CHAPTER

3

ADO.NET Managed Providers

IN THIS CHAPTER

In the last chapter, you began looking at how an ASP.NET data-driven solution works. You looked at the ADO.NET object model, and how to build a `DataSet` dynamically. In this chapter, you'll dig into the two data access Managed Providers offered in ADO.NET: the SQL Managed Provider and the OleDB Managed Provider.

In this chapter, you'll learn the following:

- How the .NET Managed Providers are a bridge from the application, such as an ASP.NET Web Form, to a data store, such as Microsoft SQL Server.
- How to create Managed Connections to connect to a data store.
- How to use Managed Commands to execute SQL statements on a database.
- How to use `DataAdapters` to retrieve data and populate a `DataSet`.
- How to create custom table and column mappings.

Managed Providers, as shown in Figure 3.1, are ADO.NET's bridge from an application, such as an ASP.NET Web Form to the data source. Data sources include Microsoft's SQL Server, Access, Oracle, or any other such data storage device.

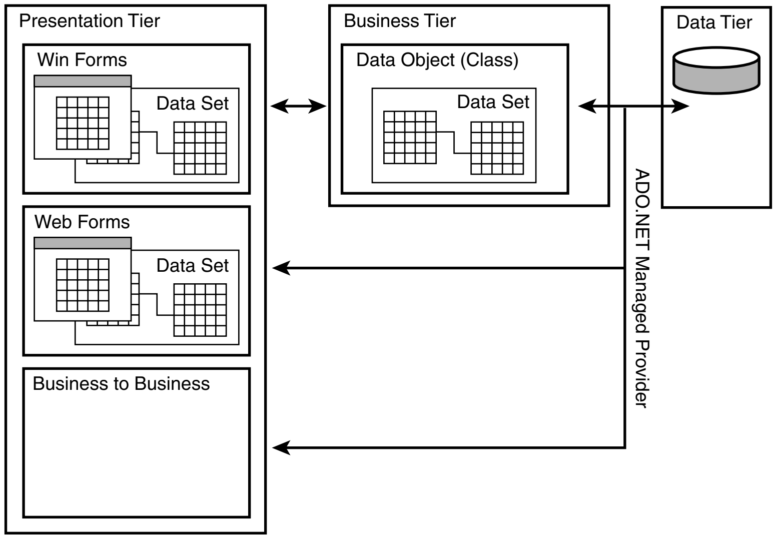


FIGURE 3.1
Managed Providers are the bridge from a data store to a .NET application.

The Managed Providers have four core components:

- **`Connection`**

  The Connection represents a unique session to a data store. This might be manifested as a network connection in a client/server database application.

- **`Command`**

  The Command represents a SQL statement to be executed on a data store.

- **`DataReader`**

  The DataReader is a forward-only, read-only stream of data records from a data store to a client.

- **`DataAdapter`**

  The DataAdapter represents a set of Commands and a Connection which are used to retrieve data from a data store and fill a DataSet.

The Two Managed Providers

ADO.NET, the successor to Microsoft's highly successful ActiveX Data Objects (ADO), offers two Managed Providers. These providers are similar in their object model, but are chosen at design-time based on the data provider being used. The SQL Managed Provider offers a direct link into Microsoft's SQL Server database application (version 7.0 or higher), while the OleDb Managed Provider is used for all other data providers. Following is a brief description of each of the Managed Providers. Throughout this chapter we will show you how the Managed Providers work, and specify when a particular object, property, method or event is unique to only one of the Managed Providers.

OleDb Managed Provider

The *OleDb Managed Provider* uses native OLEDB and COM Interop to establish a connection to a data store and negotiate commands. The OleDb Managed Provider is the data access provider to use when you are working with data from any data source that is not Microsoft's SQL Server 7.0 or higher. To use the OleDb Managed Provider, you must import the `System.Data.OleDb` namespace.

SQL Managed Provider

The *SQL Managed Provider* is designed to work directly with Microsoft SQL Server 7.0 or greater. It connects and negotiates directly with SQL Server without using OLEDB. This provides a better performance model than the OleDb Managed Provider, but it's restricted to use with Microsoft SQL Server 7.0 or higher. To use the SQL Managed Provider, you must import the `System.Data.SqlClient` namespace.

Managed Connections

Much like classic ADO, the OleDb and SQL Managed Connection objects (`OleDbConnection` and `SqlConnection`) provide a set of properties that might be familiar to you. These are listed in Table 3.1. Properties that apply to only one of the Managed Providers are indicated.

