for assemblies used in SQL Server. You can consult Books Online for a full list, but, in general, anything that supports windowing is not allowed, nor is anything marked UNSAFE, unless your assembly is to be granted access at an UNSAFE level.

Compiling an Assembly

Use of .NET assemblies requires that you enable the Common Language Runtime (CLR) in SQL Server — which is disabled by default. You can enable the CLR by executing the following in the Management Studio:

```
sp_configure 'clr enabled', 1
GO

RECONFIGURE
```

There really isn't that much to this beyond compiling a normal DLL. The real key points to compiling a DLL that is going to be utilized as a SQL Server .NET assembly are:

- You cannot reference any other assemblies that include functions related to windowing (dialogs, etc.).
- How the assembly is marked (safe, external access, unsafe) will make a large difference to whether or not the assembly is allowed to execute any functions.

From there, most things are not all that different from any other DLL you might create to make a set of classes available. You can either compile the project using Visual Studio (if you have it), or you can use the compiler that is included in the .NET SDK.

Let's go ahead and work through a relatively simple example for an assembly we'll use as a stored procedure example a little later in the chapter.

Create a new SQL Server project in Visual Studio using C# (you can translate this to VB if you wish) called `ExampleProc`. You'll find the SQL Server project type under the "Database" project group. When it comes up, cancel out of any database instance dialogs you get, and choose Class Library project in Visual Studio.

> **The actual project type you start with does not really matter other than what references it starts with. While this example suggests starting with a SQL Server project, you could just as easily start with a simple class project.**

Now add a new stored procedure by right-clicking the project and selecting Add➪Stored Procedure... as shown in Figure 14-1:

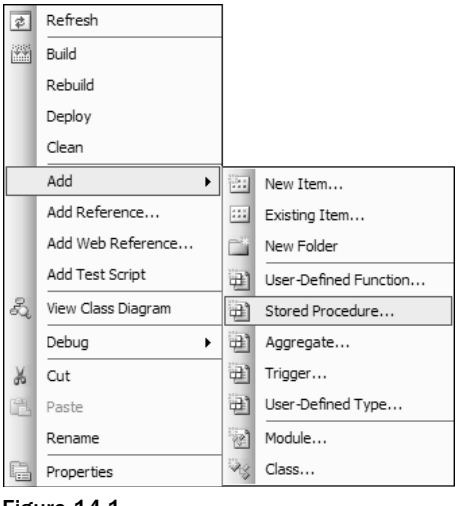

Figure 14-1

In this new stored proc, we need set a few references — some of them Visual Studio will have already done for you:

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

And then we're ready to get down to writing some real code. Code you want to put in a .NET assembly is implemented through a public class. I've chosen to call my class `StoredProcedures`, but, really, I could have called it most anything. I'm then ready to add my method declaration:

```
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void ExampleSP(out int outval)
    {
```

The method has, like the class, been declared as `public`. I can also have private classes if I so choose (they would, of course, need to be supporting methods, as they would not be exposed externally). The `void` indicates that I do not intend to supply a return value (when we run it in SQL Server, it will always return the default of zero). I could, however, declare it as returning type `int` and supply a value as appropriate in my code (most likely zero for no error, and non-zero if there was an error).

Notice also the `Microsoft.SqlServer.Server.SqlProcedure` directive. This is completely optional and is utilized by Visual Studio's deployment tool to know that the following method is a stored procedure. I left it in there primarily to show you that it is there (we're going to manually deploy the proc rather than use the deployment functionality of Visual Studio).

From there, we're ready to grab a reference to a connection. Pay particular attention to this one, as it is different from what you will see in typical .NET database connections. Everything about it is the same as a typical connection, except for the connect string — with that, we are using a special syntax that indicates that we want to utilize the same connection that called the stored procedure. The beauty of this is that we can assume a login context and do not need to explicitly provide a server or even a username.

```
        // This causes the connection to use the existing connection context
        // that the stored procedure is operating in. We could also create a
        // completely new connection to fetch data from external sources.
        using (SqlConnection cn = new SqlConnection("context connection=true"))
        {
            cn.Open();
```

So, now we have a connection. Technically, we didn't really open a new connection (remember, we're utilizing the one that was already opened to call this stored procedure). Instead, we're really creating a new reference to the existing connection.

We're now ready to create a command object. There really isn't anything mysterious about this at all — it is created using a fairly typical syntax (indeed, you can create the command object in any of the typical ways you would if you were doing this from a typical .NET client). I've chosen to define the `CommandText` and connection properties as part of my object declaration.

```
            // set up a simple command that is going to return two columns.
            SqlCommand cmd = new SqlCommand("SELECT @@SERVERNAME, @@SPID", cn);
```

And then we're ready to execute the command. This is one of the spots I'm utilizing something assembly specific, as I am not only executing the command, but also I'm explicitly saying to go ahead and send it to the client.

> **Unlike a T-SQL based stored procedure, queries you execute are not defaulted as going to the client. Instead, the assumption is that you want to work with the result set locally. The result is that you need to explicitly issue a command to send anything out to the client.**

The object that does this is the `.Pipe` object within the `SqlContext`.

```
            // The following actually sends the row for the select.
            // It could have been multiple rows, and that would be fine too.
            SqlContext.Pipe.ExecuteAndSend(cmd);
```

Last, but not least, I'm populating my output variable. In this procedure, I haven't really done anything special with it, but I'm tossing something into it just so we can see that it really does work.

```
            // Set the output value to something. It could have been anything
            // including some form of computed value, but we're just showing
            // that we can output some value for now.
            outval = 12345;
        }
    }
};
```

Now simply build your project, and you have your first assembly ready to be uploaded to your SQL Server. Later, we'll take a look at how to define the assembly for use as a stored procedure.

