CHAPTER 3

Effective Use of ADO.NET: Creating a Survey Application

In This Chapter:

- ADO.NET Overview
- The Survey Application
- Extending and Modifying the Survey Application
- Deploying the Survey Application

This chapter talks about effectively using ADO.NET. I'll spend the first part of the chapter introducing ADO.NET, mostly how it relates to the SQL Server managed provider. In the second half of the chapter, I'll walk you through a Survey application that was built using ADO.NET, and which focuses on a number of best practices along with some recommended design patterns.

Because this is an odd-numbered chapter, the featured program's (the Survey application's) source code is shown in VB.NET (VB). For the full C#.NET (C#) source code, just go to www.ASPNET-Solutions.com, follow the links to the examples for Chapter 3, and download the C# source code. (You can also download the VB.NET source code from a link on the same page.)

I have included a section toward the end of the chapter named "Extending and Modifying the Survey Application." In this section, I talk about ways to enhance the application so that it's even more useful. I'll post any changes I make to the Survey application (including these suggestions) on the Web site. Check for them on the Chapter 3 page. The site also includes a forum for discussion about the Survey application. And if you want to send me your modifications, I'll gladly post them for other users.

ADO.NET Overview

Before you can use ADO.NET components, the appropriate namespaces must be included. The System.Data namespace always needs to be included because it contains the core database components. Next, depending on the source of your data, one of two namespaces needs to be included. For a direct SQL Server connection, the System.Data.SqlClient namespace should be used for best performance. For all other connection types, such as Access and Oracle, the System.Data.OleDb namespace is required. (An Oracle provider is now available at http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/940/msdncompositedoc.xml.)

The SqlClient data provider is fast. It's faster than the Oracle provider, and faster than accessing a database via the OleDb layer. It's faster because it accesses the native library (which automatically gives you better performance), and it was written with lots of help from the SQL Server team (who helped with the optimizations).

Managed Providers

Managed providers are a central part of the ADO.NET framework. Managed providers enable you to write language-independent components that can be called from C# and VB. Currently, managed providers come in two types: one for direct access to Microsoft SQL Server 7.0 and higher, and one for accessing data via an OLE DB layer. Both types use similar naming conventions, with the only difference being their prefixes.

The managed provider classes include `Connection` (`SqlConnection` class), `Command` (`SqlCommand` class), `DataReader` (`SqlDataReader` class), and `DataAdapter` (`SqlDataAdapter` class). The first two classes provide the same functionality that was found in ADO: creating a connection to a data source and then executing a command. A data reader has a close resemblance to a read-only, forward-only recordset that is very optimized. Last, the DataAdapter allows for the retrieval and saving of data between a DataSet and the data source. The DataSet is covered in Chapter 4.

Connection

To create a database connection, you need to include the appropriate namespaces in your application. This requires the data provider to be known, so either a SqlClient or OleDb namespace connection can be included for the

best performance. The following code samples (Listings 3.1 and 3.2) show how both SqlClient and OleDb connections are made in C# and VB.

Listing 3.1 Using the SqlConnection Object

C#

```
SqlConnection myConnection  =
      new SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" );
myConnection.Open();
// Do Something with myConnection.
myConnection.Close();
```

VB

```
Dim myConnection As New
SqlConnection("server=localhost;uid=sa;pwd=;database=pubs")
myConnection.Open()
' Do something with myConnection.
myConnection.Close()
```

Listing 3.2 Using the OleDbConnection Object

C#

```
OleDbConnection myConnection =
  new OleDbConnection("Provider=SQLOLEDB.1;" +
    "Data Source=localhost;uid=sa;pwd=;Initial Catalog=pubs" );
myConnection.Open();
// Do something the myConnection.
myConnection.Close();
```

VB

```
Dim myConnection As New _
    OleDbConnection("Provider=SQLOLEDB.1;Data " + _
    "Source=localhost;uid=sa;pwd=;Initial Catalog=pubs")
myConnection.Open()
' Do something the myConnection.
myConnection.Close()
```

RECOMMENDED PRACTICE: The above connection strings are hard-coded into the source code. If at any time you need to change the connect strings (such as when the database server changes), you'll need to change the connection strings. If the connection strings are scattered all over the code, changing them will be difficult, and there's a chance you'll miss one.

The usual recommended practice is to store the connection string in the Web.config file that I discuss in detail in Chapter 7 in the section entitled "Retrieving the `Database Connection` String from Web.config." For this application, though, I use an application variable and initialize it in the Global. asax file. I chose to do it this way to give you an example of another way to store a connection string. The following code shows how to initialize an application variable in a Global.asax file:

C#

```
protected void Application_Start(Object sender,
    EventArgs e)
{
    Application["DBConnectionString"] =
      "server=localhost;uid=sa;pwd=;database=Survey";
}
```

VB

```
Sub Application_Start(ByVal sender As Object, _
    ByVal e As EventArgs)
    Application("DBConnectionString") = _
       "server=localhost;uid=sa;pwd=;database=Survey"
End Sub
```

Both managed provider connection strings look similar. In fact, the `OleDb` connection string is exactly the same as its predecessor in ADO, which should be obvious if you are familiar with programming in ADO. Now look at the differences. The SQL Server managed provider uses the private protocol called **tabular data stream** that is designed to work with SQL Server 7.0 and later. It does not use OLE DB, ADO, or ODBC. You can use an OleDb connection to SQL Server, but if you do, you will see performance degradation. The SQL Server connection also supports a variety of connection string keywords. Table 3.1 shows the OLE DB providers that are available in ADO.NET.

Command

The Command object allows direct interaction with the data through the database connection. The example shown in Listing 3.3 returns all rows from the Publishers table in Microsoft's `Pubs` database and loads them into a SqlDataReader using the Command object's `ExecuteReader()` method. The SqlDataReader enables the information to be accessed and processed accordingly.

Table 3.1 OLE DB Providers

Driver	Provider
SQLOLEDB	SQL OLE DB Provider (for SQL Server 6.5 and earlier)
MSDAORA	Oracle OLE DB Provider
JOLT	Jet OLE DB Provider

Listing 3.3 Using the Command Object

C#

```
SqlConnection myConnection  =
new SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" );
SqlCommand myCommand = new SqlCommand( "SELECT * FROM Publishers",
myConnection );

myConnection.Open();
myReader = myCommand.ExecuteReader();

while( myReader.Read() )
{
  // Do something with the data.
}

myReader.Close();
myConnection.Close();
```

VB

```
Dim myConnection as new
SqlConnection("server=localhost;uid=sa;pwd=;database=pubs")
Dim myCommand as new _
  SqlCommand("SELECT * FROM Publishers", myConnection)

myConnection.Open()
myReader = myCommand.ExecuteReader()

While myReader.Read()
  // Do something with the data.
End While

myReader.Close()
myConnection.Close()
```

In the example, the System.Data and System.Data.SqlClient namespaces must be included to get the correct SQL methods. Next, a SqlConnection is created to the `Pubs` database. A SQL `SELECT` statement and the reference to the Connection object are passed as `SqlCommand` parameters. The last declaration is a SqlDataReader that allows processing of the data fetched from the database. Finally, the connection and SqlDataReader are closed.

The example shown uses the SQL managed provider. However, if a connection to another database is required and the connection is using the OLE DB provider, then simply change the `SQL command` references to `OleDb` commands, and the remaining code will be the same.

RECOMMENDED PRACTICE: Garbage collection is non-deterministic. For this reason, you should always close ADO.NET objects, such as the SqlDataReader.

The best way to do that is in the `finally` block of a `try/catch/finally` construct, as follows:

```
// Declare objects here.
try
{
  // Open objects and use them here.
}
catch
{
}
finally
{
  // If objects are open here, close them.
}
```

DataReader

The DataReader object provides an easy and efficient way to parse a series of records, or even one record. The DataReader object behaves as a read-only, forward-only stream returned from the database, and only one record at a time is ever in memory. However, the DataReader object is not intended to handle large, complex relationships between tables and records, nor does it have the capability to pass data back to a database—a responsibility best left to the DataSet and DataRelation objects. In the previous example, the SqlDataReader was used to contain the data returned from the server.

In this example, shown in Listing 3.4, I've expanded the code to display all the data from the Authors table.

Listing 3.4 Displaying Data from the Authors Table

C#

```
SqlConnectiion myConnection = new SqlConnection("server=localhost;
uid=sa;pwd=;database=pubs" );
SqlCommand myCommand = null;
SqlDataReader myReader = null;
SqlDataReader myReader = null;

try
{
  myConnection.Open();
  myReader = myCommand.ExecuteReader();
  myCommand =  new SqlCommand( "SELECT * FROM Authors ",
    myConnection );

  Response.Write( "<table border=1>" );
  while( myReader.Read() )
  {
    Response.Write("<tr>");
    for( int i=0; i<myReader.FieldCount; i++ )
    {
      Response.Write( "<td>" + myReader[i].ToString() + "</td>" );
      Response.Write( "</tr>" );
    }
  }
  Response.Write( "</table>" );
}
catch
{
}
finally
{

  if ( myReader != null )
  {
     myReader.Close();
  }
  if( myConnection.State == ConnectionState.Open )
```

```
  {
     myConnection.Close();
  }
  }
}
```

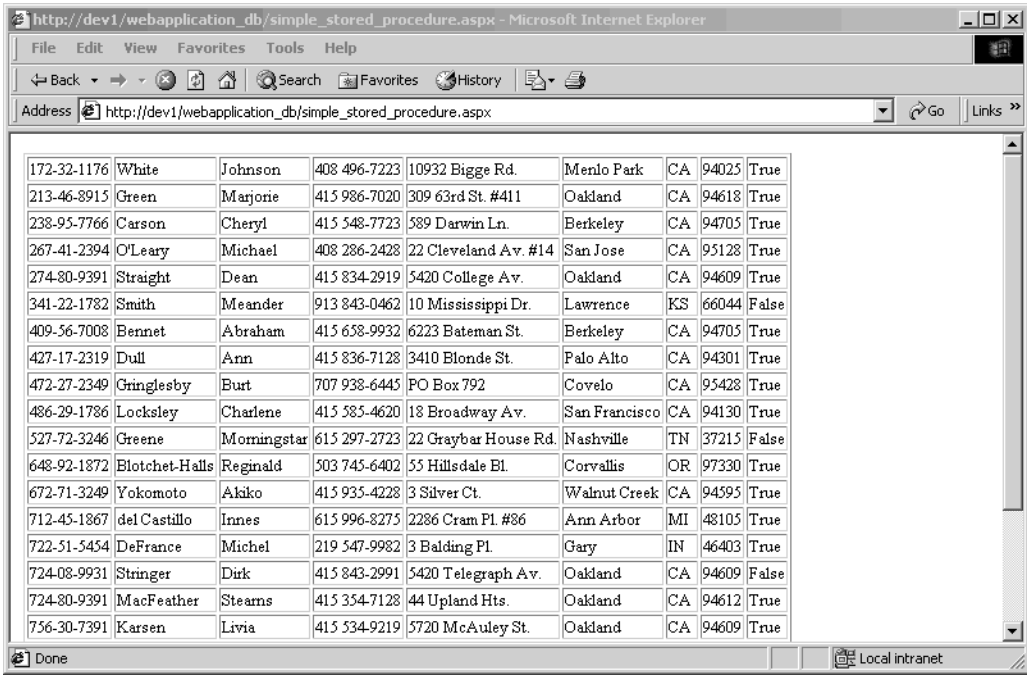
VB

```
Dim myConnection as new _
  SqlConnection("server=localhost;uid=sa;pwd=;database=pubs" )
Dim myCommand as new _
  SqlCommand( "SELECT * FROM Authors ", myConnection )
Dim myReader As SqlDataReader = nothing

Try
  myConnection.Open()
  MyReader = myCommand.ExecuteReader()

  Response.Write( "<table border=1>" )
  While myReader.Read()
    Response.Write("<tr>")
    Dim i as Integer
    For i=0 To MyReader.FieldCount-1
      Response.Write( "<td>" + myReader(i).ToString() + "</td>" )
      Response.Write( "</tr>" )
    Next
  End While
  Response.Write( "</table>" )
Catch
Finally
  If myReader <> nothing Then    myReader.Close()
  End If
  If myConnection.State = ConnectionState.Open Then
    myConnection.Close()
  End If
End Try
```

The output of this example can be seen in Figure 3.1, which creates an HTML table for displaying the data. From the code, you will first notice the `MoveNext()` method is not part of the while loop for the SqlDataReader. The SqlDataReader's `Read()` method automatically advances the cursor and initially sets the cursor to the beginning of the data. To create the table dynamically, we use the `FieldCount` property of the DataReader to deter-



172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	True
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618	True
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	94705	True
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	95128	True
274-80-9391	Straight	Dean	415 834-2919	5420 College Av.	Oakland	CA	94609	True
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	66044	False
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	94705	True
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	94301	True
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	95428	True
486-29-1786	Locksley	Charlene	415 585-4620	18 Broadway Av.	San Francisco	CA	94130	True
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	37215	False
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	97330	True
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	94595	True
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	48105	True
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	46403	True
724-08-9931	Stringer	Dirk	415 843-2991	5420 Telegraph Av.	Oakland	CA	94609	False
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts.	Oakland	CA	94612	True
756-30-7391	Karsen	Livia	415 534-9219	5720 McAuley St.	Oakland	CA	94609	True

Figure 3.1 An HTML Representation of the Authors Table

mine the number of columns, which allows sequencing through each column to get its value. Once all the data has been parsed, the `Read()` method will return a `null`. An alternate method to use to check for more data is the `HasMoreResults` property. This method is useful if you need to check for more records within a `loop` condition without advancing the record pointer.

