Chapter 6

Client-Side ADO.NET

In this chapter:

I have been a big fan of Microsoft ADO.NET since its release. I was also a big ActiveX Data Object (ADO) fan, and even a Data Access Objects (DAO) fan. I remember creating a wrapper for direct calls to the Open Database Connectivity (ODBC) application programming interface (API)—good stuff. But I am here to tout ADO.NET 2.0, and I have to tell you that my enthusiasm continues with this latest data access technology from Microsoft.

On initial glance, ADO.NET 2.0 looks very much like its predecessor—ADO.NET 1.1. All the familiar objects are there; all the functionality you expect is still there. So what has changed? There is the built-in capability to do data paging, or perhaps Multiple Active Results Sets (MARS) appeals to you more, or maybe even asynchronous database calls would pique your interest—and those are just the features that are not specific to Microsoft SQL Server 2005. ADO.NET 2.0 also includes features that are specific to SQL Server 2005—such as the fact that user-defined data types (UDTs), which are also called user-defined types, are natively supported by data access client.

ADO.NET 2.0 can really take your data access coding to a new level. In this chapter, I will disclose information about the myriad new features and changes to the existing features. You will then want to further explore this latest version of data access technology.

In the Beginning

I'll start by telling you about the features of ADO.NET 2.0 that are directly related to SQL Server 2005. In Chapter 4, "Microsoft .NET Framework Integration," you saw one new feature—server-side cursors via the *SqlResultSet* object—and probably didn't think twice about it. There is a good reason it was discussed in that chapter—it is meant to be used in the context of stored procedures and other database objects that are written in managed code. You could use *SqlResultSet* on the client, but you might pay a performance penalty for doing so without receiving any meaningful benefit. With very few exceptions, server-side cursors belong on the server.

So what else does ADO.NET 2.0 offer us that is specifically designed to work with SQL Server 2005? You might recall that Chapter 5, "User-Defined Data Types," focused on the ability to create a UDT in managed code and freely utilize it in your database. Suppose that you did just that, creating a table that contained a column defined as a point UDT, as shown in Listing 6-1.

Listing 6-1

```
CREATE TABLE Location
(
    LocationID int Identity(1, 1) NOT NULL PRIMARY KEY,
    Description varchar(150) NOT NULL,
    Coordinate point NOT NULL
)
```

You can also then create a stored procedure that selected a particular record from the table based on the *LocationID* field, exhibited in Listing 6-2.

Listing 6-2

```
CREATE PROCEDURE prApp_Location_Select
    @LocationID int
AS
SELECT   LocationID, Description, Coordinate
FROM     Location
WHERE    LocationID = @LocationID
GO
```

This all seems innocent enough until you consider what will happen when you call this stored procedure from the client. How can you use the *Coordinate* field on the client if the type definition exists in an assembly that is literally embedded in the database? As you are about to see, there are several answers to this question.

Accessing UDTs in SQL Server

If you are using *SqlClient* to provide connectivity to SQL Server, you're in luck because it natively supports those UDTs you created in your SQL Server database. There are two methods for using these types: depending on the purpose and ultimate use of your software, either method can be a reasonable choice.

Type Exists in Client Code—Early-Binding

Perhaps you are creating a complete software solution for a client or for internal use using SQL Server and .NET. In situations such as these, when you or your group have control over all aspects of the development, you can easily use the same assembly code in and out of the database as needed. This early-bound technique is marvelous. It's easy to use, and it's cleaner from a coding standpoint.

To use this technique, you have to use the same assembly in your data access code as you do in the database itself. This is simple enough to do because you had to create the assembly before registering, thus loading it into the database in the first place. That very same assembly needs to be referenced in your .NET client project, and you then have the type at your disposal for use in your project.

The example shown in Listing 6-3 demonstrates a UDT being used in client code. In the SalesContact table, the fifth column (zero-based index value of 4) is the Email Struct UDT that was presented in Chapter 5.