Uploading Your Assembly to SQL Server

That's right — I used the term *upload*. When you "create" an assembly in SQL Server, you're both creating a copy of the DLL within SQL Server as well as something of a handle that defines the assembly and the permissions associated with it.

```
CREATE ASSEMBLY <assembly name>
[ AUTHORIZATION <owner name> ]
FROM { <client assembly specifier> | <assembly bits> [ ,...n ] }
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]
[ ; ]
```

The `CREATE` portion of things adheres to the standard `CREATE <object type> <object name>` notion that we've seen throughout SQL. From there, we have a few different things to digest:

Option	Description
AUTHORIZATION	The authorization is the name of the user this assembly will be considered to belong to. If this parameter is not supplied, then the current user is assumed to be the owner. You can use this to alias to a user with appropriate network access to execute any file actions defined by the assembly.
FROM	The fully qualified path to the physical DLL file. This can be a local file or a UNC path. You can, if you so choose, provide the actual byte sequence to build the file right on the line in the place of a file (I have to admit I've never tried that one).
WITH PERMISSION_SET	You have three options for this. SAFE is the default and implies that the object is not utilizing anything that requires access outside of the SQL Server process (no file access, no external database access). EXTERNAL_ACCESS indicates that your assembly requires access outside of the SQL Server process (to files in the operating system or UNC path, or perhaps an external ODBC/OLEDB connection). UNSAFE implies that your assembly gives your assembly free access to the SQL Server memory space without regard to the CLR managed code facilities. This means your assembly has the potential to destabilize your SQL Server through improper access.

So, with all this in mind, we're ready to upload our assembly:

```
USE AdventureWorks

CREATE ASSEMBLY ExampleProc
  FROM '<solution path>\ExampleProc\bin\Debug\ExampleProc.dll'
```

Assuming that you have the path to your compiled DLL correct, you shouldn't see any messages except for the typical "Command(s) completed successfully" message, and, with that, you are ready to create the SQL Server stored procedure that will reference this assembly.

Creating Your Assembly-Based Stored Procedure

All right then; all the tough stuff is done (if you're looking for how to actually create the assembly that is the code for the stored proc, take a look back two sections). We have a compiled assembly, and we have uploaded it to SQL Server — it's time to put it to use.

To do this, we use the same CREATE PROCEDURE command we learned back in Chapter 11. The difference is that, in the place of T-SQL code, we reference our assembly. For review, the syntax looks like this:

```
CREATE PROCEDURE|PROC <sproc name>
   [<parameter name> [<schema>.]<data type> [VARYING] [= <default value>] [OUT
[PUT]][,
    <parameter name> [<schema>.]<data type> [VARYING] [= <default value>]
[OUT[PUT]][,
    ...
    ...
      ]]
[WITH
   RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
[FOR REPLICATION]
AS
   <code> | EXTERNAL NAME <assembly name>.<assembly class>
```

Some of this we can ignore when doing assemblies. The key things are:

- We use the `EXTERNAL NAME` option instead of the `<code>` section we used in our main chapter on stored procedures. The `EXTERNAL NAME` is done in a format of

  <assembly name>.<class name>.<method name>

- We still need to define all parameters (in an order that matches the order our assembly method).

Now let's apply that to the assembly we created in the previous section:

```
CREATE PROC spCLRExample
    (
        @outval int = NULL OUTPUT
    )
AS EXTERNAL NAME ExampleProc.StoredProcedures.ExampleSP
```

It is not until this point that we actually have a stored procedure that utilizes our assembly. Notice that the stored procedure name is completely different from the name of the method that implements the stored procedure.

Now go ahead and make a test call to our new stored procedure:

.

```
DECLARE @OutVal int
EXEC spCLRExample @OutVal OUTPUT

SELECT @OutVal
```

We're declaring a holding variable to receive the results from our output variable. We then execute the procedure and select the value for our holding variable. When you check the results, however, you'll find not one result set — but two:

```
------------------- ------
NEWTON              52

(1 row(s) affected)


-----------
```

```
12345

(1 row(s) affected)
```

The first of these is the result set we sent down the `SqlContext.Pipe`. When we executed the `cmd` object, the results were directed down the pipe, and so the client received them. The second result set represents the `SELECT` of our `@OutVal` variable.

This is, of course, a pretty simplistic example, but realize the possibilities here. The connection could have been, assuming we were set to `EXTERNAL_ACCESS`, to any datasource. We could access files and even Web services. We can add in complex libraries to perform things like regular expressions (careful on performance considerations there).

We will look at adding some of these kinds of things in as we explore more types of assembly-based SQL programming.

Creating Scalar User-Defined Functions from Assemblies

Scalar functions are not much different from stored procedures. Indeed, for the most part, they have the very same differences that the T-SQL versions. Much as with stored procedures, we utilize the same core `CREATE` syntax that we used in the T-SQL user-defined functions (UDFs) we created back in Chapter 11.

```
CREATE FUNCTION [<schema name>.]<function name>
    ( [ <@parameter name> [AS] [<schema name>.]<scalar data type> [ = <default
value>]
      [ ,...n ] ] )
RETURNS {<scalar type>|TABLE [(<Table Definition>)]}
    [ WITH [ENCRYPTION]|[SCHEMABINDING]|
        [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
        CALLER|SELF|OWNER|<'user name'>} ]
]
[AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

There are one or two new things once you get inside of the .NET code. Of particular note is that there are some properties that you can set for your function. Among those, probably the most significant is that you must indicate if the function is deterministic (the default is nondeterministic). We'll see an example of this in use shortly.

For the example this time, start a new SQL Server project in Visual Studio, but instead of adding a stored procedure as we did in our original assembly example, add a user-defined function.

SQL Server starts you out with a simple template:

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

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString ExampleUDF()
    {
        // Put your code here
        return new SqlString("Hello");
    }
};
```

This is actually a workable template "as is." You could compile it and add it to SQL Server as an assembly, and it would work right out of the box (though just getting back the string "Hello" is probably not all that useful).