CAUTION: One of the most common errors my students make is using a data reader when they're looking only for a single record. They almost always try to retrieve data from the DataReader before they call the `Read()` method. Remember: You must call the `Read()` method before you get any data from a DataReader.

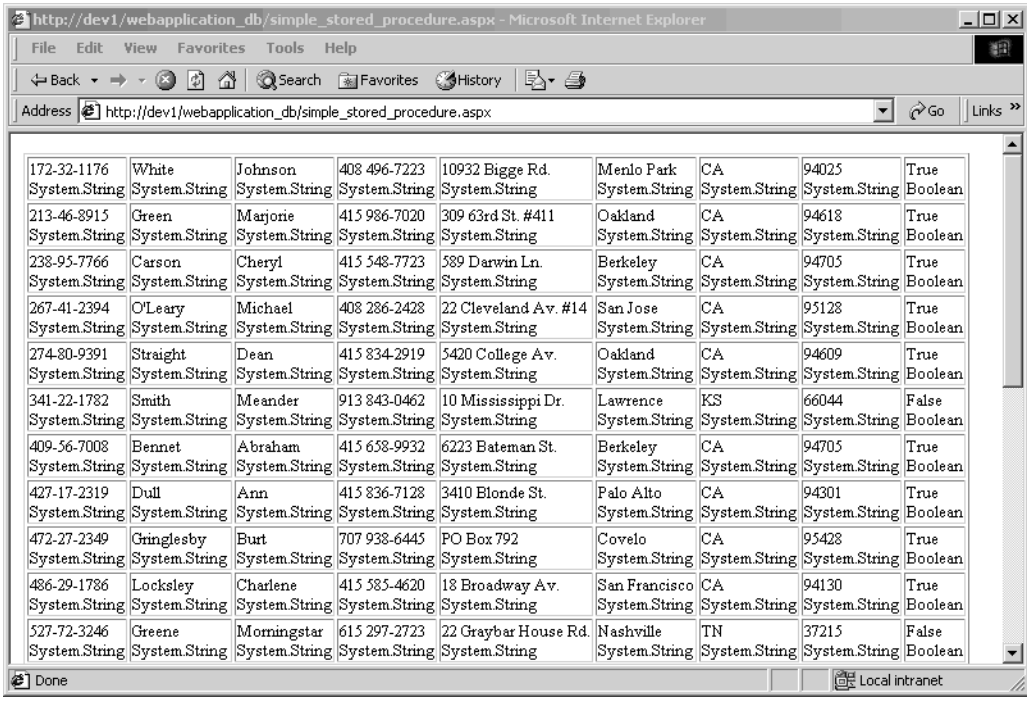
The DataReader also contains a variety of `Get` methods that enable you to access field values, such as `GetInt()`, `GetDouble()`, `GetInt32()`, and `GetString()`, in native formats. To determine which one to use, the `GetFieldType` property can be called to get the appropriate column

type. Then the correct `Get` method can be called to fetch the column data in its native format. To see the property type of each column, I could add the following code to my write statement:

```
myReader[i].GetFieldType.ToString();
```

Figure 3.2 shows the column-type name added to the output of the previous example by using the added statement.

The DataReader (unlike classic ADO) does not use the `MoveFirst()`, `MoveNext()`, and `MoveLast()` commands, or the `EOF` property. The initial call to the DataReader object's `Read()` command positions the record cursor at the beginning of the data and advances it after each subsequent call until all the data is processed. After all the data is processed, the `Read()` method returns a Boolean value. Moving the cursor back to the beginning is not permitted—remember, the DataReader is forward only. The DataSet object now provides bi-directional movement through the data.



172-32-1176 System.String	White System.String	Johnson System.String	408 496-7223 System.String	10932 Bigge Rd. System.String	Menlo Park System.String	CA System.String	94025 System.String	True Boolean
213-46-8915 System.String	Green System.String	Marjorie System.String	415 986-7020 System.String	309 63rd St. #411 System.String	Oakland System.String	CA System.String	94618 System.String	True Boolean
238-95-7766 System.String	Carson System.String	Cheryl System.String	415 548-7723 System.String	589 Darwin Ln. System.String	Berkeley System.String	CA System.String	94705 System.String	True Boolean
267-41-2394 System.String	O'Leary System.String	Michael System.String	408 286-2428 System.String	22 Cleveland Av. #14 System.String	San Jose System.String	CA System.String	95128 System.String	True Boolean
274-80-9391 System.String	Straight System.String	Dean System.String	415 834-2919 System.String	5420 College Av. System.String	Oakland System.String	CA System.String	94609 System.String	True Boolean
341-22-1782 System.String	Smith System.String	Meander System.String	913 843-0462 System.String	10 Mississippi Dr. System.String	Lawrence System.String	KS System.String	66044 System.String	False Boolean
409-56-7008 System.String	Bennet System.String	Abraham System.String	415 658-9932 System.String	6223 Bateman St. System.String	Berkeley System.String	CA System.String	94705 System.String	True Boolean
427-17-2319 System.String	Dull System.String	Ann System.String	415 836-7128 System.String	3410 Blonde St. System.String	Palo Alto System.String	CA System.String	94301 System.String	True Boolean
472-27-2349 System.String	Gringlesby System.String	Burt System.String	707 938-6445 System.String	PO Box 792 System.String	Covelo System.String	CA System.String	95428 System.String	True Boolean
486-29-1786 System.String	Locksley System.String	Charlene System.String	415 585-4620 System.String	18 Broadway Av. System.String	San Francisco System.String	CA System.String	94130 System.String	True Boolean
527-72-3246 System.String	Greene System.String	Morningstar System.String	615 297-2723 System.String	22 Graybar House Rd. System.String	Nashville System.String	TN System.String	37215 System.String	False Boolean

Figure 3.2 An HTML Representation of the Authors Table with Column Data Types Shown

Parameter Binding with SQL Commands

Another feature of the SqlCommand object is its ability to easily bind parameter data for SQL statements and stored procedures. Each parameter has four key pieces of information: the name, the type, its data size, and the direction of the parameter.

For the SQL Server Managed provider, the parameter construction uses actual names of the parameters, just like regular T-SQL syntax uses. For example, the following code contains a single `ID` parameter that is passed to the `SELECT` command:

```
SELECT * FROM Authors WHERE au_id=@ID
```

To return values, I need to add parameters to the `SELECT` statement:

```
SELECT @Fname=au_fname, @Lname=au_lname FROM Authors WHERE au_id=@ID
```

Now I have one input and two output parameters. The code to bind the parameters to the `SELECT` command starts with a standard SQL connection, followed by the SQL `SELECT` statement, and finally a set of parameter bindings. The following code illustrates how the binding process works:

C#

```
SqlConnection myConnection =
new SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" );
SqlCommand myCommand =
  new SqlCommand(
    "SELECT @Fname=au_fname, @Lname=au_lname_FROM " +
    "Authors WHERE au_id=@ID", myConnection );

myCommand.Parameters.Add( "@ID", SqlDbType.VarChar, 11 );
myCommand.Parameters["@ID"].Direction = ParameterDirection.Input;
myCommand.Parameters["@ID"].Value = "172-32-1176";

myCommand.Parameters.Add( "@Fname", SqlDbType.VarChar, 20 );
myCommand.Parameters["@Fname"].Direction = ParameterDirection.Output;

myCommand.Parameters.Add( "@Lname", SqlDbType.VarChar, 40 );
myCommand.Parameters["@Lname"].Direction = ParameterDirection.Output;

myConnection.Open();
myCommand.Execute();
```

```
Response.Write( "First Name " + myCommand.Parameters["@Fname"].Value.
ToString() + "<br>" );
Response.Write( "Last Name " + myCommand.Parameters["@Lname"].Value.
ToString() );
myConnection.Close();
```

VB

```
Dim myConnection as new _
  SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" )
Dim myCommand as New _
  SqlCommand( "SELECT @Fname=au_fname, @Lname=au_lname FROM " + _
    "Authors WHERE au_id=@ID", myConnection )

myCommand.Parameters.Add( "@ID", SqlDbType.VarChar, 11 )
myCommand.Parameters("@ID").Direction = ParameterDirection.Input
myCommand.Parameters("@ID").Value = "172-32-1176"

myCommand.Parameters.Add( "@Fname", SqlDbType.VarChar, 20 )
myCommand.Parameters("@Fname").Direction = ParameterDirection.Output

myCommand.Parameters.Add( "@Lname", SqlDbType.VarChar, 40 )
myCommand.Parameters("@Lname").Direction = ParameterDirection.Output

myConnection.Open()
myCommand.Execute()
Response.Write( "First Name " + _
  myCommand.Parameters("@Fname").Value.ToString() + "<br>" )
Response.Write( "Last Name " + _
  myCommand.Parameters("@Lname").Value.ToString() )
myConnection.Close()
```

Notice in the example that the names of the parameters must match the names declared in the SQL `SELECT` statement. Otherwise, the parameters do not match up correctly. The data types are standard SQL types.

RECOMMENDED PRACTICE: If your query will return only a single record (rowset), then using a SqlDataReader object into which the data will be retrieved is unnecessary overhead. Using bound parameters instead will cause your code to execute faster.

NOTE: In the examples I use in this book in which parameters are added to a SqlCommand, I access the parameters by name. For instance, I might call

a parameter `@ID` or `@Name`. You can alternatively use ordinals, which are zero-based numbers that identify a particular parameter from the collection. Using ordinals as opposed to names will give you a performance boost because a name lookup isn't performed during parameter access. I have been down this road, though, too many times to advise you to use ordinals. I have seen my students get into too many situations in which the ordinals got mixed up, and they ended up using the wrong ones. Consider this choice carefully. If performance is important, use ordinals; otherwise, keep with names.

The size value is necessary only for fields that contain an actual size. For values such as numeric, this value can be omitted. Finally, the direction value indicates how the parameter will be used. Table 3.2 shows the four different direction values.

Stored Procedures and Parameter Binding

Calling stored procedures and binding parameter data work much like the SQL `EXEC` statement. This section shows how to call stored procedures, pass parameters in and out, and return the exit value of the stored procedure. I will create a stored procedure, pass values in and out of the procedure, and access the stored procedure's return value.