TABLE 3.1 Managed Connection Properties

Property	*Description*
`ConnectionString`	Gets or sets the string used to open a data store.
`ConnectionTimeout`	Gets or sets the time to wait while establishing a connection before terminating the attempt and generating an error.
`Container`	Returns the `IContainer` that contains the object.
`Database`	Gets or sets the name of the current database or the database to be used once a connection is open.
`DataSource`	Gets or sets the name of the database to connect to.
`PacketSize` (`SqlConnection` only)	Gets the size of the packets the data is transferred in.
`Provider` (`OleDbConnection` only)	Gets or sets the name of the OLEDB provider.
`ServerVersion` (`SqlConnection` only)	Gets a string containing the version of the connected SQL Server.
`Site`	Gets or sets the site of the component.
`State`	Gets the current state of the connection.

OleDbConnection

The OleDb Managed Provider uses a `ConnectionString` property format identical to that of a classic ADO connection object. Listing 3.1 shows how to connect to an Access 2000 database using the `OleDbConnection` object.

WARNING

In the following code listing, the OleDb Managed Connection object is pointing to an Access 2000 database file using the path, `C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb`. This is the default path to the Northwind sample database that is installed when Access 2000 is installed. The path on your machine might vary. Alter the code as necessary.

LISTING 3.1 Connecting to an Access 2000 Database with the OleDbConnection

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
```

LISTING 3.1 Continued

```
04:  Sub Page_Load(Sender As Object, E As EventArgs)
05:   Dim myConnection As OleDbConnection
06:   myConnection = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
➥     Data Source=C:\Program Files\Microsoft
➥      Office\Office\Samples\Northwind.mdb;")
07:   myConnection.Open()
08:   ConnectionState.Text = myConnection.State.ToString()
09:   myConnection.Close()
10:  End Sub
11: </script>
```

[C#]

```
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data.OleDb" %>
03: <script runat="server">
04:  void Page_Load(Object sender, EventArgs e){
05:   OleDbConnection myConnection;
06:   myConnection = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
➥      Data Source=C:\\Program Files\\Microsoft
➥      Office\\Office\\Samples\\Northwind.mdb;");
07:   myConnection.Open();
08:   ConnectionState.Text = myConnection.State.ToString();
09:   myConnection.Close();
10:  }
11: </script>
```

[VB & C#]

```
12: <html>
13: <body>
14: <form runat="server" method="post">
15:  Connection State: <asp:Label runat="server" id="ConnectionState" />
16: </form>
17: </body>
18: </html>
```

In Listing 3.1, you import the `System.Data.OleDb` namespace on line 2. On line 5, you declare a variable for the `OleDbConnection` class and on line 6, you instantiate the `OleDbConnection` class, passing in the `ConnectionString` as the connection's only parameter. The `ConnectionString` property specifies that the OLEDB Provider is `Microsoft.Jet.OLEDB.4.0`, the provider necessary to connect to an Access 2000 database.

On line 7 you open the connection with the `Open()` method of the `OleDbConnection` class. On line 8 you set the `Text` property of an ASP.NET `Label` to the string representation of the `State` property of the `OleDbConnection` class.

WARNING

In the C# example in Listing 3.1 you will notice that the `ConnectionString` property of the Managed Connection object uses a double slash (\\) between the tree hierarchy of the path to the Northwind sample database file. This is because C# treats the slash (\) as an escape character in a string. Using a double slash (\\) lets the compiler know that you really want to use a slash character in that spot.

NOTE

If you are using Access 2000 with user name and password security, you might see an error indicating that Access can not find the installable ISAM. This error is related to your Access 2000 installation, and not the .NET Framework. For more information, see `http://support.microsoft.com/support/kb/articles/Q209/8/05.ASP`.

SqlConnection

The SQL Managed Provider uses a `ConnectionString` property format that's similar to that of a classic ADO connection object. Since you know what the database application is from using the SQL Managed Provider, the Provider property isn't required (it isn't even allowed, for that matter). Listing 3.2 shows sample code for connecting to a Microsoft SQL Server database using the `SqlConnection` object.

LISTING 3.2 Connecting to a SQL Server Database with the SqlConnection

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  myConnection = New SqlConnection("server=localhost;
➥    database=Northwind; uid=sa; pwd=;")
  myConnection.Open()
  ConnectionState.Text = myConnection.State.ToString()
  myConnection.Close()
 End Sub
</script>
```