Listing 6-3

```
SqlConnection sconn = new SqlConnection("integrated security=SSPI;data source=YUKONDEV03;ini
tial catalog=AdventureWorks;");
sconn.Open();
SqlCommand scmd = new SqlCommand("SELECT * FROM SalesContact", sconn);
SqlDataReader sdr = scmd.ExecuteReader();
sdr.Read();
//Column index 4 is the Email UDT
Email em = (Email)sdr[4];
Console.WriteLine(em.Address);
```

Because the fifth column is indeed a UDT, you should address it as a UDT by using the type itself when addressing the column (pun intended). Almost any other attempt to address this UDT column will lead to either a compile or run-time failure. Listing 6-4 demonstrates other attempts to read data from the UDT column.

Listing 6-4

```
1   SqlConnection sconn = new SqlConnection("integrated security=SSPI;data source=YUKONDEV03
;initial catalog=AdventureWorks;");
2   sconn.Open();
3   SqlCommand scmd = new SqlCommand("SELECT * FROM SalesContact", sconn);
4   SqlDataReader sdr = scmd.ExecuteReader();
5   sdr.Read();
6   //Column index 4 is the Email UDT
7   //The following code fails at runtime because this column is not a String
8   Console.WriteLine(sdr.GetString(4));
9   //
The following code fails when compiled because Object doesn't define an Address field or pro
perty
10  Console.WriteLine(sdr[4].Address);
11  //But the following code work fine
12  Console.WriteLine(sdr[4]);
```

```
13  Console.WriteLine(sdr[4].ToString());
14  Console.WriteLine(((Email)sdr[4]).Address);
```

The *GetString* method on line 8 compiles without any problem, but it throws an *InvalidCastException* when executed because an *Email* is not a *String*. The second example on line 10 uses the Address property of the *Email* type directly against the column of the DataReader. And, although this seems like it should work, the compiler won't even compile the code since a DataReader column does not have any such property.

On the other hand, the last three lines not only compile, but execute without a hitch. The example on line 14 of the code simply casts the column to the UDT before trying to use the Address property. This is essentially the same code as that in Listing 6-3, but instead of creating the type ahead of time, the data is being cast to the type as needed. Lines 12 and 13, however, do no such cast to the *Email* type, yet they both work without any problems. The details of why is beyond the scope of this book, but it should suffice to say that it works because the *ToString* method is common to all objects, regardless of their type, including (and required for) all UDTs created for SQL Server 2005.

Accessing UDT Bytes

Let's start with a couple of assumptions. First, assume that the type is not available for use on the client. Second, assume that dynamically downloading the assembly that contains the UDT is not feasible, perhaps because it is part of a large assembly and you want to avoid the additional overhead of downloading such an assembly, or maybe security prevents the type from being downloaded. What do you do if you want to call a stored procedure that returns a result set with a column that is in fact a UDT and you do not have that type on the client?

Not to worry—you can access the UDT in the same way as the serialization methods of the UDT. More specifically, because the column data is in the form of raw bytes, you can read these raw bytes using the same technique as the *Read* method of the UDT itself, as shown in Listing 6-5.

Listing 6-5

```
SqlConnection sconn = new SqlConnection("integrated security=SSPI;data source=YUKONDEV03;ini
tial catalog=AdventureWorks; UDT Assembly Download=True");
sconn.Open();
SqlCommand scmd = new SqlCommand("SELECT * FROM SalesContact", sconn);
SqlDataReader sdr = scmd.ExecuteReader();
sdr.Read();
Byte[] bEmail = new Byte[122];
sdr.GetBytes(4, 0, bEmail, 0, 122);
using (BinaryReader br =

    new BinaryReader(new MemoryStream(bEmail))
{
    Byte bNull = br.ReadByte();

    if (bNull == 0)
```

```
            Console.WriteLine(br.ReadString().TrimEnd(new Char[] {'0'}));
        else
            Console.WriteLine("{NULL}");

        br.Close();
    }
```

Again, you might recognize some of the code from Chapter 5. This code uses the same technique to read raw bytes and convert them into a string as the *Read* method of the *Email* type itself. The code varies in what it does with these values, but not in how it extracts them. A byte array is read from the raw bytes of the column that has the UDT. This is fed into a *MemoryStream*, which in turn is fed into a *BinaryReader*, which calls its *ReadByte* and *ReadString* methods to evaluate the content of the UDT.