First, I have to create a stored procedure that does all this. For this example I'll take the `SELECT` statement used in the "Parameter Binding with SQL Commands" section and create a stored procedure in the Microsoft SQL Server `Pubs` database, as shown below.

```
Create Procedure sp_GetAuthor
    @ID varchar(11),
    @Fname varchar(20) output,
    @Lname varchar(40) output
```

Table 3.2 Direction Values for Parameterized Queries

Direction	Description
`Input`	The parameter is an input parameter.
`InputOutput`	The parameter is capable of both input and output.
`Output`	The parameter is an output parameter.
`ReturnValue`	The parameter represents a return value.

```
AS
    SELECT @Fname = NULL
    SELECT @LName = NULL
    SELECT @Fname=au_fname, @Lname=au_lname FROM authors WHERE au_id=@ID
    if(@Fname IS NULL)
      return -100
    else
      return 0
```

To illustrate the return value parameter, I have included an `error` condition in the stored procedure. When the `SELECT` statement fails, a `-100` is returned after the procedure checks the `@Fname` value for `null`. The initialization of the two output parameters is a precaution in the event a value is passed.

RECOMMENDED PRACTICE: Use `nVarChar` whenever possible. It looks like this may not make sense for the `Pubs` database, but `nVarChar` is better in most cases. Essentially, using `NVarChar` makes internationalization much easier and is inexpensive to do up front.

If you don't use `nVarChar` whenever you can, then you must have custom installations of SQL Server that use a different alphabet for each language you need to support. I've seen successful sales or company growth turn this simple mistake into something very expensive.

The SQL Server parameter binding works exactly the same as the SQL command statement. The only parameter addition is the binding to reference the stored procedure's return value.

C#

```
SqlConnection myConnection =
      new SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" );
SqlCommand myCommand =
  new SqlCommand("sp_GetAuthor", myConnection );
myCommand.CommandType = CommandType.StoredProcedure;

myCommand.Parameters.Add( "@ID", SqlDbType.VarChar, 11 );
myCommand.Parameters["@ID"].Direction = ParameterDirection.Input;
myCommand.Parameters["@ID"].Value = List1.SelectedItem.Text;
```

```
myCommand.Parameters.Add( "@Fname", SqlDbType.VarChar, 20 );
myCommand.Parameters["@Fname"].Direction = ParameterDirection.Output;

myCommand.Parameters.Add( "@Lname", SqlDataType.VarChar, 40 );
myCommand.Parameters["@Lname"].Direction = ParameterDirection.Output;

myCommand.Parameters.Add( "RETURN_VALUE", SqlDbType.Int );
myCommand.Parameters["RETURN_VALUE"].Direction =
ParameterDirection.ReturnValue;

myConnection.Open();
myCommand.ExecuteNonQuery();

string strFirstName = myCommand.Parameters["@Fname"].Value.ToString();
string strLastName = myCommand.Parameters["@Lname"].Value.ToString();
strError = myCommand.Parameters["RETURN_VALUE"].Value.ToString();

myConnection.Close();
```

VB

```
Dim myConnection as _
 new SqlConnection( "server=localhost;uid=sa;pwd=;database=pubs" )
Dim myCommand as new SqlCommand("sp_GetAuthor", myConnection )
myCommand.CommandType = CommandType.StoredProcedure

myCommand.Parameters.Add( "@ID", SqlDbType.VarChar, 11 )
myCommand.Parameters("@ID").Direction = ParameterDirection.Input
myCommand.Parameters("@ID").Value = List1.SelectedItem.Text

myCommand.Parameters.Add( "@Fname", SqlDbType.VarChar, 20 )
myCommand.Parameters("@Fname").Direction = ParameterDirection.Output

myCommand.Parameters.Add( "@Lname", SqlDataType.VarChar, 40 )
myCommand.Parameters("@Lname").Direction = _
  ParameterDirection.Output

myCommand.Parameters.Add( "RETURN_VALUE", SqlDbType.Int )
myCommand.Parameters("RETURN_VALUE").Direction = ParameterDirection.
ReturnValue

myConnection.Open()
myCommand.ExecuteNonQuery()
```

```
string strFirstName = myCommand.Parameters("@Fname").Value.ToString()
string strLastName = myCommand.Parameters("@Lname").Value.ToString()
strError = myCommand.Parameters("RETURN_VALUE").Value.ToString()

myConnection.Close()
```

RECOMMENDED PRACTICE: Stored procedures are almost always preferred over ad hoc queries in your code. The following list summarizes the reasons:

- Stored procedures execute faster than ad hoc SQL because SQL Server has already compiled the procedures and created a plan.
- Stored procedures give you a single place to make changes or fix bugs when queries need changes.
- Stored procedures offer an abstraction to the application code under circumstances in which data access is separated from code.

The Survey Application

The application that's featured in this chapter is a Survey application, and it demonstrates the use of ADO.NET. The program in Chapter 2 used ADO.NET to perform database access, but in this chapter and with this application, we'll take time to explain the ADO.NET functionality in greater detail.

The Survey application contains three distinct parts. One is the administrative part, in which questions can be edited, added, and deleted. Another part consists of the code that generates the survey questions and answers based on some parameters. And the third part of the demo application is the main screen on which the actual survey data is shown, and user interaction mechanisms are provided.

NOTE: The Survey application can be viewed from the www.ASPNet-Solutions.com Web site. From the main page of the site, go to the Chapter Examples page. Then click on the Chapter 3 link. This page offers the capability to run the Survey application.

You can go directly to the www.ASPNet-Solutions.com/Chapter_3.htm page for the Survey application link.

The C# and VB source code and the backed-up database can be downloaded from the Chapter 3 page.

The Administrative Code

If you download the code from the Web site, you'll find all of the administrative functionality in administer.aspx.cs or administer.aspx.vb (depending on whether you are using the C# or VB version). This source code module contains all the methods that perform the administrative functions for the application. Table 3.3 shows what the methods are and describes the purpose of each.

When the program runs, you'll see a way to log in to the administrative section, and you'll see a survey appear in the right side of the screen, as shown in Figure 3.3.

PopulateQuestionList() and Page_Load()

The `PopulateQuestionList()` method is called from the `Page_Load()` method (which executes during the initial page-loading sequence). This method needs to be called only the first time that the page is loaded. When a page load is a post back, the QuestionList ListBox object is already populated because its state persists in the `VIEWSTATE` hidden field. In many situations, not calling `PopulateQuestionList()` in response to post backs will save the server some processor time.

In some situations, though, the `PopulateQuestionList()` method is called in response to a user-generated event. Examples of this are when a user adds or deletes a question. In these cases, the QuestionList object needs to be repopulated.

RECOMMENDED PRACTICE: Make sure your applications don't repopulate user interface objects for post backs unless it's absolutely necessary. For objects that query a database for their contents but never change throughout the life cycle of the application page requests, repopulating would represent an unnecessary burden on the server.

Check the `IsPostback` property to see whether the current request is a post back. The property will be `true` if it is.

We talked earlier in the chapter about SqlConnection objects. These objects will be used throughout the entire Survey application to connect to the database. The first thing that's done in Listing 3.5 (on page 87) is to create a SqlConnection object. Its one and only parameter is the connection string, which is contained in the Global.asax. Making any changes to this code is an easy matter because only one place must be edited for

Table 3.3 The Administrative Methods Found in administer.aspx.cs and administer.aspx.vb

Method	Listing	Description
`PopulateQuestionList`	3.5	This method populates the QuestionList ListBox object with all questions that are found in the database. It can optionally populate the CategoryList DropDownList object if the `bPopulateCategoryList-Also` flag is `true`.
`UpdateButton_Click`	3.6	This is the event handler that is fired when the Update This Question button is clicked. This method updates the database with the information that's in the user interface, such as the Question Text and the Answers.
`DeleteButton_Click`	3.7	This is the event handler that is fired when the Delete button is clicked. This method uses the QuestionID Session—`Session["QuestionID"]` for C# and `Session("QuestionID")` for VB—variable so that it knows which question number to delete from the database.
`AddCategory_Click`	3.8	This is the event handler that is fired when the Add It button is clicked. It takes the value that is in the NewCategory TextField object and adds it to the database.
`AddButton_Click`	3.9	This is the event handler that is fired when the Add New Question button is clicked. It essentially clears the editable text fields and sets other values to `-1`, which indicates there is no currently selected question.
`MainButton_Click`	3.9	This is the event handler that is fired when the Main Survey Page button is clicked. It simply redirects to the main Survey page.
`QuestionList_Selected IndexChanged`	3.10	This is the event handler that is fired when the QuestionList ListBox object detects a change in the selection index. This method populates the editable items on the page with the question information.

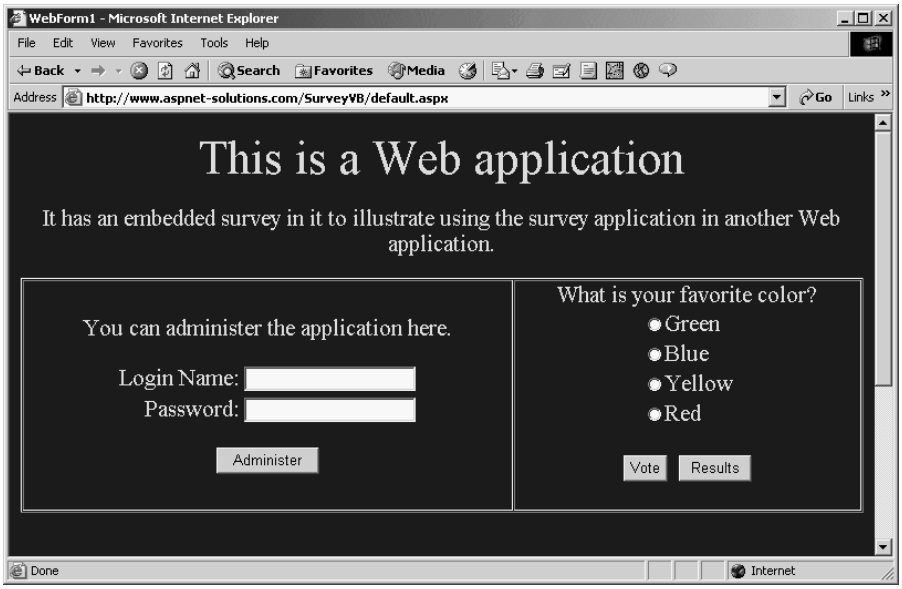


Figure 3.3 The Survey Application Lets Users Log In to the Administrative Functionality and Offers Them a Survey Item.

changes to take effect (this was mentioned as a recommended practice earlier in the chapter).

RECOMMENDED PRACTICE: It is always a bad idea to leave database connections open longer than necessary. I once had a student who opened a connection in the Global.asax. The connection stayed open until the application shut down. Several problems are inherent with doing this. The first is that a connection can have only one open DataReader, and if more than one user requests a page that causes simultaneous readers to be open, at least one exception will be thrown.

In addition, open connections consume resources. This means that if you leave a connection open for a long time, and you have a large number of users accessing the database, the server will have a large portion of its resources allocated to database connections.

Another issue is deploying the Web application in a Web farm. In these cases, you might really slow down SQL Server, connection pooling is the best bet. You can set the connection pool size in the database connection string as follows:

```
server=localhost;uid=sa;pwd=;database=pubs;Pooling=true;Max
Pool Size=500
```

After the database connection has been opened with the `Open()` method, a SqlCommand object is created that will call the `sp_QuestionList` stored procedure. This stored procedure returns a recordset containing all the questions in the database (whether or not they are enabled). The `sp_QuestionList` stored procedure follows.

```
CREATE PROCEDURE sp_QuestionList
AS
    SELECT Text FROM Questions ORDER BY Text
GO
```

Once a recordset has been obtained from the `sp_QuestionList` stored procedure by calling the SqlCommand object's `ExecuteReader()` method, the recordset will be bound to the QuestionList ListBox object. The `QuestionList DataTextField` and `DataValueField` property values are set so that the data-binding process knows to bind using the Text field that's in the recordset. The last two things to be done are to set the `DataSource` property to the SqlDataReader object, and to call the `DataBind()` method.

A flag named bPopulateCategoryListAlso indicates whether the CategoryList DropDownList object should be populated. Population will need to happen only once at initial page load (not for post backs).