LISTING 3.2 Continued

```
[C#]

01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data.SqlClient" %>
03: <script runat="server">
04:  void Page_Load(Object sender, EventArgs e){
05:   SqlConnection myConnection;
06:   myConnection = new SqlConnection("server=localhost;
➥      database=Northwind; uid=sa; pwd=;");
07:   myConnection.Open();
08:   ConnectionState.Text = myConnection.State.ToString();
09:   myConnection.Close();
10:  }
11: </script>

[VB & C#]


12: <html>
13: <body>
14: <form runat="server" method="post">
15:  Connection State: <asp:Label runat="server" id="ConnectionState" />
16: </form>
17: </body>
18: </html>
```

In Listing 3.2 you create a connection to a SQL Server database. The code in Listing 3.2 is nearly identical to that of Listing 3.1. The only two differences are on lines 2 and 6. On line 2 you import the `System.Data.SqlClient` namespace rather than the `System.Data.OleDb` namespace. This allows you access to the SQL Managed Provider classes, like the `SqlConnection` class. On line 6 you create an instance of the `SqlConnection` class and pass in the `ConnectionString` property as the only parameter. In the `ConnectionString` property you *do not* specify a provider since the `SqlConnection` is designed to connect only to a Microsoft SQL Server database.

Managed Commands

Managed Commands represent SQL syntax to be executed on the data store. Managed Commands can be simple `SELECT` statements or complex, parameterized commands.

Once a connection to a data store is established, you can retrieve, update, or insert data. One way of accomplishing this is to use a Managed Command. This is the most direct way to execute a SQL statement on a data store.

As with the Managed Connection object, there are both OleDb and SQL versions of the Managed Command—`OleDbCommand` and `SqlCommand`.

The Managed Command is similar to the classic ADO command object. In its simplest form, you create a Managed Command with the SQL statement and the connection (as either a `ConnectionString` or a Managed Connection object) as its parameters. In Listing 3.3 you create a SqlCommand to execute a simple `SELECT` statement against the Northwind database. This example will not render any output, but you will build on it in the following examples.

NOTE

For the bulk of this chapter I will be showing samples using the SQL Managed Provider. You can use the OleDb Managed Provider by changing the Managed Provider classes from `SqlWidget` to `OleDbWidget`. For example, in Listing 3.3 you could use the `OleDbCommand` class in replacement of the `SqlCommand` class. Remember that the OleDb Managed Provider uses the `System.Data.OleDb` namespace instead of the `System.Data.SqlClient` namespace.

LISTING 3.3 Creating a SqlCommand Object

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  Dim myCommand As SqlCommand
  myConnection = New SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;")
  myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
 End Sub
</script>

[C#]

<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlConnection myConnection;
  SqlCommand myCommand;
  myConnection = new SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;");
  myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
 }
</script>
```

LISTING 3.3 Continued

```
[VB & C#]

<html>
<body>
<form runat="server" method="post">
 <asp:DataGrid runat="server" id="myDataGrid" />
</form>
</body>
</html>
```

In Listing 3.3 you create a Web Form that uses the SQL Managed Provider to create a `SqlConnection` and a `SqlCommand`. To execute the `SqlCommand` on the database you call one of the provided execute methods.

- `ExecuteNonQuery`: Executes a SQL statement that does not return any records.
- `ExecuteReader`: Returns a `DataReader` object.
- `ExecuteScalar`: Executes the SQL statement and returns the first column of the first row.
- `ExecuteXmlReader` (SQL Managed Provider only): Executes a SQL statement and returns the results as an XML stream.

The only thing you're missing before calling one of the execute methods is an object to hold the results that are returned. For this example you will use the `ExecuteReader()` method and return the results as a `DataReader` object (`SqlDataReader` or `OleDbDataReader`). Once you have a `DataReader` object you can work with the data. For now you will bind the `DataReader` to a `DataGrid` server control. In Listing 3.4 you will use the `ExecuteReader()` method to return the results of the command execution into a `DataReader` object.

LISTING 3.4 Executing a SqlCommand Object and Returning the Results in a DataReader

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  Dim myCommand As SqlCommand
  myConnection = New SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;")
  myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
  myConnection.Open()
  Dim myDataReader As SqlDataReader = myCommand.ExecuteReader()
  myDataGrid.DataSource = myDataReader
```

LISTING 3.4 Continued

```
12:   myDataGrid.DataBind()
13:   myConnection.Close()
14:  End Sub
15: </script>
```