We'll replace that, but this time we're going to write something that is still amazingly simple. In the end, we'll see that, while simple, it is much more powerful than our stored procedure example.

In previous books, I have often lamented the issues with trying to validate e-mail fields in tables. E-mail, when you think about it, is a strongly typed notion, but one that SQL Server has only been able to perform minimal validation of. What we need are regular expressions.

We could approach this issue by writing a validation function and implementing it as a user-defined data type. This approach would have some validity, but has a problem — the rules for validating e-mails change on occasion (such as when new country codes or added, or when the `.biz` and `.info` top domains were added several years ago). Instead, we're going to implement simple regex functionality and then utilize a call to that function in a constraint.

We can do this with relatively minimal changes to the function template that SQL Server gave us. First, we can get rid of a few library declarations, since we won't be really working with SQL Server data to speak of, and add back two of our own. We wind up with just three `using` declarations:

```
using System;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;
```

We're then ready to implement out function with very few changes:

```
        [SqlFunction(IsDeterministic = true, IsPrecise = true)]
        public static bool RegExIsMatch(string pattern, string matchString)
        {
              Regex reg = new Regex(pattern.TrimEnd(null));
              return reg.Match(matchString.TrimEnd(null)).Success;
        }
```

Oh sure, we completely replaced the old function, but not by much. Indeed, we only have two more lines of code — and that's including the determinism declaration!

I'm not going to review it much here, but take a look back at Chapter 11 if you need to be reminded of how determinism works The key thing is that, given the same inputs, the function must always yield the same outputs.

Go ahead and compile this, and we're ready to upload the assembly:

```
USE AdventureWorks

CREATE ASSEMBLY ExampleUDF
FROM '<solution path>\ExampleUDF\bin\Debug\ExampleUDF.dll'
```

And then create the function reference:

```
CREATE FUNCTION fCLRExample
    (
        @Pattern nvarchar(max),
        @MatchString nvarchar(max)
    )
RETURNS BIT
AS EXTERNAL NAME ExampleUDF.UserDefinedFunctions.RegExIsMatch
```

Notice the use of the `nvarchar` type instead of `varchar`. The string data type is a Unicode data type, and our function data type declaration needs to match.

This done, we're ready to test things out a bit:

```
SELECT ContactID, FirstName, LastName, EmailAddress, Phone
FROM Person.Contact
WHERE dbo.fCLRExample('[a-zA-Z0-9_\-]+@([a-zA-Z0-9_\-]+\.)+(com|org|edu|mil|net)',
                      EmailAddress) = 1
```

If you have the default data, then this will actually return every row in the table since they all are adventure-works.com addresses. So, let's try a simple test to show what works versus what doesn't:

```
DECLARE @GoodTestMail varchar(100),
        @BadTestMail varchar(100)

SET @GoodTestMail = 'robv@professionalsql.com'
SET @BadTestMail = 'misc. text'

SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+@([a-zA-Z0-9_\-]+\.)+(com|org|edu|nz|au)',
                       @GoodTestMail) AS ShouldBe1
SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+@([a-zA-Z0-9_\-]+\.)+(com|org|edu|nz|au)',
                       @BadTestMail) AS ShouldBe0
```

For the sake of brevity, I have not built the full e-mail regex string here. It would need to include all of the valid country code top domans such as `au`, `ca`, `uk`, *and* `us`. *There are a couple hundred of these, so it wouldn't fit all that well. That said, the basic construct is just fine, you can tweak it to meet your particular needs.*

This gets us back what we would expect:

```
ShouldBe1
---------
1

(1 row(s) affected)

ShouldBe0
---------
0

(1 row(s) affected)
```

But let's not stop there. We have this nice function, let's apply it a little further by actually applying it as a constraint to the table.

```
ALTER TABLE Person.Contact
ADD CONSTRAINT ExampleFunction
CHECK (dbo.fCLRExample('[a-zA-Z0-9_\-]+@([a-zA-Z0-9_\-]+\.)+(com|org|edu|nz|au)',
                       EmailAddress) = 1)
```

Now we try to update a row to insert some bad data into our column, and it will be rejected:

```
UPDATE Person.Contact
SET EmailAddress = 'blah blah'
WHERE ContactID = 1
```

And SQL Server tells you the equivalent of "no way!":

```
Msg 547, Level 16, State 0, Line 1
The UPDATE statement conflicted with the CHECK constraint "ExampleFunction". The
conflict occurred in database "AdventureWorks", table "Person.Contact", column
'EmailAddress'.
The statement has been terminated.
```

Creating Table-Valued Functions

Functions are going to continue to be a focus of this chapter for a bit. Why? Well, functions have a few more twists to them than some of the other assembly uses.

In this section, we're going to focus in on table valued functions. They are among the more complex things we need to deal with in this chapter, but, as they are in the T-SQL version, they were also among the more powerful. The uses range far and wide. They can be as simple as special treatment of a column in something you could have otherwise done in a typical T-SQL function or can be as complex as a merge of data from several disparate and external datasources.