To retrieve the list of categories, the `sp_CategoryList` stored procedure is called. To do this, we almost literally repeat the process used to retrieve the question list. The only difference is that we set the SqlCommand object to access the `sp_CategoryList` stored procedure. This stored procedure is shown below.

```
CREATE PROCEDURE sp_CategoryList
AS
    SELECT Text FROM Categories ORDER BY ID
GO
```

The `DataTextField` and `DataValueField` properties are set, the `DataSource` property is set, and the `DataBind()` method is called. This completes the process of populating the CategoryList object.

NOTE: SqlDataReader objects must always be closed before you retrieve another SqlDataReader object because you can't open more than one reader per connection. Not closing the SqlDataReader objects has two negative results: Resources won't be released, and an exception will be thrown. The SqlConnection object won't allow more than one simultaneous open SqlDataReader.

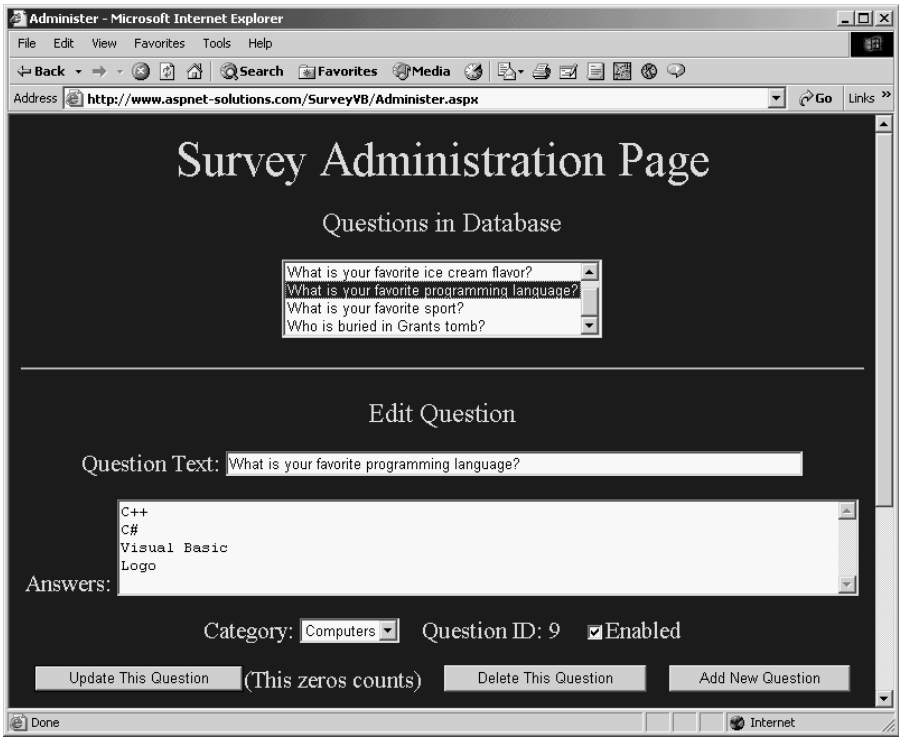


Figure 3.4 Selecting Questions in the QuestionList Object Populates the Fields.

The last thing to note, in Listing 3.5, is that code to close the database connection is in the `catch` block. This location is in case the database connection opens successfully, but, at some point after it has been opened, an exception is thrown. Figure 3.4 shows the application during execution.

Listing 3.5 The `PopulateQuestionList()` Method

```
Private Sub PopulateQuestionList(ByVal bPopulateCategoryListAlso As
Boolean)
   ' Create the connection object
   Dim myConnection As New _
     SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        myConnection.Open()
```

```
        ' Create the command object specifying the sp_QuestionList
        '   stored procedure, and set the CommandType property to
        '   CommandType.StoredProcedure.
        Dim myCommand As New SqlCommand("sp_QuestionList", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        ' Retrieve a SqlDataReader by calling the ExecuteReader()
        '   method. Then, databind the recordset with the
        '   QuestionList object. It's important to specify the
        '   column name for the
        '   DataTextField and DataValueField properties.
        Dim reader As SqlDataReader = myCommand.ExecuteReader()
        QuestionList.DataTextField = "Text"
        QuestionList.DataValueField = "Text"
        QuestionList.DataSource = reader
        QuestionList.DataBind()
        reader.Close()

        ' If the Boolean variable is true, we'll need to populate
        '   the CategoryList object.
        If bPopulateCategoryListAlso Then
            myCommand = New SqlCommand("sp_CategoryList", _
              myConnection)
            myCommand.CommandType = CommandType.StoredProcedure
            reader = myCommand.ExecuteReader()
            CategoryList.DataTextField = "Text"
            CategoryList.DataValueField = "Text"
            CategoryList.DataSource = reader
            CategoryList.DataBind()
            reader.Close()
        End If
    Catch ex As Exception
        Message.Text = ex.Message.ToString()
    Finally
        If myConnection.State = ConnetionState.Open Then
          myConnection.Close()
        End If
    End Try

End Sub

Private Sub Page_Load(ByVal sender As System.Object, ByVal e As _
  System.EventArgs) Handles MyBase.Load
```

```
    If Not IsPostBack Then
        If Session("CurrentQuestionID") = Nothing Then
            Session("CurrentQuestionID") = -1
        End If
        PopulateQuestionList(True)
    End If
End Sub
```

The `UpdateButton_Click()` Method

Users click the `UpdateButton_Click()` method on the update of a question's information. The items that are saved are the question text, the answers, the category, and whether the question is enabled.

The method starts by making sure the Question and Answers TextField objects contain data. Updating the question would be pointless without data in these fields. If either field is empty, a message is shown to the user (by being placed in the Message Label object). After the error message is set, the method then ends when a Return statement is encountered.

After the method checks for data in the Question and Answers objects, a SqlConnection object is created. This connection is used for all database access in this method.

The question ID (which is a unique key for the Question table in the database) is stored in a session variable. You will see that an integer variable named `nID` is assigned with the integer value in the `Session("CurrentQuestionID")` variable.

The database connection is opened with a call to the `Open()` method. A SqlCommand object named myCommand is created, and the `sp_UpdateQuestionInfo` stored procedure is specified as the command that will be performed. This `CommandType` property of the SqlCommand object is set to `StoredProcedure`. You can see the `sp_UpdateQuestionInfo` stored procedure below.

```
CREATE PROCEDURE sp_UpdateQuestionInfo
  @Text varchar(254),
  @CategoryID int,
  @Enabled int,
  @ID as int output
AS

  if( @ID <> -1 )
    begin
```

```
      DELETE Answers WHERE QuestionID=@ID
      UPDATE Questions SET
        Text=@Text,CategoryID=@CategoryID,Enabled=@Enabled
        WHERE ID=@ID
    end
  else
    begin
      INSERT INTO Questions (Text,CategoryID,Enabled)
        VALUES (@Text,@CategoryID,@Enabled)
          SELECT @ID=@@IDENTITY
    end
GO
```

The stored procedure expects four parameters: the question text (variable named `@Text`), the category ID (variable named `@CategoryID`), an indicator of whether the question is enabled (variable `@Enabled`), and the question ID (variable `@ID`). The question ID can be a valid question ID or `-1`, which indicates that this is a new question and should be added rather than updated.

The database provides unique question IDs because the `ID` field in the Questions table is an identity column. This arrangement means that the database will enforce uniqueness, and as a matter of fact will assign the ID value at the time a record is created. SQL Server makes available a mechanism whereby it is easy to get an identity column after a record has been created. The following code shows how T-SQL or a stored procedure can get an identity column into a parameter named `@ID`:

```
INSERT INTO SomeTable (FieldName1,FieldName2) VALUES ('Data1',
  'Data2')  SELECT @ID=@@IDENTITY
```

Once the four parameters have been set up, a call to the `ExecuteNonQuery()` method is made. This updates or inserts the record, and for new records returns a unique question ID. This unique ID is retrieved with the following code:

```
If nID = -1 Then
    nID = Convert.ToInt32(myCommand.Parameters("@ID").Value)
End If
```

With the question added, we'll need to add the answers (or survey choices). The Answers TextField object contains answers that are all sepa-

rated by carriage return/line feed (CR/LF) pairs. This is the perfect opportunity to use the String object's `Split()` method. The `Split()` method can easily find separator characters and split a string into an array of strings.

The only difficulty here is that our separator is a pair of characters, not the single character that the `Split()` method needs. For this reason, we'll use a newly created string that replaces the `CR/LF` pairs with `'|'` characters. This only works when there is no `'|'` symbol in the answer string, so make sure that your survey administrators understand this limitation. We can then easily use the `Split()` method and specify the `'|'` as the separator character. The following code shows how a single string of four answers (separated by three `CR/LF` pairs) is split into four substrings:

```
   ' Here's our initial string.
   Dim strData as string = "Red"+ vbCrLf + "Green" + vbCrLf + "Blue" + _
   vbCrLf + "Yellow"
' Here's the new string with '|' replacing the CR/LF pairs.
Dim strNewData as string = strData.Replace( vbCrLf, "|" )
' Here we perform the split.
Dim strAnswers as string() = _
  strNewData.Split( New Char {Chr(124)}, 100 )

' Now we'll loop through and use each substring.
Dim i as Integer
For i=0 to strAnswers.Length - 1
  ' Now do something with strAnswers(i)
Next
```

A stored procedure named `sp_AddAnswer` takes a question ID, the order of the answer (such as 0, 1, or 2), and the answer text and creates an answer in the database that can be used later when the question data is retrieved. The stored procedure can be seen below.

```
CREATE PROCEDURE sp_AddAnswer
    @QuestionID int,
    @Text varchar(254),
    @Ord int
AS
    INSERT INTO Answers (Text,QuestionID,Ord) VALUES
       (@Text,@QuestionID,@Ord)
GO
```

After the parameters (`@Text`, `@QuestionID`, and `@Ord`) are added to the SqlCommand object, a loop is used to treat each substring separately. Each substring is set into the `@Text` parameter, along with the `@Ord` parameter. A call to the stored procedure is made, thus storing the data in the database. Finally, the database connection is closed and a call to the `PopulateQuestionList()` method is made.

Listing 3.6 The `UpdateButton_Click()` Method

```
Private Sub UpdateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles UpdateButton.Click
    If Question.Text.Length = 0 Or Answers.Text.Length = 0 Then
        Message.Text = "You need text in the answers field."
        Return
    End If

    Dim myConnection As New _
SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        Dim nID As Integer = _
          Convert.ToInt32(Session("CurrentQuestionID"))

        myConnection.Open()
        Dim myCommand As New SqlCommand("sp_UpdateQuestionInfo", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@Text", _
          SqlDbType.VarChar, 254))
        myCommand.Parameters("@Text").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@Text").Value = Question.Text

        myCommand.Parameters.Add(New SqlParameter("@CategoryID", _
          SqlDbType.Int))
        myCommand.Parameters("@CategoryID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@CategoryID").Value = _
          CategoryList.SelectedIndex

        myCommand.Parameters.Add(New SqlParameter("@Enabled", _
          SqlDbType.Int))
```

```
myCommand.Parameters("@Enabled").Direction = _
  ParameterDirection.Input
myCommand.Parameters("@Enabled").Value = 0

If Enabled.Checked Then
    myCommand.Parameters("@Enabled").Value = 1
End If

myCommand.Parameters.Add(New SqlParameter("@ID", _
  SqlDbType.Int))
myCommand.Parameters("@ID").Direction = _
  ParameterDirection.InputOutput
myCommand.Parameters("@ID").Value = nID

myCommand.ExecuteNonQuery()

If nID = -1 Then
    nID =
     Convert.ToInt32(myCommand.Parameters("@ID").Value)
    QuestionID.Text = Convert.ToString(nID)
End If

Dim strWork As String = Answers.Text.Replace(vbCrLf, "|")
Dim strAnswers As String() = _
  strWork.Split(New Char() {Chr(124)}, 100)
myCommand = New SqlCommand("sp_AddAnswer", myConnection)
myCommand.CommandType = CommandType.StoredProcedure
myCommand.Parameters.Add(New SqlParameter("@Text", _
  SqlDbType.VarChar, 254))
myCommand.Parameters("@Text").Direction = _
  ParameterDirection.Input

myCommand.Parameters.Add(New SqlParameter("@QuestionID", _
  SqlDbType.Int))
myCommand.Parameters("@QuestionID").Direction = _
  ParameterDirection.Input

myCommand.Parameters.Add(New SqlParameter("@Ord", _
  SqlDbType.Int))
myCommand.Parameters("@Ord").Direction = _
  ParameterDirection.Input

Dim i As Integer
For i = 0 To strAnswers.Length - 1
```

```
            If strAnswers(i).Length > 0 Then
                myCommand.Parameters("@Text").Value = _
                  strAnswers(i)
                myCommand.Parameters("@QuestionID").Value = nID
                myCommand.Parameters("@Ord").Value = i
                myCommand.ExecuteNonQuery()
            End If
        Next

        myConnection.Close()

        PopulateQuestionList(False)

    Catch ex As Exception
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
        Message.Text = ex.Message.ToString()
    End Try
End Sub
```

The `DeleteButton_Click()` Method

Questions can be deleted by clicking the Delete button. When users click the Delete button, the code in Listing 3.7 is called. The question ID is retrieved from the `Session("CurrentSessionID")` variable and stored in a local integer variable named `nID`. If the question ID is negative, this means no question is currently selected and therefore the user can't delete the question. A message is placed in the Message Label object indicating this condition, and the method is ended with a `Return` command.