Although this technique is a viable solution, it requires additional code management; at the lines-of-code level (not at the assembly level), which to many is definitely not preferred. My suggestion (one last time) is to use the early-bound technique and use the assembly in the client code. You still have to manage code in multiple locations, but at least it's a single unit of code—an assembly—that can be distributed to both server and client, as needed.

Caution Using this technique of direct byte access requires that you know the method of serialization that the type is using. But because the implementation of the type's serialization can be changed in the database, your client code could all of a sudden be not only inaccurate, but possibly dangerous, because any changes saved by this now rogue code could corrupt your data.

Data Paging

Shortly after I devised a means of paging data in ASP.NET using data from SQL Server, I started digging into the new features of ADO.NET 2.0 and discovered that one of these new features was the capability to page data. As I learned about this nifty little feature, I realized that it will make obsolete many of the data-paging methods currently in use in the development community, including the data-paging methods I myself had devised. However, it will simplify development of Web-based applications that need to be able to page data results.

The *ExecutePageReader* Method

ExecutePageReader is a new method of the *Command* object. It is not yet available to all variations of the *Command* object—including the *OdbcCommand* and *OleDbCommand* objects. As of this writing, it is available only to the SQL Server data access objects, specifically *SqlCommand*. Keep in mind, however, that this is beta software and it will be enhanced before its final public release.

So how does it work? Under the covers, ADO.NET is doing something that I usually advise against when developing data layer software: It opens a server-side cursor against the entire set of data, positions to the rows to fetch, fetches those rows, and finally closes that server-side cursor. Server-side cursors can adversely affect your application in several aspects, but in this case it is used very effectively, and its implementation is hidden to prevent any misuse of the technology.

ExecutePageReader takes three parameters: one defines its behavior, one indicates the row position, and one tells it how many rows to fetch, as shown in its syntax:

```
public SqlDataReader ExecutePageReader
(CommandBehavior behavior, Int32 startRow , Int32 pageSize)
```

The first parameter, *behavior*, is one of the *System.Data.CommandBehavior* enumeration values. This enumeration is already a part of ADO.NET 1.1 and behaves in the same fashion in ADO.NET 2.0. The *startRow* parameter is a *zero-based* position that indicates where to start fetching data. If this value exceeds the actual number of rows, no data is returned. The *pageSize* parameter tells the database how many rows should be fetched. If fewer rows are available, the remaining rows are returned.

The code sample shown in Listing 6-6 is a variation on a paging methodology—published in *asp.netPro* magazine in June 2003 by my friend and colleague, Jeff Prosise—that currently can be accomplished in ADO.NET 1.1.

Listing 6-6

```
SqlDataReader GetPage (Int32 index, Int32 size)
{
    //index:zero-based page number
    //size: number of records per page
    String command = String.Format(
         "SELECT * FROM " +
         "(SELECT TOP {0} * FROM " +
         " Contact ORDER BY ContactID) AS t1 " +
         WHERE ContactID NOT IN " +
         "(SELECT TOP {1} ContactID FROM Contact " +
         "ORDER BY ContactID) ",
         size * (index + 1), size * index,
    );

   SqlConnection conn = new SqlConnection
          ("server=.;database=AdventureWorks;Trusted_Connection=yes");
    conn.Open();
    SqlCommand cmd = new SqlCommand(command, conn);
    SqlDataReader dr = cmd.ExecuteReader();
    return dr;
}
```

Note Although this method uses dynamically created T-SQL, a feature that I have often touted as a no-no, the manner in which it is implemented protects it from SQL injection attacks because of the strong typing of the method parameters.