Go ahead and start another Visual Studio project called `ExampleTVF`, using the SQL Server project template — also add a new user-defined function. We're going to be demonstrating accessing the file system this time, so add the following references:

```
using System;
using System.IO;
using System.Collections;
using Microsoft.SqlServer.Server;
```

Before we get too much into the code, let's look ahead a bit at some of the things a table-valued function — or TVF — requires:

The entry function must implement the `IEnumerable` interface. This is a special, widely used, interface in .NET that essentially allows for the iteration over some form of row (be it in an array, collection, table, or whatever). As part of this concept, we must also define the `FillRowMethodName` property. The function specified in this special property will be implicitly called by SQL Server every time SQL Server has the need to move between rows. You will find that a good many developers call whatever function they implement this in `FillRow` — for me it will vary depending on the situation and whether I feel it warrants something more descriptive of what it's doing or not.

So, with those items described, let's look at the opening of our function. Our function is going to be providing a directory listing, but one based on information that must be retrieved from individual files. This means that we have to enumerate the directory to retrieve each file's information. Just to add to the flavor of things a bit, we will also support the notion of subdirectories — which means we have to understand the notion of directories within directories.

We'll start with our top-level function call. This accepts the search filter criteria, including the directory we are considering the root directory for our list, the filename criteria for the search, and a Boolean indicator of whether to include subdirectories or not.

```
public partial class UserDefinedFunctions
{

    [SqlFunction(FillRowMethodName = "FillRow")]

    public static IEnumerable DirectoryList(string sRootDir, string sWildCard, bool
bIncludeSubDirs)
    {
        // retrieve an array of directory entries. Where this an object of our own
making,
        // it would need to be one that supports IEnumerable, but since ArrayList
already
        // does that, we have nothing special to do here.
        ArrayList aFileArray = new ArrayList();
        DirectorySearch(sRootDir, sWildCard, bIncludeSubDirs, aFileArray);

        return aFileArray;
    }
```

This has done little other than establish an array that will hold our file list and call to internal functions to populate it. Next, we need to implement the function that is enumerating the directory list to get the files in each directory:

```
    private static void DirectorySearch(string directory, string sWildCard, bool
bIncludeSubDirs, ArrayList aFileArray)
    {
        GetFiles(directory, sWildCard, aFileArray);

        if (bIncludeSubDirs)
        {
              foreach (string d in Directory.GetDirectories(directory))
              {
                    DirectorySearch(d, sWildCard, bIncludeSubDirs, aFileArray);
              }
        }
    }
```

For each directory we file, we make a simple call of the `GetFiles` method (it is implemented in `System.IO`) and enumerate the results for the current directory. As we enumerate, we populate our array with the `FullName` and a `LastWriteTime` properties from the file:

```
    private static void GetFiles(string d, string sWildCard, ArrayList aFileArray)
    {
        foreach (string f in Directory.GetFiles(d, sWildCard))
        {
              FileInfo fi = new FileInfo(f);

              object[] column = new object[2];
              column[0] = fi.FullName;
              column[1] = fi.LastWriteTime;

              aFileArray.Add(column);
        }

    }
```

From there, we're ready to bring it all home by actually implementing our `FillRow` function, which does nothing more than serve as a conduit between our array and the outside world — managing the feed of data to one row at a time.

```
    private static void FillRow(Object obj, out string filename, out DateTime date)
    {
        object[] row = (object[])obj;

        filename = (string)row[0];
        date = (DateTime)row[1];
    }
};
```

With all that done, we should be ready to compile and upload our assembly. We use the same `CREATE ASSEMBLY` command we've used all chapter long, but there is a small change — we must declare the assembly as having the `EXTERNAL_ACCESS` permission set. One of two conditions that must be met in order to do this:

- The assembly is signed with a certificate (more on these in Chapter 22) that corresponds to a user with proper `EXTERNAL_ACCESS` rights.
- The database owner has `EXTERNAL_ACCESS` rights *and* the database has been marked as being `TRUSTWORTHY` in the database properties.

We're going to take the unsigned option, so we need to set the database to be marked as trustworthy:

```
ALTER DATABASE AdventureWorks
SET TRUSTWORTHY ON
```

And we're now ready to finish uploading our assembly with proper access:

```
USE AdventureWorks

CREATE ASSEMBLY fExampleTVF
FROM '<solution path>\ExampleTVF\bin\Debug\ExampleTVF.dll'
WITH PERMISSION_SET = EXTERNAL_ACCESS
```

The actual creation of the function reference that utilizes our assembly is not bad but is slightly trickier than the one for the simple scalar function. We must define the table that is to be returned in addition to the input parameters:

```
CREATE FUNCTION fTVFExample
    (
        @RootDir nvarchar(max),
        @WildCard nvarchar(max),
        @IncludeSubDirs bit
    )
RETURNS TABLE
    (
        FileName nvarchar(max),
        LastWriteTime nvarchar(max)
    )
AS EXTERNAL NAME ExampleTVF.UserDefinedFunctions. DirectoryList
```

And, with that, we're ready to test:

```
SELECT FileName, LastWriteTime
FROM dbo.fTVFExample('C:\', '*.sys', 0)
```

What you get back when you run this will vary a bit depending on what components and examples you have installed, but, in general, it should look something like:

```
FileName                                           LastWriteTime
-------------------------------------------------- -----------------------
C:\CONFIG.SYS                                      2006-04-01 00:21:43.470
C:\IO.SYS                                          2006-04-01 00:21:43.470
C:\MSDOS.SYS                                       2006-04-01 00:21:43.470
C:\pagefile.sys                                    2006-05-02 20:08:56.500

(4 row(s) affected)
```

We've now shown not only how we can do table valued functions but also how we can access external data — powerful stuff!