If, however, the current question has a valid ID (greater than or equal to zero), the process moves forward to delete the question. First, a connection to the database is created and opened.

A SqlCommand object is then created that specifies that `sp_DeleteQuestion` stored procedure. This stored procedure takes a single parameter that represents the question ID, and deletes the question and all of its related answers. This stored procedure can be seen below.

```
CREATE PROCEDURE sp_DeleteQuestion
    @ID as int
AS
```

```
        DELETE Answers WHERE QuestionID=@ID
        DELETE Questions WHERE ID=@ID
    GO
```

A call is made to the `ExecuteNonQuery()` method that performs the question-delete operation. The connection is closed, and several of the application variables, such as `Session("CurrentQuestionID")`, are set to indicate that there is no currently selected question.

Listing 3.7 The `DeleteButton_Click()` Method

```
Private Sub DeleteButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button2.Click
    Dim nID As Integer = _
      Convert.ToInt32(Session("CurrentQuestionID"))
    If nID < 0 Then
       Message.Text = _
   "There is not a valid question that is currently being edited."
        Return
    End If

    Dim myConnection As New _
SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        myConnection.Open()
        Dim myCommand As New SqlCommand("sp_DeleteQuestion", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@ID", _
         SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
         ParameterDirection.Input
        myCommand.Parameters("@ID").Value = nID

        myCommand.ExecuteNonQuery()
        myConnection.Close()

        QuestionList.SelectedIndex = -1
        Session("CurrentQuestionID") = -1
        Enabled.Checked = True
        QuestionID.Text = ""
```

```
        PopulateQuestionList(False)

    Catch ex As Exception
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
        Message.Text = ex.Message.ToString()
    End Try
End Sub
```

The `AddCategoryButton_Click()` Method

Users can add to the list of categories if they don't find what they want. To do this, they simply enter a category into the New Category editable text field (which is type EditBox named NewCategory), and click the Add It button. This action invokes the `AddCategoryButton_Click()` method shown in Listing 3.8. This code takes the text in the NewCategory object and sends it to a stored procedure named `sp_AddCategory`, which is shown below.

```
CREATE PROCEDURE sp_AddCategory
    @Text varchar(254)
AS
    INSERT INTO Categories (Text) VALUES (@Text)
GO
```

This code follows the pattern that we've seen thus far: Create a database connection and open it (using a SqlConnection object), create a Command object (using a SqlCommand object), set up the parameters that the stored procedure expects (by using the SqlCommand object's Parameters collection), and execute the stored procedure (with the `ExecuteNonQuery()` method).

The only thing added to the basic pattern is that the newly created category's text is added to the CategoryList DropDownList object so that it is available to the user for selection.

Listing 3.8 The `AddCategoryButton_Click()` Method

```
Private Sub AddCategoryButton_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) _
        Handles Button5.Click
```

```
    If NewCategory.Text.Length > 0 Then
     Dim myConnection As New _
SqlConnection(Convert.ToString(Application("DBConnectionString")))

        Try
            myConnection.Open()
            Dim myCommand As New SqlCommand("sp_AddCategory", _
              myConnection)
            myCommand.CommandType = CommandType.StoredProcedure

            myCommand.Parameters.Add(New SqlParameter("@Text", _
              SqlDbType.VarChar, 254))
            myCommand.Parameters("@Text").Direction = _
              ParameterDirection.Input
            myCommand.Parameters("@Text").Value = NewCategory.Text

            myCommand.ExecuteNonQuery()

            Message.Text = "New category: '" + _
               NewCategory.Text + _
               "' was added."
            Dim item As New ListItem(NewCategory.Text)
            CategoryList.Items.Add(item)
        Catch ex As Exception
            Message.Text = ex.Message.ToString()
        Finally
            If myConnection.State = ConnectionState.Open Then
                myConnection.Close()
            End If
        End Try
    End If
    NewCategory.Text = ""
End Sub
```

The `AddButton_Click()` and `MainButton_Click()` Methods

Two short and simple methods named `AddButton_Click()` and `MainButton_Click()` can be seen in Listing 3.9. The `AddButton_Click()` method is triggered in response to the user clicking on the Add New Question button. The `AddButton_Click()` method sets the `Session("CurrentQuestionID")` variable to `-1` to indicate no currently selected question, clears the TextBox objects, deselects any question in the

QuestionList object by setting its `SelectedIndex` property to `-1`, and then sets the `Enabled` check so that it is on.

The `MainButton_Click()` method just redirects users to the Survey application's main page.

Listing 3.9 The `AddButton_Click()` and `MainButton_Click()` Methods

```
Private Sub AddButton_Click(ByVal sender As System.Object, _
  ByVal e As _
  System.EventArgs) Handles Button3.Click
    Session("CurrentQuestionID") = -1
    Question.Text = ""
    Answers.Text = ""
    QuestionList.SelectedIndex = -1
    Enabled.Checked = True
End Sub

Private Sub MainButton_Click(ByVal sender As System.Object, _\
  ByVal e As _
  System.EventArgs) Handles Button1.Click
    Response.Redirect("default.aspx")
End Sub
```

The `QuestionList_SelectedIndexChanged()` Method

A good bit of code executes when the user selects a question in the QuestionList object, as you can see in Listing 3.10. The purpose of this code is to find all the related data and populate all the fields on the page so that questions can be edited.

An interesting thing happens at the top of the `QuestionList_SelectedIndexChanged()` method. It declares a ListBox object named `lb` because that is the object type for which this event handler was created. The `lb` variable is then set to reference the Sender object that was passed into this method.

In C#, the declared object must be cast as a ListBox object, as follows:

```
ListBox lb = (ListBox) sender;
```

With a reference to the ListBox object, the text for the selected question can be retrieved. We'll eventually use this text as one of the stored procedure parameters.

As with most of the methods in this source-code module, a connection to the database is created and opened. A Command object specifying the `sp_QuestionInfoFromText` stored procedure is created. The `sp_QuestionInfoFromText` can be seen below.

```
CREATE PROCEDURE sp_QuestionInfoFromText
    @Text varchar(254),
    @ID int output,
    @CategoryID int output,
    @Enabled int output
AS
    SELECT @ID=ID,@CategoryID=CategoryID,@Enabled=Enabled FROM
        Questions WHERE Text=@Text

    if( @ID IS NULL )
        SELECT @ID = -1
GO
```

Four parameters must be created and set up for the sp_Question-InfoFromText stored procedure. These parameters are `@Text` (for the question text), `@ID` (for the unique question ID), `@CategoryID` (for the category ID that has been assigned to the question), and `@Enabled` (which indicates whether a question is enabled). After the parameters are set up, the `ExecuteNonQuery()` method is called.

Three of the four parameters are marked for output and will contain important information. The question ID, category ID, and enabled flag are all available after the `QuestionInfoFromText` stored procedure has been executed.

The answers must all be obtained from the database. This is done with the `sp_AnswerInfo` stored procedure shown below.

```
CREATE PROCEDURE sp_AnswerInfo
    @ID int
AS
    SELECT Text FROM Answers WHERE QuestionID=@ID
GO
```

Each answer that is retrieved is appended to the Answers TextField object. And all variables, such as `Session("CurrentQuestionID")`, are set so that proper application behavior will result.

Listing 3.10 The `QuestionList_SelectedIndexChanged()` Method

```
Private Sub QuestionList_SelectedIndexChanged(ByVal sender As _
  System.Object, ByVal e As System.EventArgs) Handles _
  QuestionList.SelectedIndexChanged
    Dim lb As ListBox lb = sender

    Dim myConnection As New _
      SqlConnection(Application("DBConnectionString").ToString())

    Try
        myConnection.Open()
      Dim myCommand As New SqlCommand("sp_QuestionInfoFromText", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@Text", _
          SqlDbType.VarChar, 254))
        myCommand.Parameters("@Text").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@Text").Value = _
         lb.SelectedItem.Value

        myCommand.Parameters.Add(New SqlParameter("@ID", _
         SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Output

        myCommand.Parameters.Add(New SqlParameter("@CategoryID", _
          SqlDbType.Int))
        myCommand.Parameters("@CategoryID").Direction = _
          ParameterDirection.Output

        myCommand.Parameters.Add(New SqlParameter("@Enabled", _
          SqlDbType.Int))
        myCommand.Parameters("@Enabled").Direction = _
          ParameterDirection.Output

        myCommand.ExecuteNonQuery()

        Dim nCatID As Integer = _
          Convert.ToInt32(myCommand.Parameters("@CategoryID").Value)
        Dim nID As Integer = _
```

```
        Convert.ToInt32(myCommand.Parameters("@ID").Value)
    If nID <> -1 Then
        Session("CurrentQuestionID") = nID
        QuestionID.Text = Convert.ToString(nID)
        Question.Text = lb.SelectedItem.Value
        Enabled.Checked = True
     If _
  Convert.ToInt32(myCommand.Parameters("@Enabled").Value)= 0 _
       Then
           Enabled.Checked = False
        End If
        Answers.Text = ""

        myCommand = New SqlCommand("sp_AnswerInfo", _
           myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@ID", _
          SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@ID").Value = nID

        Dim reader As SqlDataReader = _
           myCommand.ExecuteReader()

        While reader.Read()
            Answers.Text += (reader.GetString(0) + vbCrLf)
        End While

        reader.Close()

        If nCatID < 0 Then
            nCatID = 0
        End If

        CategoryList.SelectedIndex = nCatID
    End If

    myConnection.Close()
Catch ex As Exception
    If myConnection.State = ConnectionState.Open Then
        myConnection.Close()
```

```
        End If
        Message.Text = ex.Message.ToString()
    End Try
End Sub
```

As you can see, the code in the Administer source code is straightforward. It follows a fairly predictable pattern and uses stored procedures for optimal performance.

The Main Survey Application Code

In the project code, you'll find all of the main screen functionality in default.aspx.cs or default.aspx.vb (depending on whether you are using the C# or VB version). This source code module contains all of the methods that perform the administrative functions for the application. Table 3.4 shows what the methods are and describes their purpose.

The `Page_Load()` Method

The `Page_Load()` method performs a fairly powerful procedure. It obtains the data for a question (both the question and all choices) and populates the user interface objects (SurveyQuestion and AnswerList). Although this procedure is powerful, it appears simple because a Web Service is invoked that returns the information.

The code in Listing 3.11 instantiates a `Web Service` class (named `com.aspnet_solutions.www.SurveyItem`), invokes its `GetSurveyData()` method, and receives a populated SurveyData structure that contains all the necessary survey question information.

You might notice that the `GetSurveyData()` method takes two arguments, both of which are `-1` here. The first argument lets the caller specify a category ID. That way, a specific category can be selected from. If the value is `-1`, then the survey question is selected from all categories.

The second argument allows a specific question ID to be asked for. This way, if you want to make sure a certain question is asked, you can pass the question's ID number as the second argument. If this value is `-1`, it is ignored.