[C#]

```
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data.SqlClient" %>
03: <script runat="server">
04:  void Page_Load(Object sender, EventArgs e){
05:   SqlConnection myConnection;
06:   SqlCommand myCommand;
07:   myConnection = new SqlConnection("server=localhost;
➥      database=Northwind; uid=sa; pwd=;");
08:   myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
09:   myConnection.Open();
10:   SqlDataReader myDataReader = myCommand.ExecuteReader();
11:   myDataGrid.DataSource = myDataReader;
12:   myDataGrid.DataBind();
13:   myConnection.Close();
14:  }
15: </script>
```

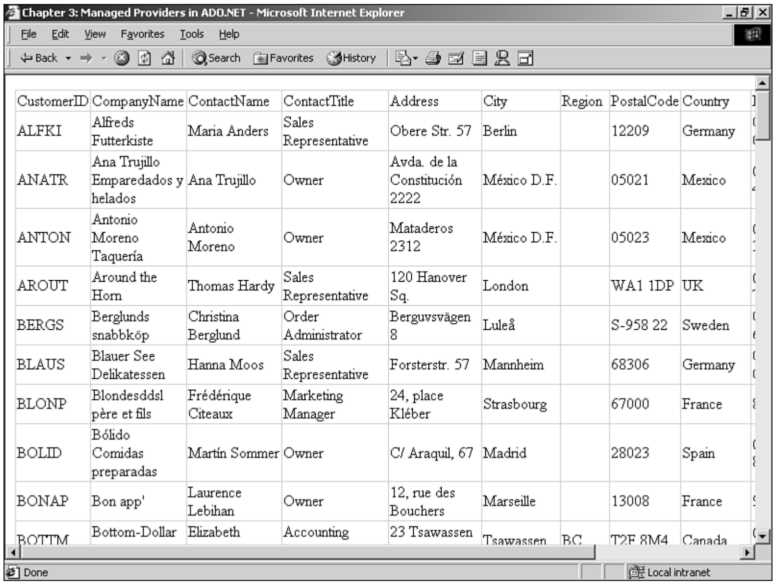
[VB & C#]

```
16: <html>
17: <body>
18: <form runat="server" method="post">
19:  <asp:DataGrid runat="server" id="myDataGrid" />
20: </form>
21: </body>
22: </html>
```

In Listing 3.4 you extend the code in Listing 3.3 to execute the `SqlCommand` object using the `ExecuteReader()` method. Since this method returns the results in a `DataReader` object, you first create an instance of the `SqlDataReader` class on line 10, and set it to the returned result of the `SqlCommand.ExecuteReader()` method. You will notice that I create an instance of the `DataReader` and assign it to the results of the `ExecuteReader()` on the same line. This is just another way of constructing an object in .NET, rather than doing the same thing across two lines of code.

Once the command has executed and the `DataReader` has been created, you bind the `DataReader` to a `DataGrid` server control (line 11) by setting the `DataSource` property of the `DataGrid` as the `DataReader`. Once the `DataSource` property is set you call the `DataBind()` method of the `DataGrid` to bind the command results to the output of the `DataGrid`.

Figure 3.2 shows the ASP.NET Web Form with the results from the Managed Command execution in a DataGrid.


Chapter 3: Managed Providers in ADO.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Done
Local intranet

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin		12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.		05021	Mexico
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.		05023	Mexico
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London		WA1 1DP	UK
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå		S-958 22	Sweden
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim		68306	Germany
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg		67000	France
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid		28023	Spain
BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille		13008	France
BOTTM	Bottom-Dollar	Elizabeth	Accounting	23 Tsawassen	Tsawassen	BC	T2F 8M4	Canada

FIGURE 3.2
Using the Managed Providers you can create a connection to a database, execute a command, and display the results on an ASP.NET Web Form.

The DataReader

The *DataReader* provides a forward-only, read-only stream of data from the data store. The `DataReader` is best used when either there are many records in the result set and pulling them all in at once would use too much memory, or when you want to iterate through the records to work with the data returned. As a stream of records, the `DataReader` helps manage memory allocation. Rather than all of the records in the result set being returned at once and using up a chunk of memory on the server, the `DataReader` streams in one record at a time.

You've seen how to execute a command against a data store, and in the previous listings, the results weren't too big. But imagine if your Managed Command returned a result set with over 100,000 records in it. Now imagine 1,000 users doing that all at the same time. It would use up the memory space for 100,000,000 records of data, and that could spell disaster for your Web application.