Creating Aggregate Functions

Now this one is going to be the one thing in this chapter that's really new. When we look at user-defined data types a little later, we'll see something with a bigger shift than some of the other constructs we've looked at here, but aggregate functions are something that you can't do any other way — the T-SQL version of a UDF does not allow for aggregation.

So, what am I talking about here? Well, examples are `SUM`, `AVG`, `MIN`, and `MAX`. These all look over a set of data and then return a value that is based on some analysis of the whole. It may be based on your entire result set or on some criteria defined in the `GROUP BY` clause.

Performing the analysis required to support your aggregate gets rather tricky. Unlike other functions, where everything can be contained in a single call to your procedure, aggregates require mixing activities your function does (the actual aggregation part) with activities SQL Server is doing at essentially the same time (organizing the groups for the `GROUP BY` for example). The result is something of staged calls to your assembly class. Your class can be called at any of four times and can support methods for each of these calls:

- `Init` — This supports the initialization of your function. Since you're aggregating, there's a good chance that you are setting some sort of accumulator or other holding value — this is the method where you would initialize variables that support the accumulation or holding value.
- `Accumulate` — This is called by SQL Server once for every row that is to be aggregated. How you choose to utilize the function is up to you, but presumably it will implement whatever accumulation logic you need to support your aggregate.
- `Merge` — SQL Server is a multithreaded application, and it may very well utilize multiple threads that can each be calling into your function. As such, you utilize this function to deal with merging the results from different threads into one final result. Depending on the type of aggregate you're doing, this can make things rather tricky. You can, however, make use of private members in your class to keep track of how many threads were running and reconcile the differences. It's worth noticing that this function receives a copy of your class as an argument (consider it to be what amounts to recursive when you are in this method) rather than whatever other type of value you've been accumulating — this is so you get the proper results that were calculated by the other thread.
- `Terminate` — This is essentially the opposite of `Init`. This is the call that actually returns the end result.

Now, let's see what this looks like in practice.

To start things off, create a new project in Visual Studio (I'm calling mine `ExampleAggregate`), and then add a new aggregate to the project (right-click on the project in the solution and choose Add➪Aggregate). SQL Server builds you a stock template that includes all four of the methods we just discussed:

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


[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct ExampleAggregate
{
    public void Init()
    {
        // Put your code here
    }

    public void Accumulate(SqlString Value)
    {
        // Put your code here
    }

    public void Merge(ExampleAggregate Group)
    {
        // Put your code here
    }

    public SqlString Terminate()
    {
        // Put your code here
        return new SqlString("");
    }

    // This is a place-holder member field
    private int var1;

}
```

This is the genuine foundation — complete with templates for all four method calls.

What we're going to be doing for an example in this section is to build an implementation of a `PRODUCT` function, which is essentially the same concept as `SUM` but multiplies instead of adding. Like the `SUM` function, we will ignore `NULL` values (unless they are all `NULL`, and then we will return `NULL`), but we will warn the user about the `NULL` being found and ignored should we encounter one.

We need to start with some simple changes. First, I'm going to change the class name to `Product` instead of `ExampleAggregate`, which we called the project. In addition, I need to declare some member variables to hold our accumulator and some flags.

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


[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct Product
{
    private SqlDouble dAccumulator;
    private bool fContainsNull;
    private bool fAllNull;
```

The `fContainsNull` variable will be used to tell us if we need to warn the user about any values being ignored. The `fAllNull` is used to tell if every value received was null — in which case we want to return null as our result.

We then need to initialize our member variables as part of the `Init` function:

```
    public void Init()
    {
        // Initialize our flags and accumulator
        dAccumulator = 1;
        fContainsNull = false;
        fAllNull = true;

    }
```

We are then ready to build the main accumulator function:

```
    public void Accumulate(SqlDouble Value)
    {
        // This is the meat of things. This one is where we actually apply
        // whatever logic is appropriate for our accumulation. In our example,
        // we simply multiply whatever value is already in the accumulator by
        // the new input value. If the input value is null, then we set the
        // flag that indicates that we've seen null values and then ignore
        // the value we just received and maintain the existing accumulation.

        if (Value.IsNull)
        {
            fContainsNull = true;
        }
        else
        {
            fAllNull = false;
            dAccumulator *= Value;
        }
    }
```

The comments pretty much tell the tale here. We need to watch to make sure that none of our flag conditions have changed. Other than that, we simply need to continue accumulating by multiplying the current value (assuming it's not null) by the existing accumulator value.

With the accumulator fully implemented, we can move on to dealing with the merge scenario.

```
    public void Merge(Product Group)
    {
        // For this particular example, the logic of merging isn't that hard.
        // We simply multiply what we already have by the results of any other
        // instances of our Product class.

        if (Group.dAccumulator.IsNull)
        {
            if (Group.fContainsNull)
                fContainsNull = true;
            if (!Group.fAllNull)
                fAllNull = false;
            dAccumulator *= dAccumulator;
        }
    }
```

For this particular function, the implementation of a merge is essentially just applying the same checks that we did in the `Accumulate` function.