It's important to take a look at the data structures that are used in the application. They can be seen in Listing 3.11.

Table 3.4 The Survey Application Main Page Methods Found in default.aspx.cs and default.aspx.vb

Method	Listing	Description
`Page_Load()`	3.11	This method executes when the default.aspx page is requested. If the request is not a post back, a survey question is retrieved from the TheSurvey Web Service.
`LoginButton_Click`	3.12	This is the event handler that is fired when the Login button is clicked. This method takes the user name and password, checks them for a match in the database, and then goes to the Administer.aspx page if a match has been found.
`VoteButton_Click`	3.13	This is the event handler that is fired when the Vote button is clicked. The Web Service is called upon to register the vote.
`ResultsButton_Click`	3.14	This is the event handler that is fired when the Results button is clicked. The Web Service is called upon to retrieve the results.

Listing 3.11 The `Page_Load()` Method

```
If Not IsPostBack Then
    Dim srv As New com.aspnet_solutions.www.SurveyItem()
    Dim data As com.aspnet_solutions.www.SurveyData = _
      srv.GetSurveyData(-1, -1)
    SurveyQuestion.Text = data.strQuestion
    If SurveyQuestion.Text.Length = 0 Then
        SurveyQuestion.Text = data.strError
    End If
    Dim i As Integer
    For i = 0 To data.Answers.Length - 1
        Dim item As New ListItem(data.Answers(i))
        AnswerList.Items.Add(item)
    Next
    QuestionID.Text = Convert.ToString(data.nQuestionID)
End If
```

The `LoginButton_Click()` Method

The `LoginButton_Click()` method shown in Listing 3.12 checks the database for a match with the user's name and password. It uses a stored procedure named `sp_Login` that's shown below.

```
CREATE PROCEDURE sp_Login
    @Name varchar(254),
    @Password varchar(254),
    @ID int output
AS
    SELECT @ID=ID FROM Administrators WHERE Name=@Name
        AND Password=@Password

    if( @ID IS NULL )
        SELECT @ID = -1
GO
```

The `sp_Login` stored procedure takes three parameters: `@Name`, `@Password`, and `@ID`. The `@ID` parameter will contain the ID of the user if a match was found. If not match was found, the ID will be `-1`.

If the login was successful, the user is redirected to Administer.aspx. If not, a message stating that the login failed is placed into the Message Label object.

Listing 3.12 The `LoginButton_Click()` Method

```
Private Sub LoginButton_Click(ByVal sender As System.Object, ByVal e _
  As System.EventArgs) Handles Button1.Click
    Dim myConnection As New _
SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        myConnection.Open()
        Dim myCommand As New SqlCommand("sp_Login", myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@Name", _
          SqlDbType.VarChar, 254))
        myCommand.Parameters("@Name").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@Name").Value = Name.Text
```

```
        myCommand.Parameters.Add(New SqlParameter("@Password", _
          SqlDbType.VarChar, 254))
        myCommand.Parameters("@Password").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@Password").Value = Password.Text

        myCommand.Parameters.Add(New SqlParameter("@ID", _
         SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Output

        myCommand.ExecuteNonQuery()
        myConnection.Close()

        Dim nID As Integer = _
          Convert.ToInt32(myCommand.Parameters("@ID").Value)
        If nID = -1 Then
            Message.Text = "Login failure"
        Else
            Session("AdminID") = nID
            Response.Redirect("Administer.aspx")
        End If
    Catch ex As Exception
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
        Message.Text = ex.Message.ToString()
    End Try
End Sub
```

The `VoteButton_Click()` Method

You would think that the `VoteButton_Click()` method as shown in Listing 3.13 would be complicated. It's not; it's simple. That's because it calls the Web Service's `Vote()` method, which takes care of the dirty work of registering the vote in the database.

That's the beauty of using Web Services; your application focus on program logic and not on procedural things that can easily be encapsulated in Web Services. Other situations in which to use a Web Service might include when you want to allow voting from other client applications and when you want to keep vote functionality close to the database server but deploy the larger application across a Web farm.

The code in the `VoteButton_Click()` method starts by setting Button2's `Visible` property to `False`. This helps prevent users from voting more than once (although they could simply reload the page and vote again).

The SurveyMessage Label object is set with a message thanking the user for voting.

The Web Service is instantiated, and the `Vote()` method is called. The `Vote()` method needs two parameters, the answer number (0, 1, 2, and so on) and the question ID number. If you want to skip ahead, the code for the `Vote()` method can be seen in Listing 3.18.

Listing 3.13 The `VoteButton_Click()` Method

```
Private Sub Vote_Click(ByVal sender As System.Object, ByVal e As_
  System.EventArgs) Handles Button2.Click
    Vote.Visible = False
    SurveyMessage.Text = "Thanks for voting!"
    Try
      Dim srv As New com.aspnet_solutions.www.SurveyItem()
      Dim nAnswerNumber As Integer = AnswerList.SelectedIndex
      srv.Vote(Convert.ToInt32(QuestionID.Text), nAnswerNumber)
    Catch ex As Exception
      SurveyMessage.Text = ex.Message.ToString()
    End Try
End Sub
```

The `ResultsButton_Click()` Method

When users click on the Results button, the `ResultsButton_Click()` method is invoked, as shown in Listing 3.14. This method goes to the Web Service for the results that pertain to the currently displayed survey question.

The first thing the method does is instantiate a `Web Service` class. A call to the `GetResults()` method is then made. The only parameter this method requires is the question ID, and this is supplied by converting a hidden field named QuestionID to an integer.

A data structure containing the relevant information is returned from the `GetResults()` method. For details, see Listing 3.15.

Once the survey results have been retrieved, the SurveyMessage Label object is populated with the survey result data.

Listing 3.14 The `Results_Click()` Method

```
Private Sub ResultsButton_Click(ByVal sender As System.Object, ByVal e_
  As System.EventArgs) _
       Handles Button3.Click
    Dim srv As New com.aspnet_solutions.www.SurveyItem()
    Dim res As com.aspnet_solutions.www.SurveyResults = _
       srv.GetResults(Convert.ToInt32(QuestionID.Text))
    If res.strError.Length > 0 Then
        SurveyMessage.Text = res.strError
        Return
    End If
    SurveyMessage.Text = "The results are:<br>" + vbCrLf
    Dim i As Integer
    For i = 0 To res.nCount.Length - 1
        SurveyMessage.Text += (AnswerList.Items(i).Value + ": ")
        Dim strPercent As String = res.dPercent(i).ToString(".00")
        If res.dPercent(i) = 0 Then
            strPercent = "0"
        End If
        SurveyMessage.Text += (strPercent + "%<br>" + vbCrLf)
    Next
End Sub
```

As you can see, the code in the Survey application's main page is simple. This simplicity is a direct result of using a Web Service to encapsulate the survey functionality.

TheSurvey Web Service

In the TheSurvey Web Service project, you'll find all of the Web Service functionality in surveyItem.asmx.cs or surveyItem.asmx.vb (depending on whether you are using the C# or VB version). This source code module contains all the methods that perform the administrative functions for the application. Table 3.5 shows what the methods are and describes their purpose.

The Data Structures

To return all the information necessary to display a survey question on the client machine, the application needs a data structure. A Web Service can return only one thing (via a return statement), and it can't have reference

Table 3.5 The Survey Web Service Methods Found in surveyItem.asmx.cs and surveyItem.aspx.vb

Method	Listing	Description
`_GetSurveyData()`	3.16	This method creates the survey data. It takes two integer arguments—`nCategoryID` and `nQuestionID`—and retrieves the appropriate question and answer data.
`GetSurveyData()`	3.17	This method simply returns the values that are obtained by calling the `_GetSurveyData()` method.
`Vote()`	3.18	This method takes the vote data and records it in the database.
`GetResults()`	3.19	This method gets the results for the survey question number that's passed in.

(noted by the `ref` keyword) variables that are passed in (which expect to be populated before a method returns). To solve the problem in which we need to pass back the question, an error (if it occurs), the list of answers, the question ID, and the category ID, we'll collect all of the information into a data structure.

The Web Service also needs to return information pertaining to survey results. For this, another data structure collects the information so that it can be returned as a single data type. The data structure that contains the survey question data is called SurveyData, and it can be seen in Listing 3.15. Also shown in Listing 3.15 is the SurveyResults data structure.

Listing 3.15 The Data Structures Used to Return Information

C#

```
public struct SurveyData
{
    public string strQuestion;
    public string strError;
    public StringCollection Answers;
    public int nQuestionID;
    public int nCategoryID;
}

public struct SurveyResults
{
```

```
    public string strError;
    public int[] nCount;
    public double[] dPercent;
}
```

VB

```
Public Structure SurveyData
    Public strQuestion As String
    Public strError As String
    Public Answers As StringCollection
    Public nQuestionID As Integer
    Public nCategoryID As Integer
End Structure

Public Structure SurveyResults
    Public strError As String
    Public nCount As Integer()
    Public dPercent As Double()
End Structure
```

The `_GetSurveyData()` Method

The `_GetSurveyData()` method is marked as private. The publicly callable method is called `GetSurveyData()`. The real work is done in `_GetSurveyData()`, and `GetSurveyData()` simply calls `_GetSurveryData()` (as shown in Listing 3.16) to return its results.

This was done so that the Web Service can be easily extended at a later time. When I developed the Web Service, I considered returning two versions of the data: one with the question and a list of answers (as is returned now in the `GetSurveyData()` method), and one that includes user-interface HTML codes so that the client application doesn't have to construct the presentation's objects but can just use what is retrieved from the Web Service.

If you ever extend the Web Service so that you have a method called `GetSurveyInHTML()` that returns the survey with the appropriate HTML, you can still call the `_GetSurveyData()` method to get the actual survey data. You can then construct the HTML data in your `GetSurveyInHTML()` method before returning the HTML data to the client application.

The `_GetSurveyData()` method has two paths: one when a specific question ID has been given, and the other when the question ID value has been given as `-1`, which indicates the pool of all questions can be drawn upon. If the first path is taken, the routine calls the `sp_QuestionFrom-`

ID stored procedure to retrieve the information corresponding to the question ID.

The second path of the _GetSurveyData() method follows this sequence: Find the number of survey questions in the database that match the criteria (either a given category ID or all questions), generate a random number that's in the correct range, and then retrieve the row that matches the random number. To accomplish the first task, a stored procedure named sp_QuestionCount (which is shown below) is called. This procedure requires a single input parameter that indicates the requested category ID. If this parameter value is less than 0, then all categories are selected.

```
CREATE PROCEDURE sp_QuestionCount
    @CategoryID int,
    @Count as int output
AS
    if( @CategoryID < 0 )
        SELECT @Count=Count(*) FROM Questions WHERE Enabled=1
    else
        SELECT @Count=Count(*) FROM Questions WHERE Enabled=1
          AND CategoryID=@CategoryID
GO
```

An instance of the Random class is created to provide random number functionality. A call is made to its Next() method, with a parameter indicating the largest number desired, thus generating the random number. Remember that this number is zero based—it ranges from zero to the record count minus one. The following code shows how the random number is generated:

```
Dim rnd As New Random()
Dim nRandomNumber As Integer = rnd.Next(nCount - 1)
```

With the random number generated, a call to the sp_GetSingle-Question stored procedure can be made (shown below). This stored procedure takes two parameters—the random number and the category ID. Here again, the category ID can be –1, which indicates that all categories can be drawn upon. The random number can't be zero based because the SQL FETCH Absolute command considers the first row to be numbered as 1. For this reason, we add one to the random number when we assign the @RecordNum parameter's value.

```
CREATE PROCEDURE sp_GetSingleQuestion
    @RecordNum int,
    @CategoryID int
AS
    if( @CategoryID >= 0 )
    begin
        DECLARE MyCursor SCROLL CURSOR
            For SELECT Text,ID,CategoryID FROM Questions WHERE
               Enabled=1
               AND CategoryID=@CategoryID
        OPEN MyCursor
        FETCH Absolute  @RecordNum from MyCursor
        CLOSE MyCursor
        DEALLOCATE MyCursor
    end
    else
    begin
        DECLARE MyCursor SCROLL CURSOR
            For SELECT Text,ID,CategoryID FROM Questions WHERE
            Enabled=1
        OPEN MyCursor
        FETCH Absolute  @RecordNum from MyCursor
        CLOSE MyCursor
        DEALLOCATE MyCursor
    end
GO
```

Once we have the question information (which includes the question ID), whether the code took the first or second paths, we can get the answers for this question. The code calls the `sp_AnswerInfo` stored procedure to retrieve all the answers for this survey question. The answers will be in a SqlDataReader object, and the code just loops through and gets each record.