What would be ideal is a way to connect to the data store, bring the results back in a stream, and evaluate those results one record at a time. Ideally, this would only use up the memory for one record at a time.

This type of functionality is exactly what the DataReader provides. The DataReader is the classic "fire hose" of data access—a forward-only, read-only stream returned from the data store. In Listing 3.4, you bound a DataGrid server control to the DataReader. The DataGrid is covered in depth in Chapter 5, "Using a Basic DataGrid," and Chapter 6, "Altering DataGrid Output." This will set up the DataGrid to display the entire contents of the stream. However, with a data stream, you can step through the data that's returned very easily and decide how to react to it.

The DataReader exposes a Read() method which advances to the next record in the stream. Using the Read() method you can iterate through the result set evaluating or working with the data.

```
[VB]

While myDataReader.Read
  'Do something with the current row
End While

[C#]

while (myDataReader.Read()){
  //Do something with the current row
}
```

You should be able to come up with a good reason to step through the results of a command execution and evaluate the results. This is something you've done relentlessly as a classic ASP developer (does Do While Not RecordSet.EOF sound familiar?). How often have you looped through ADO RecordSets, checking the values of a particular column and using Response.Write on the RecordSet row if the criterion is met? Or granted access to a page if the user name and password columns match the submitted values?

In Listing 3.5 you will use the DataReader class to iterate through the result set and add any record with the value "USA" to a new DataTable. The DataTable and DataRow classes you learned about in Chapter 2, "What Is ADO.NET?" are used in Listing 3.5.

LISTING 3.5 Evaluating Data with the DataReader

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  Dim myCommand As SqlCommand
  Dim myDataTable As New DataTable
  Dim myRow As DataRow
```

LISTING 3.5 Continued

```

  myConnection = New SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;")
  myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
  myConnection.Open()
  Dim myDataReader As SqlDataReader = myCommand.ExecuteReader()

  myDataTable.Columns.Add("CustomerID",
➥   System.Type.GetType("System.String"))
  myDataTable.Columns.Add("CompanyName",
➥   System.Type.GetType("System.String"))
  myDataTable.Columns.Add("Address",
➥   System.Type.GetType("System.String"))
  myDataTable.Columns.Add("City",
➥   System.Type.GetType("System.String"))
  myDataTable.Columns.Add("Region",
➥   System.Type.GetType("System.String"))
  myDataTable.Columns.Add("Country",
➥   System.Type.GetType("System.String"))

  While myDataReader.Read
    If myDataReader("Country") = "USA" Then
      myRow = myDataTable.NewRow()
      myRow("CustomerID") = myDataReader("CustomerID")
      myRow("CompanyName") = myDataReader("CompanyName")
      myRow("Address") = myDataReader("Address")
      myRow("City") = myDataReader("City")
      myRow("Region") = myDataReader("Region")
      myRow("Country") = myDataReader("Country")
      myDataTable.Rows.Add(myRow)
    End If
  End While

  myDataGrid.DataSource = myDataTable
  myDataGrid.DataBind()

  myConnection.Close()
 End Sub
</script>
```

[C#]

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
```