Finally, we're ready to implement our `Terminate` function to close out our aggregation when it's done:

```
    public SqlDouble Terminate()
    {
        // And this is where we wrap it all up and output our results
        if (fAllNull)
        {
            return SqlDouble.Null;
        }
        else
        {
            SqlContext.Pipe.Send("WARNING: Aggregate values exist and were
ignored");
            return dAccumulator;
        }
    }
}
```

With all that done, we should be ready to compile our procedure and upload it:

```
CREATE ASSEMBLY ExampleAggregate
FROM '<solution path>\ExampleAggregate\bin\Debug\ExampleAggregate.dll'
```

And create the aggregate. Note that while an aggregate is a type of function, we use a different syntax to create it. The basic syntax looks like this:

```
CREATE AGGREGATE [ <schema name> . ] <aggregate name>
        (@param_name <input sqltype> )
RETURNS <return sqltype>
EXTERNAL NAME <assembly name> [ .<class name> ]
```

So, to create the aggregate from our assembly, we would do something like:

```
CREATE AGGREGATE dbo.Product(@input float)
RETURNS float
EXTERNAL NAME ExampleAggregate.Product
```

And, with that, we're ready to try it out. To test it, we'll create a small sample table that includes some data that can be multiplied along with a grouping column, so we can test out how our aggregate works with a `GROUP BY` scenario.

```
CREATE TABLE TestAggregate
(
    PK            int        NOT NULL    PRIMARY KEY,
    GroupKey    int        NOT NULL,
    Value        float    NOT NULL
)
```

Now we just need some test data:

```
INSERT INTO TestAggregate(PK, GroupKey, Value)
    VALUES (1, 1, 2)
INSERT INTO TestAggregate(PK, GroupKey, Value)
    VALUES (2, 1, 6)
INSERT INTO TestAggregate(PK, GroupKey, Value)
    VALUES (3, 1, 1.5)
INSERT INTO TestAggregate(PK, GroupKey, Value)
    VALUES (4, 2, 2)
INSERT INTO TestAggregate(PK, GroupKey, Value)
    VALUES (5, 2, 6)
```

And we're ready to give our aggregate a try. What we're going to be doing is returning the `PRODUCT` of all the rows within each group (our sample data has two groups, so this should work out to two rows).

```
SELECT GroupKey, dbo.Product(Value) AS Product
FROM TestAggregate
GROUP BY GroupKey
```

Run this and we get back two rows (just as we expected):

```
GroupKey    Product
----------- ----------------------
1           18
2           12

(2 row(s) affected)
```

Do the match on our sample data, and you'll see we got back just what we wanted.

If you're thinking about it, you should be asking yourself "OK, this is great, but how often am I really going to use this?" For most of you, the answer will be "never." There are, however, those times where what's included just isn't ever going to do the job. Aggregates are one of those places where special cases come rarely, but when they come, they really need exactly what they need and nothing else. In short, I wouldn't crowd your brain cells by memorizing every little thing about this section, but do take the time to learn what's involved and get a concept for what it can and can't do so you know what's available should you need it.

Creating Triggers from Assemblies

Much like the other assembly types we've worked with so far in this chapter, triggers have a lot in common with the rest, but also their own little smattering of special things.

The differences will probably come to mind quickly if you think about it for any length of time:

- How do we deal with the contextual nature of triggers? That is, how do we know to handle things differently if it's an `INSERT` trigger situation versus a `DELETE` or `UPDATE` trigger?
- How do we access the inserted and deleted tables?

You may recall from earlier examples, how we can obtain the "context" of the current connection — it is by utilizing this context that we are able to gain access to different objects that we are interested in. For example, the `SqlContext` object that we've obtained a connection from in prior examples also contains a `SqlTriggerContext` object — we can use that to get properties such as whether we are dealing with an insert, update, or delete scenario (the first question we had). The fact that we have access to the current connections also implies that we are able to access the inserted and deleted tables simply by querying them. Let's get right to putting this to use in an example.

Start by creating a new SQL Server project in Visual Studio (I've called mine `ExampleTrigger` this time). Once your project is up, right-click the project in the solution explorer and select Add➪Trigger....

Visual Studio is nice enough to provide you with what is, for the most part, a working template. Indeed, it would run right as provided except for one issue:

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


public partial class Triggers
{
    // Enter existing table or view for the target and uncomment the attribute line
    // [Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
Target="Table1", Event="FOR UPDATE")]
```

```
    public static void ExampleTrigger()
    {
        // Replace with your own code
        SqlContext.Pipe.Send("Trigger FIRED");
    }
}
```

I've highlighted the key code line for you. At issue is that we must provide more information to SQL Server than we do in our other object types. Specifically, we must identify what table and events we're going to be executing our trigger against. We're actually going to create a special demonstration table for this before the trigger is actually put into action, so we can just use the table name `TriggerTable` for now.

```
    [Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
Target="TriggerTable", Event="FOR INSERT, UPDATE, DELETE")]
```

Notice that I've also altered what events will fire our trigger to include all event types.

Now we'll update the meat of things just a bit, so we can show off different actions we might take in our trigger and, perhaps more importantly, how we can check the context of things and make our actions specific to what has happened to our table. We'll start by getting our class going:

```
    public static void ExampleTrigger()
    {
        // Get a handle to our current connection
        SqlConnection cn = new SqlConnection("context connection=true");
        cn.Open();

        SqlTriggerContext ctxt = SqlContext.TriggerContext;
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = cn;
```

So far, this isn't much different from what we've used in our other .NET examples. Perhaps the only significant difference from things we've seen already is the `SqlTriggerContext` object — we will use this later on to determine what action caused the trigger to fire.

We're ready to start code that is conditional on the action the trigger is firing for (based on the `TriggerAction` property of the `TriggerContext` of the `SqlContext`. For this, I'm going to use a simple `switch` command (though there are those that will call me a programming charlatan for using a `switch` statement — to them I say "deal with it!"). I'm also going to pipe out various things to the client to report what we're doing.

> **In practice, you generally do not want to be outputting information from a trigger — figure that they should usually run silently as far as the client is concerned. I've gone ahead and output several items in this example just to make it readily apparent what the trigger is doing under what scenario.**