Listing 3.16 The `_GetSurveyData()` Method

```
Private Function _GetSurveyData(ByVal nCategoryID As Integer, _
  ByVal nQuestionID As Integer) As SurveyData

    ' Create a SurveyData object and set its
    '   properties so the it will contain a
    '   StringCollection object, the question id
    '   and the category id.
```

```
Dim sd As SurveyData
sd.strQuestion = ""
sd.strError = ""
sd.Answers = New StringCollection()
sd.nQuestionID = nQuestionID
sd.nCategoryID = nCategoryID

' Create the connection object.
Dim myConnection As New _
  SqlConnection(Application("DBConnectionString").ToString())

Try
    ' Open the connection
    myConnection.Open()

    Dim myCommand As SqlCommand
    Dim reader As SqlDataReader = nothing

    ' If we have a valid question id, perform this code.
    If nQuestionID >= 0 Then
        ' Create a command the will use the sp_QuestionFromID
        '   stored procedure.
        myCommand = New SqlCommand("sp_QuestionFromID", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        ' Add a parameter for the question id named @ID
        '   and set the direction and value.
        myCommand.Parameters.Add(New SqlParameter("@ID", _
          SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@ID").Value = nQuestionID

        ' Retrieve a recordset by calling the ExecuteReader()
        '   method.
        reader = myCommand.ExecuteReader()

        ' If we got a record, set the question text and
        '   the category id from it.
        If reader.Read() Then
            sd.strQuestion = reader.GetString(0)
            sd.nCategoryID = reader.GetInt32(1)
        End If
```

```
        ' Set the question id and close the reader.
        sd.nQuestionID = nQuestionID
        reader.Close()
    Else
        ' This is a new question, so we'll need the count from
        '   the category.
        myCommand = New SqlCommand("sp_QuestionCount", _
           myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        ' The parameter is CategoryID since we need to specify
        '   the category id.
        myCommand.Parameters.Add(_
          New SqlParameter("@CategoryID", _
          SqlDbType.Int))
        myCommand.Parameters("@CategoryID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@CategoryID").Value = -
            nCategoryID

        ' The count will be retrieved, and is therefore set
        '   for output direction.
        myCommand.Parameters.Add(New SqlParameter("@Count", _
          SqlDbType.Int))
        myCommand.Parameters("@Count").Direction = _
          ParameterDirection.Output

        ' Execute the stored procedure by calling the
        '    ExecuteNonQuery() method.
        myCommand.ExecuteNonQuery()

        ' Get the count as in Int32.
        Dim nCount As Integer = _
          Convert.ToInt32(myCommand.Parameters("@Count").Value)

        ' If the count is zero, we have a problem and will
        '   alert the user to the error and return.
        If nCount = 0 Then
            sd.strError = _
              "The sp_QuestionCount procedure returned zero."
            myConnection.Close()
            Return
        End If
```

```
' We need a random number from 0 to nCount - 1.
Dim rnd As New Random()
Dim nRandomNumber As Integer = rnd.Next(nCount - 1)

' We're going to call the sp_GetSingleQuestion
'   stored procedure.
myCommand = _
  New SqlCommand("sp_GetSingleQuestion", myConnection)
myCommand.CommandType = CommandType.StoredProcedure

' We need to specify the category id.
myCommand.Parameters.Add(_
  New SqlParameter("@CategoryID", _
  SqlDbType.Int))
myCommand.Parameters("@CategoryID").Direction = _
  ParameterDirection.Input
myCommand.Parameters("@CategoryID").Value = _
  nCategoryID

' We need to specify the record number that we're
'   after.
myCommand.Parameters.Add(_
  New SqlParameter("@RecordNum", _
  SqlDbType.Int))
myCommand.Parameters("@RecordNum").Direction = _
  ParameterDirection.Input
myCommand.Parameters("@RecordNum").Value = _
  nRandomNumber + 1

' Execute the stored procedure by calling the
'   ExecuteReader() method. This returns a recordset.
reader = myCommand.ExecuteReader()

' If we got a record, perform this code.
If reader.Read() Then
    ' Store the question text.
    sd.strQuestion = reader.GetString(0)
    ' Store the question id.
    sd.nQuestionID = reader.GetInt32(1)
    sd.nCategoryID = reader.GetInt32(2)
    ' Store the category id.
```

```
            MyReader.Close()
        End If

        ' We're going to call the sp_AnswerInfo stored procedure.
        myCommand = New SqlCommand("sp_AnswerInfo", myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        ' Create an id parameter and set its value.
        myCommand.Parameters.Add(New SqlParameter("@ID", -
          SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@ID").Value = sd.nQuestionID

        ' Execute the stored procedure by calling the
        '   ExecuteReader() method. This returns a recordset.
        reader = myCommand.ExecuteReader()

        ' For each record, add the string to the StringCollection
        '   object.
        While reader.Read()
            sd.Answers.Add(reader.GetString(0))
        End While
        reader.Close()
    Catch ex As Exception
        sd.strError = ex.Message.ToString()
    Finally
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
    End Try

    Return (sd)

End Function
```

The `GetSurveyData()` Method

There isn't much to the `GetSurveyData()` method. It simply calls the `_GetSurveyData()` method and returns the results. As discussed earlier in the text, this was done so that the survey generation code could be a

private method that other methods (added at a later date) could call upon to retrieve survey data.

Listing 3.17 The `GetSurveyData()` Method

```
<WebMethod()> Public Function GetSurveyData(ByVal nCategory As Integer,_
   ByVal nQuestionID As Integer) As SurveyData
    Return (_GetSurveyData(nCategory, nQuestionID))
End Function
```

The `Vote()` Method

The `Vote()` method is straightforward. It takes the question number and the answer key (which is actually the order of the answer, with a value such as `0`, `1`, `2`, and so on) and calls the `sp_Vote` stored procedure. This stored procedure simply increments that value in the database of the appropriate question, as shown below:

```
CREATE PROCEDURE sp_Vote
    @ID int,
    @Answer int
AS
    UPDATE Answers SET Cnt=Cnt+1 WHERE Ord=@Answer AND
     QuestionID=@ID
GO
```

The actual `Vote()` method creates and opens a database connection (SqlConnection), creates a Command object (SqlCommand), sets up the `@ID` and `@Answer` parameters, and executes the stored procedure (with the `ExecuteNonQuery()` method). The code can be seen in Listing 3.18.

Listing 3.18 The `Vote()` Method

```
<WebMethod()> Public Function Vote(ByVal nQuestionID As Integer,_
   ByVal nAnswerNumber As Integer)
    Dim myConnection As New _
      SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        myConnection.Open()
```

```
        Dim myCommand As New SqlCommand("sp_Vote", myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        myCommand.Parameters.Add(New SqlParameter("@ID", _
          SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@ID").Value = nQuestionID

        myCommand.Parameters.Add(New SqlParameter("@Answer", _
          SqlDbType.Int))
        myCommand.Parameters("@Answer").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@Answer").Value = nAnswerNumber

        myCommand.ExecuteNonQuery()
        myConnection.Close()
    Catch ex As Exception
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
    End Try
End Function
```

The `GetResults()` Method

The `GetResults()` method performs three main tasks: It gets a record set with the number of votes for the answers, it creates a list of the raw answer counts in the data structure, and it creates a list of the percentages for each answer in the data structure.

The `sp_Results` stored procedure is called upon to retrieve the answers, and this stored procedure can be seen below.

```
CREATE PROCEDURE sp_Results
    @ID int
AS
    SELECT Cnt FROM Answers WHERE QuestionID=@ID ORDER BY Ord
GO
```

The next chunk of code that's in the `GetResults()` method takes care of creating the list of answer counts. These values are the counts for each answer, and they indicate how many times the answers have been voted for.

The last part of the method takes the counts for each answer and calculates the total number of votes for the question. It then goes through and calculates the percentage of votes that each answer has received. The entire `GetResults()` method can be seen in Listing 3.19.

Listing 3.19 The `GetResults()` Method

```
<WebMethod()> Public Function GetResults(ByVal nQuestionID As_
  Integer) As SurveyResults
    ' Create a SurveyResults object and initialize some members.
    Dim sr As SurveyResults
    sr.strError = ""
    sr.nCount = Nothing
    sr.dPercent = Nothing

    ' Create the connection object.
    Dim myConnection As New _
      SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        ' Open the connection.
        myConnection.Open()

        ' We're going to call the sp_Results stored procedure.
        Dim myCommand As New SqlCommand("sp_Results", _
          myConnection)
        myCommand.CommandType = CommandType.StoredProcedure

        ' We'll have to specify the ID as a parameter and set its
        '    value.
        myCommand.Parameters.Add(New SqlParameter("@ID", _
           SqlDbType.Int))
        myCommand.Parameters("@ID").Direction = _
          ParameterDirection.Input
        myCommand.Parameters("@ID").Value = nQuestionID

        ' Call the ExecuteReader() method, which returns a
        '    recordset that's contained in a SqlDataReader object.
        Dim reader As SqlDataReader = myCommand.ExecuteReader()

        ' Go through the records and store the new result.
        Dim i As Integer
```

```
        Dim nCount As Integer = 0
        While reader.Read()
            ' Increment the counter            nCount = nCount + 1

            ' Create a temporary Integer array and copy
            '    the values from the nCount array into it.

            Dim nTempCounts(nCount) As Integer
            For i = 0 To nCount - 2
                nTempCounts(i) = sr.nCount(i)
            Next

            ' Now reinitialize the nCount Integer array to contain
            '    one more than it contains now. Copy the old
            '    values into it.
            sr.nCount(nCount) = New Integer()
            For i = 0 To nCount - 2
                sr.nCount(i) = nTempCounts(i)
            Next
           ' Copy the new value into the newly-created array.
            sr.nCount(nCount - 1) = reader.GetInt32(0)
        End While

        ' We're now going to total up all of the counts.
        Dim dTotal As Double = 0
        For i = 0 To nCount = 1
            dTotal = dTotal + sr.nCount(i)
        Next

        ' Create a double array for the percents.
        sr.dPercent(nCount) = New Double()
        ' Loop through the list.
        For i = 0 To nCount - 1
            ' Either set the percent to zero, or calculate it.
            If dTotal = 0 Then
                sr.dPercent(i) = 0
            Else
                sr.dPercent(i) = (sr.nCount(i) * 100.0) / dTotal
            End If

        Next
    Catch ex As Exception
        sr.strError = ex.Message.ToString()
```

```
    Finally
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
    End Try

    Return (sr)
End Function
```

Extending and Modifying the Survey Application

I will do some things in the future to make the Survey application even more useful and flexible. I plan to do a couple of things to streamline the source code, too. I'm going to talk about my ideas here, and I might post them on the Web site in the future.

Streamlining the Code

I wrote this chapter so that it was easy to understand. Sometimes, I optimize code and sacrifice code readability as a result. I didn't want that to happen in the chapter examples, so as a result, some code could have been streamlined. The most obvious example in which I could have streamlined code is that for adding parameters to SqlCommand objects. Three lines of code are required to add a parameter along with its value, as shown here:

```
myCommand.Parameters.Add(New SqlParameter("@ID", SqlDbType.Int))
myCommand.Parameters("@ID").Direction = ParameterDirection.Input
myCommand.Parameters("@ID").Value = nQuestionID
```

It would be good to create a helper method named `AddParameter()` that offers a single line of code to do what is done here in three lines of code. By **helper method,** I'm referring to a method that performs a small amount of functionality, not a full-blown sequence of functionality. We actually need two versions of the `AddParameter()` method for the Survey application, one for integers and one for strings. The two methods are shown below.