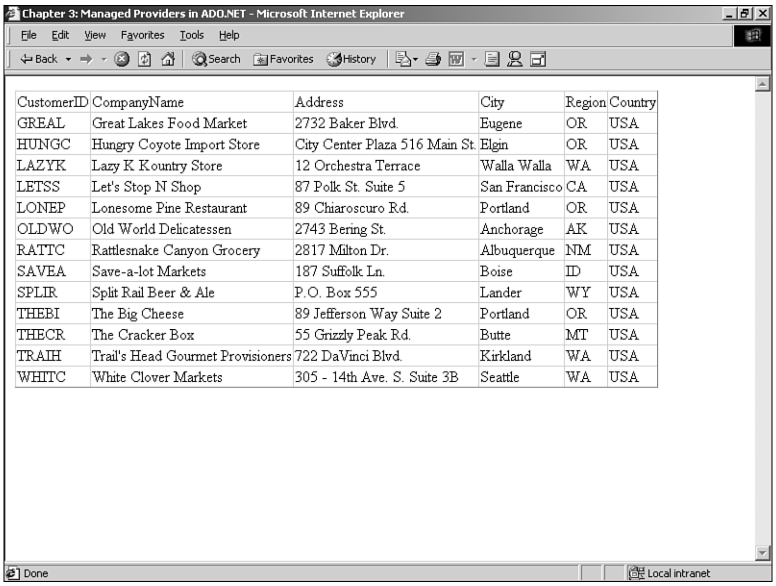
LISTING 3.5 Continued

```
 void Page_Load(Object sender, EventArgs e){
  SqlConnection myConnection;
  SqlCommand myCommand;
  DataTable myDataTable = new DataTable();
  DataRow myRow;

  myConnection = new SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;");
  myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
  myConnection.Open();
  SqlDataReader myDataReader = myCommand.ExecuteReader();

  myDataTable.Columns.Add("CustomerID",
➥   System.Type.GetType("System.String"));
  myDataTable.Columns.Add("CompanyName",
➥   System.Type.GetType("System.String"));
  myDataTable.Columns.Add("Address",
➥   System.Type.GetType("System.String"));
  myDataTable.Columns.Add("City",
➥   System.Type.GetType("System.String"));
  myDataTable.Columns.Add("Region",
➥   System.Type.GetType("System.String"));
  myDataTable.Columns.Add("Country",
➥   System.Type.GetType("System.String"));

  while(myDataReader.Read()){
    if(myDataReader["Country"].ToString() == "USA"){
      myRow = myDataTable.NewRow();
      myRow["CustomerID"] = myDataReader["CustomerID"].ToString();
      myRow["CompanyName"] = myDataReader["CompanyName"].ToString();
      myRow["Address"] = myDataReader["Address"].ToString();
      myRow["City"] = myDataReader["City"].ToString();
      myRow["Region"] = myDataReader["Region"].ToString();
      myRow["Country"] = myDataReader["Country"].ToString();
      myDataTable.Rows.Add(myRow);
    }
  }

  myDataGrid.DataSource = myDataTable;
  myDataGrid.DataBind();

  myConnection.Close();
 }
</script>
```

LISTING 3.5 Continued

```
[VB & C#]

42: <html>
43: <body>
44: <form runat="server" method="post">
45:  <asp:DataGrid runat="server" id="myDataGrid" />
46: </form>
47: </body>
48: </html>
```

In Chapter 2, you read an overview of the ADO.NET Document Object Model and saw samples of creating `DataTables` dynamically. Listing 3.5 makes use of that knowledge. On line 08, you create a `DataTable` dynamically. On lines 16–21, you add columns to the `DataTable`'s `Columns` collection. On lines 23–34, you step through the results of the SqlCommand execution with the `DataReader.Read()` method. If the "Country" field is "USA", you create a new `DataRow` (line 25) and add the values of the current record in the `DataReader` stream to the row (lines 26–31). On line 32, you add the new `DataRow` object to the `DataTable`'s `Rows` collection. Finally, you bind the dynamically created `DataTable` to the `DataGrid` server control. You end up with a table limited to customers in the USA, as shown in Figure 3.3.


Chapter 3: Managed Providers in ADO.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Done
Local intranet

CustomerID	CompanyName	Address	City	Region	Country
GREAL	Great Lakes Food Market	2732 Baker Blvd.	Eugene	OR	USA
HUNGC	Hungry Coyote Import Store	City Center Plaza 516 Main St.	Elgin	OR	USA
LAZYK	Lazy K Kountry Store	12 Orchestra Terrace	Walla Walla	WA	USA
LETSS	Let's Stop N Shop	87 Polk St. Suite 5	San Francisco	CA	USA
LONEP	Lonesome Pine Restaurant	89 Chiaroscuro Rd.	Portland	OR	USA
OLDWO	Old World Delicatessen	2743 Bering St.	Anchorage	AK	USA
RATTC	Rattlesnake Canyon Grocery	2817 Milton Dr.	Albuquerque	NM	USA
SAVEA	Save-a-lot Markets	187 Suffolk Ln.	Boise	ID	USA
SPLIR	Split Rail Beer & Ale	P.O. Box 555	Lander	WY	USA
THEBI	The Big Cheese	89 Jefferson Way Suite 2	Portland	OR	USA
THECR	The Cracker Box	55 Grizzly Peak Rd.	Butte	MT	USA
TRAIH	Trail's Head Gourmet Provisioners	722 DaVinci Blvd.	Kirkland	WA	USA
WHITC	White Clover Markets	305 - 14th Ave. S. Suite 3B	Seattle	WA	USA

FIGURE 3.3
You can use the `DataReader.Read()` *method to iterate through the result set and react to the data.*

Managed Commands with Stored Procedures

Although the example in Listing 3.5 allows you to evaluate data and output only what is desired, a more economical way to get a restricted result set is to use a SQL statement with a WHERE clause. Rather than querying the data store and bringing back all of the records in the table, you can write a SQL statement or stored procedure to bring back only the data you want. Stored procedures typically give better performance to your application than a SQL statement passed via a command.