```
            switch (ctxt.TriggerAction)
            {
                case TriggerAction.Insert:
                    cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM INSERTED";
                    SqlContext.Pipe.Send("Insert Trigger Fired");
                    SqlContext.Pipe.ExecuteAndSend(cmd);
                    break;

                case TriggerAction.Update:
                    // This time, we'll use datareaders to show how we can
                    // access the data from the inserted/deleted tables

                    SqlContext.Pipe.Send("Update Trigger Fired");
                    SqlContext.Pipe.Send("inserted rows...");
                    cmd.CommandText = "SELECT * FROM INSERTED";
                    SqlContext.Pipe.Send(cmd.ExecuteReader());
                    break;

                case TriggerAction.Delete:
                    // And now we'll go back to what we did with the inserted rows...
                    cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM DELETED";
                    SqlContext.Pipe.Send("Delete Trigger Fired");
                    SqlContext.Pipe.ExecuteAndSend(cmd);
                    break;
            }

            SqlContext.Pipe.Send("Trigger Complete");
        }
    }
```

And, with that, we're ready to compile and upload it. The assembly upload works just as most of them have so far (we're back to not needing anything other than the default `PERMISSION_SET`).

```
CREATE ASSEMBLY ExampleTrigger
FROM '<solution path>\ExampleTrigger\bin\Debug\ExampleTrigger.dll'
```

Before we get to creating the reference to the trigger, however, we need a table. For this example, we'll just create something very simple:

```
CREATE TABLE TestTrigger
(
    PK        int                NOT NULL PRIMARY KEY,
    Value    varchar(max)    NOT NULL
)
```

With the assembly uploaded and the table created, we're ready to create our trigger reference.

Much like stored procedures and functions, a .NET trigger creation is made from the same statement as T-SQL-based triggers. We eliminate the T-SQL side of things and replace it with the `EXTERNAL NAME` declaration.

```
CREATE TRIGGER trgExampleTrigger
ON TestTrigger
FOR INSERT, UPDATE, DELETE
AS EXTERNAL NAME ExampleTrigger.Triggers.ExampleTrigger
```

And with that, our trigger should be in place on our table and ready to be fired whenever one of its trigger actions is fired (which happens to be for every trigger actions), so let's test it.

We'll start by getting a few rows inserted into our table. And, wouldn't you just know it? That will allow us to test the insert part of our trigger.

```
INSERT INTO TestTrigger
    (PK, Value)
VALUES
    (1, 'first row')

INSERT INTO TestTrigger
    (PK, Value)
VALUES
    (2, 'second row')
```

Run this, and we not only get our rows in but we also get a little bit of feedback that is coming out of our trigger:

```
Insert Trigger Fired
NumRows
-----------
1

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)
Insert Trigger Fired
NumRows
-----------
1

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)
```

As you can see, we're getting output from our trigger. Notice that we're getting the "`(1 row(s) affected)`" both from the query running inside the trigger and from the one that actually inserted the data. We could have taken any action that could have been done a T-SQL trigger (though many are more efficient if you stay in the T-SQL world). The key is that we could do so much more if we had the need. We could, for example, make an external call or perform a calculation that isn't doable in the T-SQL world.

There is an old saying: "Caution is the better part of valor." This could have been written with triggers in mind. I can't possibly express enough about the "be careful" when it comes to what you're doing in triggers. Just because you can make an external call doesn't make it a smart thing to do. Assess the need — is it really that important that the call be made right then? Realize that these things can be slow, and whatever transaction that trigger is participating in will not complete until the trigger completes — this means you may be severely damaging performance.

Okay, so with all that done, let's try an update:

```
UPDATE TestTrigger
SET Value = 'Updated second row'
WHERE PK = 2
```

And let's see what we get back:

```
Update Trigger Fired
inserted rows...
PK          Value
----------- ---------------------------------------
2           Updated second row

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)
```

The result set we're getting back is the one our trigger is outputting. That's followed by some of our other output as well as the base "`(1 row(s) affected)`" that we would normally expect from our single row update. Just as with the insert statement, we were able to see what had happened and could have adapted accordingly.

And so, that leaves us with just the delete statement. This time, we'll delete all the rows, and we'll see how the count of our deleted table does indeed reflect both of the deleted rows.

```
DELETE TestTrigger
```

And again check the results:

```
Delete Trigger Fired
NumRows
-----------
2

(1 row(s) affected)

Trigger Complete

(2 row(s) affected)
```

Now, these results may be just a little confusing, so let's look at what we have.

We start with the notification that our trigger fired — that comes from our trigger (remember, we send that message down the pipe ourselves). Then comes the result set from our `SELECT COUNT(*)`. Notice the "`(1 row(s) affected)`" — that's from our result set rather than the `UPDATE` that started it all. We then get to the end of execution of our trigger (again, we dropped that message in the pipe), and, finally, the "`(2 row(s) affected)`" that was from the original `UPDATE` statement.

And there we have it. We've done something to address every action scenario, and we could, have course, done a lot more within each. We could also do something to address a `BEFORE` trigger if we needed to.

Custom Data Types

Sometimes you have the need to store data that you want to be strongly typed, but that SQL Server doesn't fit within SQL Server's simple data type list. Indeed, you may need to invoke a complex set of rules in order to determine whether the data properly meets the type requirement or not.

> *Requests for support of complex data types have been around a very long time. Indeed, I can recall being at the Sphinx Beta 2.0 — known to most as Beta 2 for SQL Server 7.0 — event in 1998, and having that come up as something like the second most requested item in a request session I was at. Well, it took a lot of years, but it's finally here.*

By utilizing a .NET assembly, we can achieve a virtually limitless number of possibilities in our data types. The type can have complex rules or even contain multiple properties.

Before we get to the syntax for adding assemblies, let's get an assembly constructed.

> **The sample used here will be the `ComplexNumber.sln` solution included in the SQL Server samples. You will need to locate the base directory for the solution — the location of which will vary depending on your particular installation.**

We need to start by creating the signature keys for this project. To do this, I recommend starting with your solution directory being current and then calling `sn.exe` using a fully qualified path (or, if your .NET framework directory is already in your `PATH`, then it's that much easier!). For me, it looks like this:

```
C:\Program Files\Microsoft.NET\SDK\v2.0 64bit\LateBreaking\SQLCLR\UserDefinedDat
aType>"C:\Program Files (x86)\Microsoft Visual Studio 8\SDK\v2.0\Bin\sn" -k temp
.snk
```

And with that, you're ready to build your DLL.

Let's go ahead and upload the actual assembly (alter this to match the paths on your particular system):

```
CREATE ASSEMBLY ComplexNumber
  FROM <solution path>\ComplexNumber\bin\debug\ComplexNumber.dll'
WITH PERMISSION_SET = SAFE;
```

And with the assembly loaded, we're ready to begin.

Creating Your Data Type from Your Assembly

So, you have an assembly that implements your complex data type and have uploaded it to SQL Server using the `CREATE ASSEMBLY` command. You're ready to instruct SQL Server to use it. This works pretty much as other assemblies have. The syntax looks like this:

```
CREATE TYPE [ <schema name>. ] <type name>
{
    FROM <base type>
    [ ( <precision> [ , <scale> ] )  ]
    [ NULL | NOT NULL ]
  | EXTERNAL NAME <assembly name> [ .<class name> ]
} [ ; ]
```

I've put the full type definition there, but really it's just the highlighted line that is different from the older style user-defined data type definition. You'll notice immediately that it looks like our previous assembly-related constructs, and, indeed, the use is the same.

So, utilizing our complex type created in the last section, it would look like this:

```
CREATE TYPE ComplexNumber
  EXTERNAL NAME [ComplexNumber].[Microsoft.Samples.SqlServer.ComplexNumber];
```

Accessing Your Complex Data Type

Microsoft has provided a filed called `test.sql` for testing the assembly we just defined as our complex data type, but I find it falls just slightly short of where we want to be in our learning here. What I want to emphasize is how the various functions of the supporting class for our data type are still available. In addition, each individual property of the variable is fully addressable. So, let's run a modified version of the provided script:

```
USE AdventureWorks
GO

-- create a variable of the type, create a value of the type and invoke
-- a behavior over it

DECLARE @c ComplexNumber;

SET @c = CONVERT(ComplexNumber, '(1, 2i)');


SELECT @c.ToString() AS FullValueAsString;

SELECT @c.Real AS JustRealProperty
GO
```

Now run it, and check out the results:

```
FullValueAsString
------------------
 (1,2i)

(1 row(s) affected)

JustRealProperty
------------------
1

(1 row(s) affected)
```

In the first result that was returned, the `ToString` function was called as defined as a method of our class. The string is formatted just as our method desires. If we had wanted to reverse the order of the numbers or some silly thing like that, we would only have needed to change the `ToString` function in the class, recompile it, and re-import it our database.

In our second result, we address just one property of our complex data type. The simple dot "." delimiter told SQL Server that we were looking for a property — just as it would in C# or VB.NET.

Dropping Data Types

As you might expect, the syntax for dropping a user defined data type works just like other drop statements:

```
DROP TYPE [ <schema name>. ] <type name> [ ; ]
```

And it's gone — maybe.

Okay, so why a "maybe" this time? Well, if there is most any object out there that references this data type, then the `DROP` will be disallowed and will fail. So, if you have a table that has a column of this type, then an attempt to drop it would fail. Likewise, if you have a schema bound view, stored procedure, trigger, or function defined that utilizes this type, then a drop would also fail.

> *Note that this form of restriction appears in other places in SQL Server — such as dropping a table when it is the target of a foreign key reference — but those restrictions tend to be less all encompassing than this one is (virtually any use of it in your database at all will block the drop), so I haven't felt as much need to point it out (they were more self-explanatory).*

Summary

Well, if you aren't thinking to yourself something along the lines of "Wow, that's powerful," then I can only guess you somehow skipped straight to the summary without reading the rest of the chapter. That's what this chapter is all about — giving you the power to do very complex things (or, in a few cases, simple things that still weren't possible before). What this chapter was also about, in a very subtle way, was a mechanism that can break your system very quickly. Yeah, yeah, yeah — I know I'm resorting to scare tactics, but I make no apologies for it. When using assemblies, you need to be careful. Think about what you're doing, and analyze each of the steps that your assembly is going to be taking even more

thoroughly than you already do. Consider latency you're going to be adding if you create long-running processes. Consider external dependencies you are creating if you make external calls — how reliable are those external processes? You need to know, as your system is now only as reliable as the external systems you're calling.

Now, having hopefully scared you into caution about assemblies, let me ease up a bit and say don't avoid them solely out of fear. Assemblies were added for a reason, and they give us a power we both need and can use (not to mention that they are just plain cool). As always, think about what you need, and don't make your solution any more complex than it needs to be. Keep in mind, however, that what seems at first to be the more complex solution may actually be simpler in the end. I've seen stored procedures that solved the seemingly unsolvable T-SQL problem. Keeping your system away from assemblies would seem to make it simpler, but what's better: a 300-line, complex T-SQL stored proc or an assembly that is concise and takes only 25 lines including declarations?

Choose wisely.