C#

```
void AddParameter( SqlCommand myCommand, string strParamName,
  int nValue )
{
    myCommand.Parameters.Add( new SqlParameter( strParamName,
```

```
            SqlDbType.Int ) );
        myCommand.Parameters[strParamName].Direction =
            ParameterDirection.Input;
        myCommand.Parameters[strParamName].Value = nValue;
    }

    void AddParameter( SqlCommand myCommand, string strParamName,
      string strValue, int nSize )
    {
        myCommand.Parameters.Add( new SqlParameter( strParamName,
          SqlDbType.VarChar, nSize ) );
        myCommand.Parameters[strParamName].Direction =
          ParameterDirection.Input;
        myCommand.Parameters[strParamName].Value = strValue;
    }
```

VB

```
Sub AddParameter(ByVal myCommand As SqlCommand,_
  ByVal strParamName As string, ByVal nValue As Integer)
    myCommand.Parameters.Add(New SqlParameter(strParamName, _
      SqlDbType.Int))
    myCommand.Parameters(strParamName).Direction = _
      ParameterDirection.Input
    myCommand.Parameters(strParamName).Value = nValue
End Sub

Sub AddParameter(ByVal myCommand As SqlCommand, _
  ByVal strParamName As string, ByVal strValue as string, _
  ByVal nSize As Integer )
    myCommand.Parameters.Add(New SqlParameter(strParamName,_
      SqlDbType.VarChar, nSize))
    myCommand.Parameters(strParamName).Direction = _
      ParameterDirection.Input
    myCommand.Parameters(strParamName).Value = strValue
End Sub
```

Each of the above methods can be called from Survey application code, thus replacing three lines of code with one. Note one thing, though: These methods add only parameters that have an Input direction.

The following code,

```
myCommand.Parameters.Add(New SqlParameter("@Text", _
  SqlDbType.VarChar, 254))
```

```
myCommand.Parameters("@Text").Direction = ParameterDirection.Input
myCommand.Parameters("@Text").Value = Question.Text

myCommand.Parameters.Add(New SqlParameter("@CategoryID", _
  SqlDbType.Int))
myCommand.Parameters("@CategoryID").Direction = -
  ParameterDirection.Input
myCommand.Parameters("@CategoryID").Value = _
  CategoryList.SelectedIndex

myCommand.Parameters.Add(New SqlParameter("@Enabled", -
  SqlDbType.Int))
myCommand.Parameters("@Enabled").Direction = _
  ParameterDirection.Input
myCommand.Parameters("@Enabled").Value = 0
```

can be replaced with these three lines:

```
AddParameter( myCommand, "@Text", Question.Text, 254 )
AddParameter( myCommand, _
   "@CategoryID", CategoryList.SelectedIndex)
AddParameter( myCommand, "@Enabled", 0 )
```

Using helper methods can reduce the amount of code in your applications. The `AddParameter()` method is a good example of how you can streamline your code with helper methods.

Here are the tradeoffs to consider when you are creating and using helper functions:

- Does the helper method really simplify things? It can actually complicate matters if a lot of parameters must be passed.
- Does the helper method make the code hard to read and thus hard to maintain? If the answer is Yes, then consider avoiding the use of helper functions that make code hard to read, especially if another developer will be maintaining the code.
- Does the helper function reduce the overall amount of code without obfuscating the code's intent? If so, then the use of helper methods is desirable.
- Does the helper function offer reusability so that the functionality can be maintained and bugs can be fixed in a single location? If so, then the user of helper methods is desirable.

Creating a `GetSurveyInHTML()` Method

You might want to create a method in the Web Service that takes a survey question and wraps it in HTML presentation code, by adding the appropriate HTML tags. If doing this would be helpful, then this section will get you started (although you most certainly will change the specifics of the presentation).

The method will start by calling the `_GetSurveyData()` method to retrieve the survey question and answer data. A single string will be returned to the client application that contains all of the HTML data.

The following method shown in Listing 3.20 creates HTML data that renders a survey question:

Listing 3.20 The `GetSurveyInHTML()` Method

C#

```
[WebMethod]
public string GetSurveyInHTML( int nCategoryID, int nQuestionID, string
strVoteURL )
{

    // Create the objects we'll need.
    SurveyData sd = _GetSurveyData( nCategoryID, nQuestionID );
    SurveyResults sr = GetResults( sd.nQuestionID );

    // Create the start of the Html data string.
    string strHTMLData =
      "<form name=\"Survey\" method=\"post\" action=\"" +
      strVoteURL +
      "?ID=4\">\r\n";
    strHTMLData += "<script language=\"javascript\">\r\n";
    strHTMLData += "\tfunction ShowResults()\r\n";
    strHTMLData += "\t{\r\n";
    strHTMLData += "\t\tSurveyResults.innerHTML = ";

    // Loop through each answer.
    for( int i=0; i<sd.Answers.Count; i++ )
    {

 // Add the answer, the supporting Html, and the formatted number.
      strHTMLData +=
       ( sd.Answers[i] + ": " + sr.dPercent[i].ToString( ".00" ) +
          "%");
```

```
        if( i < sd.Answers.Count - 1 )
        {
            strHTMLData += "<br>";
        }
        else
        {
            strHTMLData += "\r\n";
        }
    }

    // End this part of the Html.
    strHTMLData += "\t{\r\n";
    strHTMLData += "</script>\r\n";
    strHTMLData += ( "<P>" + sd.strQuestion + "<BR>\r\n" );

    // Loop through each answer again.
    for( int i=0; i<sd.Answers.Count; i++ )
    {
        // Create radio button Html code.
   strHTMLData += ( "<INPUT type=\"radio\" name=\"sr\" value=\"" +
          Convert.ToString( i ) + "\"> " +
           sd.Answers[i] + "<BR>\r\n" );
    }

    // Add the Vote button.
    strHTMLData +=
        "<INPUT type=\"submit\" value=\"Vote\">  " +
        "<INPUT type=\"button\" +
        " onclick=\"ShowResults()\" value=\"Results\"></P>\r\n";
    strHTMLData += "<div id=\"SurveyResults\"></div>\r\n";
    strHTMLData += "</form>\r\n";

    return( strHTMLData );
}
```

VB

```
<WebMethod()> Public Function GetSurveyInHTML(ByVal nCategoryID _
 As _
  Integer, ByVal nQuestionID As Integer, _
  ByVal strVoteURL As String) As String

 ' Create the objects we'll need.
 Dim sd As SurveyData = _GetSurveyData(nCategoryID, nQuestionID)
    Dim sr As SurveyResults = GetResults(sd.nQuestionID)
```

```
' Create the start of the Html data string.
Dim strHTMLData As String = _
  "<form name=Survey method=post action=" + strVoteURL + _
      "?ID=4>" + vbCrLf
strHTMLData += ("<script language=javascript>" + vbCrLf)
strHTMLData += ("function ShowResults()" + vbCrLf)
strHTMLData += ("    {" + vbCrLf)
strHTMLData += "        SurveyResults.innerHTML = "

' Loop through each answer.
Dim i As Integer
For i = 0 To sd.Answers.Count - 1
' Add the answer, the supporting Html, and the formatted number.
    strHTMLData += (sd.Answers(i) + ": " + _
      sr.dPercent(i).ToString(".00") + "%")
    If i < sd.Answers.Count - 1 Then
        strHTMLData += "<br>"
    Else
        strHTMLData += vbCrLf
    End If
Next

' End this part of the Html.
strHTMLData += ("    {" + vbCrLf)
strHTMLData += ("</script>" + vbCrLf)
strHTMLData += ("<P>" + sd.strQuestion + "<BR>" + vbCrLf)

' Loop through each answer again.
For i = 0 To sd.Answers.Count - 1

    ' Create radio button Html code.
    strHTMLData += ("<INPUT type=radio name=sr value=" + _
      Convert.ToString(i) + "> " + _
       sd.Answers(i) + "<BR>" + vbCrLf)
Next

' Add the Vote button.
strHTMLData += _
("<INPUT type=submit value=Vote>  <INPUT type=button" + _
    " onclick=ShowResults() value=Results></P>" + vbCrLf)
strHTMLData += ("<div id=SurveyResults></div>" + vbCrLf)
strHTMLData += ("</form>" + vbCrLf)
```

```
    Return (strHTMLData)
End Function
```

Adding a `ResetResults()` Method

At times, the results for a question need to be reset to zero. This reset might happen on a schedule, or as part of the administrative portion of the application. This section presents a method that can be added to the Web Service to do this, and it can be seen in Listing 3.21.

Listing 3.21 The `ResetResults()` Method

C#

```
[WebMethod]
public string ResetResults( int nQuestionID )
{
SqlConnection myConnection =
 new SqlConnection(
   Convert.ToString(Application["DBConnectionString"]));

    try
    {
        myConnection.Open();
        string strSql =
          "update Answers set Cnt=0 where QuestionID=" +
          Convert.ToString( nQuestionID );
        SqlCommand myCommand =
          new SqlCommand( strSql, myConnection );
        myCommand.ExecuteNonQuery();
        myConnection.Close();
    }
    catch( Exception ex )
    {
        if( myConnection.State == ConnectionState.Open )
        {
            myConnection.Close();
        }
    }
}
```

VB

```
<WebMethod()> Public Function ResetResults(ByVal nQuestionID As _
  Integer)
```

```
    Dim myConnection As New _
      SqlConnection(Convert.ToString(Application("DBConnectionString")))

    Try
        myConnection.Open()
        Dim strSql as string = _
          "update Answers set Cnt=0 where QuestionID=" + _
          Convert.ToString(nQuestionID)
        Dim myCommand As New SqlCommand(strSql, myConnection)
        myCommand.ExecuteNonQuery()
        myConnection.Close()
    Catch ex As Exception
        If myConnection.State = ConnectionState.Open Then
            myConnection.Close()
        End If
    End Try
End Function
```

Deploying the Survey Application

You must follow a few steps to deploy the Survey application on your server:

1. Start by creating a database named `Survey`.
2. Then, restore the database that can be obtained from the Web site. (The page that has a link to this download is www.ASPNET-Solutions. com/Chapter_3.htm.) A SQL Script also is available in the same place—you can use it in SQL Query Analyzer to create the database.
3. Next, run Visual Studio .NET. Create a project on the server named Survey for a C# project, or SurveyVB for a VB project. You'll have to make sure you select the appropriate (C# or VB) type of ASP.NET project.
4. Compile the application once it is created.
5. Now, create a project on the server named TheSurvey for a C# project, or TheSurveyVB for a VB project. Here again, you'll have to make sure you select the appropriate (C# or VB) type of ASP.NET Web Service project.
6. Compile the Web Service once it is created.
7. Close the projects in Visual Studio .NET. Not doing so will create a sharing violation for the next steps.

8. Make sure you have downloaded the zipped projects from the Web site. Unzip them (the Survey/SurveyVB application and the TheSurvey/TheSurveyVB Web Service) into the default Web site directory (usually c:\inetpub\wwwroot). You will be asked if you want to overwrite existing directories—answer Yes.
9. Open the application project. Check the Global.asax file to make sure that the database connection string matches the connection string for your situation. Compile it. Do the same with the Web Service.

All of this is easy so far, but here's the more difficult part. The application is using a Web Reference to the Web Service on the www.ASPNETSolutions.com server. That's OK (except that you're using my bandwidth!), but any changes you make to the Web Service will have no effect when your application runs. To fix this, do the following:

10. Delete the Web Reference in your Survey application.
11. Add a Web Reference to your Web Service.
12. Change the references in the code from the class to my Web Service to the class name that wraps your Web Service.

Your Survey application should now be ready to use and ready to modify as you see fit.

Summary

This chapter gives you a starting point for the effective use of ADO.NET. You've learned how to use the SqlConnection, SqlCommand, and SqlDataReader objects. You've also learned how to add parameters to a command and how to call a stored procedure.

A complete and reusable application was introduced to illustrate the ADO.NET topics that were presented. Along the way, best practices and design patterns were presented that will assist you in developing robust applications.

Although this chapter provides 90 percent of what you need for database access, Chapter 4 covers more advanced ADO.NET topics.