You can use the Managed Command to call a stored procedure. In many cases, the stored procedure will have the parameters you pass it to determine the result set, such as the country abbreviation in the previous examples.

Listing 3.6 is the syntax for a SQL stored procedure. In the SQL Server Enterprise Manager, create a new stored procedure in the Northwind database named `GetCustomersByCountry`.

WARNING

Listing 3.6 shows a stored procedure called `GetCustomersByCountry`. This must be added to the Northwind database before the following samples will work.

LISTING 3.6 `GetCustomersByCountry` Stored Procedure for SQL Server Northwind Database

```
CREATE PROCEDURE [GetCustomersByCountry]
@country varchar (50)
AS
SELECT * FROM Customers WHERE Country = @country
```

When using a Managed Command to execute a stored procedure you must set the `CommandType` property. The default value of the `CommandType` property is `Text`. To execute a stored procedure you set the `CommandType` property to `CommandType.StoredProcedure` and pass in the name of the stored procedure.

Just as classic ADO command objects have a `Parameters` collection for passing parameters to the stored procedure, ADO.NET Managed Command classes (`SqlCommand` and `OleDbCommand`) also have a `Parameters` collection. You can call to a stored procedure, passing the required parameters in the collection using the following steps:

1. Create a Managed Command object (`SqlCommand` or `OleDbCommand`).
2. Set the Managed Command's `CommandType` property to `CommandType.StoredProcedure`.
3. Declare a parameter variable (`SqlParameter` or `OleDbParameter`).

4. Add a new instance of the parameter class to the Managed Command's `Parameters` collection, passing in its name and data type (`SqlDbType` for `SqlParameter` and `OleDbType` for `OleDbParameter—see` Appendix B for a list of valid `SqlDbTypes` and `OleDbTypes`).
5. Set the parameter's `Direction` property. (Optional—"Input" is the default.)
6. Set the Parameter's `Value` property.
7. Repeat steps 4-6 for additional parameters.
8. Execute the Managed Command.

Listing 3.7 demonstrates these steps.

WARNING

When you're using the SqlCommand object, the names of the parameters added to the Parameters collection must match the names of the markers in the stored procedure. The SQL Managed Provider will treat the parameters as named parameters, and it will look for markers in the stored procedure of the same name.

LISTING 3.7 Calling a Parameterized Stored Procedure with a SqlCommand

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  Dim myCommand As SqlCommand
  Dim myParameter As SqlParameter

  myConnection = New SqlConnection("server=localhost;
➥    database=Northwind; uid=sa; pwd=;")
  myCommand = New SqlCommand("GetCustomersByCountry", myConnection)
  myCommand.CommandType = CommandType.StoredProcedure
  myParameter = myCommand.Parameters.Add(New SqlParameter("@country",
➥    SqlDbType.VarChar, 50))
  myParameter.Direction = ParameterDirection.Input
  myParameter.Value = "USA"
  myConnection.Open()
  Dim myDataReader As SqlDataReader = myCommand.ExecuteReader()

  myDataGrid.DataSource = myDataReader
  myDataGrid.DataBind()

```

LISTING 3.7 Continued

```
  myConnection.Close()
 End Sub
</script>
```

[C#]

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlConnection myConnection;
  SqlCommand myCommand;
  SqlParameter myParameter;

  myConnection = new SqlConnection("server=localhost;
➥   database=Northwind; uid=sa; pwd=;");
  myCommand = new SqlCommand("GetCustomersByCountry", myConnection);
  myCommand.CommandType = CommandType.StoredProcedure;
  myParameter = myCommand.Parameters.Add(new SqlParameter("@country",
➥   SqlDbType.VarChar, 50));
  myParameter.Direction = ParameterDirection.Input;
  myParameter.Value = "USA";
  myConnection.Open();
  SqlDataReader myDataReader = myCommand.ExecuteReader();

  myDataGrid.DataSource = myDataReader;
  myDataGrid.DataBind();

  myConnection.Close();
 }
</script>
```