This method dynamically constructs a T-SQL statement that will indeed fetch a page of data from the requested table. Because the paging work is being done on the server, it is in many cases more efficient than other paging techniques. That was true until ADO.NET 2.0 came along. Now the technique shown in Listing 6-6 will never be as efficient as the code sample shown in Listing 6-7, which uses the new *ExecutePageReader* method.

Listing 6-7

```
SqlDataReader GetPage (Int32 index, Int32 size)
{
   //index:zero-based page number
   //size: number of records per page
   String command = "SELECT * FROM Contact ORDER BY ContactID";

   SqlConnection conn = new SqlConnection
         ("server=.;database=AdventureWorks;Trusted_Connection=yes");
   conn.Open();
   SqlCommand cmd = new SqlCommand(command, conn);
   SqlDataReader dr = cmd.ExecutePageReader(
         CommanBehavior.CloseConnection, (size * index) + 1 , size);
   return dr;
}
```

It is true that both techniques can potentially access the entire set of data. The former technique must, however, perform several sorts of the data; whereas *ExecutePageReader* sorts once and uses a forward-only, read-only server-side cursor to get to the data. It outperforms the more complex and bulky T-SQL statement of the former technique.

This is not necessarily the best way to implement a data paging solution—there are other methodologies that can be more efficient. Unfortunately, some of these techniques involve complex code and the need to make structural changes to the database design to get the job done right. *ExecutePageReader*, on the other hand, lets you create a quick and efficient paging solution while keeping the code simple—and without having to make any design changes to your database.

Asynchronous Commands

The concept of making asynchronous calls is not new to the world of application development, especially .NET development, but it is new to ADO.NET 2.0. Objects that support asynchronous execution offer several methods in which to implement this coding technique. This section describes two of these methods: polling and callback.

Asynchronous Polling

The polling technique works by starting an asynchronous call and then testing to see whether the job is complete. If the job is not complete, it can do other tasks and then check again, which is accomplished by using a *loop* statement that checks the completion state of the asynchronous job. This is comparable to a family car trip: A child keeps asking "Are we there yet?" and, between each inquiry, watches a DVD or plays with a portable game system. Sound horrific? My friend and colleague Jeffrey Richter thought it was horrible—polling wasted processor time that could otherwise be used by other processes. We discussed this technique while at a conference and continued the conversation when I called him a few weeks later.

In defense of asynchronous polling as a solution, Jeffrey and I tried to find a viable example that used this technique, but for each example that we conjured up, we realized that it would be better implemented using callbacks. After a lengthy chat, we found that we couldn't find a good example of using polling because every polling example we thought of was better if written by using callbacks. By the end of the discussion, I was in agreement with Jeffrey's first assessment: Polling is quite often the wrong technique to implement and it should be used only if you cannot use callbacks to achieve the same results.

Although there are times when you will have no other choice but to use polling as a technique (client demands, lack of database support for this technique, and so on), using callbacks instead of polling almost always will be a better choice.

Asynchronous Callback

To continue the car trip analogy, this child makes one simple request at the beginning of the trip: "Let me know when we get there." You only have to tell your child when you actually arrive and in the meantime, this child keeps busy with DVDs or games. This is in essence how the callback technique works: You make a command request; when the command finishes, it notifies you in another method. Many factors can affect the decision to use callbacks (including the complexity of implementing a solution), but using asynchronous callbacks is the preferred technique when using asynchronous methodologies.

When you decide that using callbacks is what you need, Listing 6-8 shows you how to do just that.

Listing 6-8

```
private SqlCommand sqlCmd;
public void StartCommand()
{
    using (SqlConnection sconn = new SqlConnection("integrated security=SSPI;data source=YUK
ONDEV03;initial catalog=AdventureWorks"))
    {
        sconn.Open();
        sqlCmd = new SqlCommand("SELECT * FROM Person.Contact", sconn);

    IAsyncResult ar = sqlCmd.BeginExecuteReader (
```

```
            this.EndCommand, null);
    }
}
private void EndCommand(IAsyncResult ar)
{
    SqlDataReader sdr = sqlCmd.EndExecuteReader(ar);
    // do something with the SqlDataReader
}
```

Although a detailed explanation of the way callbacks work in .NET is beyond the scope of this book, you should know that this technique can be very useful when you need to have other completely different code segments running independently of the request for data. The classic example occurs when a client application needs to fetch data that might take time to retrieve and allows the user to perform other tasks while waiting for the command to finish its job on the database server.

The ability to execute commands asynchronously has many excellent benefits, but, like other features that have been discussed, it is only one type of solution. A prototype application, for example, is easier to implement by using synchronous calls. And although synchronous calls are certainly easier to implement in an application than asynchronous calls, a well-written application should use asynchronous calling techniques.

Multiple Active Results Sets

Also known as the more familiar MARS, Multiple Active Results Sets allow you to have concurrent access to more than one results set on the same connection. To think that this feature is like the current ability to get multiple results sets using the *NextResult* method of a *DataReader* is selling MARS short. MARS isn't about retrieving sequential sets of results from a database server. It allows you to have multiple results sets, each acting independently of the others as if they were all using separate connections, when in fact they are all using the same connection to get the job done.

Important For Beta 2 of SQL Server 2005, the connection string requires the additional setting *async=true* to use MARS. If this setting is not present, an InvalidOperationException exception will be thrown when attempting to fetch data from the second *SqlCommand*.

It might not seem like such a big deal. However, if you consider a Web application with lots (thousands or more) of users, each of whom needs to deal with simultaneous multiple sets of data that would normally require multiple connections, and then take into consideration the "cost" of a connection, it can make a major difference in the performance of the Web application if the simultaneous access of multiple results sets can use a single connection.

Important Access to multiple active sets of data on a single connection does not mean that you can open multiple instances of a *SqlDataReader* using a single *SqlCommand* as the source. Although multiple *SqlCommand* objects can be associated with a single *SqlConnection*, each *SqlDataReader* must be associated with a single *SqlCommand*.

This feature can also be used in conjunction with asynchronous commands. Imagine some process that needs to hook into the same SQL Server multiple times to process two distinct queries. The code in Listing 6-9 demonstrates the concept (although the actual tables do not exist). This code starts two separate asynchronous commands on the same connection.

Listing 6-9

```
using (SqlConnection sconn = new SqlConnection("integrated security=SSPI;data source=YUKONDE
V03;initial catalog= AdventureWorks;use mdac9=True"))
{
    sconn.Open();
    SqlCommand scmd = new SqlCommand("SELECT * FROM Person.Contact WHERE ContactID = 1", sco
nn);
    SqlCommand scmd2 = new SqlCommand("SELECT * FROM Sales.Customer WHERE CustomerID = 1", s
conn);
    IAsyncResult ar = scmd.BeginExecuteReader();
    IAsyncResult ar2 = scmd2.BeginExecuteReader();
    SqlDataReader sdr = scmd.EndExecuteReader(ar);
    SqlDataReader sdr2 = scmd2.EndExecuteReader(ar2);
    }
}
```

Because both commands are executing on the same connection, a negligible amount of time is needed to make the "second" connection. Thus, by synchronously executing the second command on the same connection as the first command that is asynchronously executing, you effectively get an execution time that is equivalent to the length of the query that takes more time to execute. Using some simplified accounting methods, if you assume that the first command requires 1 second to execute and the second requires .5 second to execute, executing these two commands asynchronously on the same connection requires about 1 second because the second command executes within the timeframe of the first executing. The same process executed completely synchronously would take 1.5 seconds. That's a potential saving of 33.33 percent over the equivalent synchronously executing code. This explanation describes the process well enough for an introductory understanding without going into an explanation that goes beyond the scope of this book.

Summary

ADO.NET has certainly moved up to a higher plateau. The data access capabilities discussed in this chapter are a portion of the new features you will see in version 2.0. These features will enable you to create most robust applications that will perform better as well.

Because this chapter wraps up the primary aspects of writing managed code to access SQL Server, You might think that you should be zealous in using all these new features; however—as far as real development projects are concerned—remember the old adage: everything in moderation.