[VB & C#]

```
<html>
 <head>
  <title>Chapter 3: Managed Providers in ADO.NET</title>
 </head>
<body>
<form runat="server" method="post">
 <asp:DataGrid runat="server" id="myDataGrid" />
</form>
</body>
</html>
```

In Listing 3.7, you create a `SqlConnection`, `SqlCommand`, and `SqlDataReader`, the same as in previous listings. On line 8, you declare a `SqlParameter` object that will be used to create and add parameters to the `SqlCommand`. On line 13, you instantiate the `SqlParameter` object and set its name to `@country`, which is the same name given to the input parameter in the stored procedure in Listing 3.6. Additionally, you set the data type to `SqlDbType.VarChar, 50`, as required by the database table. On line 14, you set the `SqlParameter`'s direction to `ParameterDirection.Input`, and you set the `SqlParameter`'s `Value` to "USA" on line 15. Lastly, you open the connection and execute the command.

NOTE

In Listing 3.5 you created a series of new `DataRow` objects, and assigned their values after all the `DataRows` were created. The same approach can be used for `SqlParameters`.

```
myCommand.Parameters["@country"].Value = "USA";
```

NOTE

`Input` is the default direction for a Managed Command's parameter. There's no need to set a parameter's direction to `Input` explicitly. Listing 3.7 demonstrated how to set the parameter's `Direction` property. The possible values are `ParameterDirection.Input` and `ParameterDirection.Output`.

The DataAdapter

In Chapter 2, you looked at the `DataSet` as a collection of `DataTable objects`. You used the `DataTables` in the `DataSet` to populate one or more server controls on an ASP.NET Web Form. The *`DataAdapter`* is the bridge between the `DataSet` and the data store.

Unlike the past model of connection-based data processing, the `DataAdapter` works on a disconnected message-based model, revolving around and delivering chunks of info-rmation in a disconnected fashion. The `DataAdapter` is made up of four command methods, a `TableMappings` collection, a `Command` collection, and an `Exception` collection (for OleDbErrors). Like the other objects in the Managed Providers, the `DataAdapter` comes in two stock flavors, the `SqlDataAdapter` and the `OleDbDataAdapter`. Figure 3.4 illustrates the `DataAdapter` Object Model.

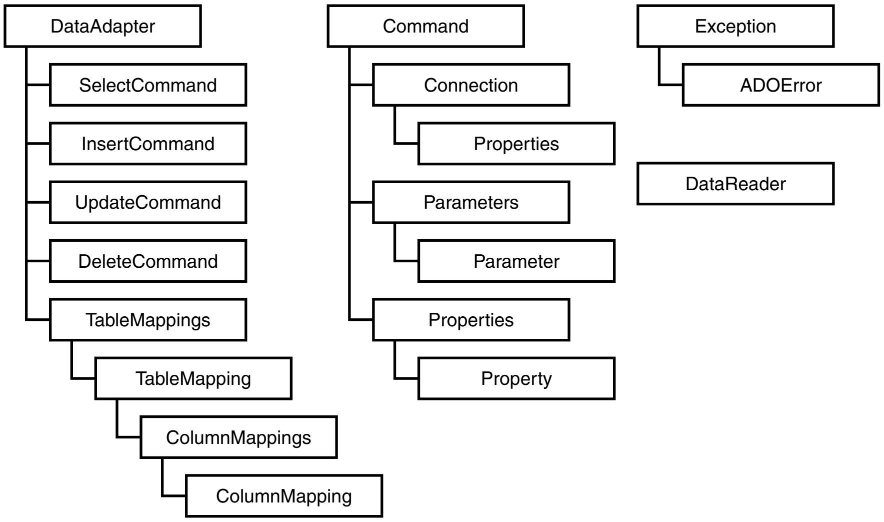


FIGURE 3.4
The `DataAdapter` *Object Model.*

The primary function of a `DataAdapter` is to retrieve data from a data store and push it into a `DataTable` in the `DataSet`. To complete this task, the `DataAdapter` requires two pieces of information, or parameters:

- A Managed Connection
- A Select Command

The `DataAdapter` constructor can accept either the command and connection values as text, or a Managed Command object as a single parameter. Listing 3.7 demonstrated constructing a `DataAdapter` with text values, while Listing 3.8 demonstrates constructing the `DataAdapter` with a single Managed Command.

LISTING 3.8 Creating a `SqlDataAdapter` with Connection and Command Text Values

```
[VB]

01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  Sub Page_Load(Sender As Object, E As EventArgs)
06:   Dim myDataAdapter As SqlDataAdapter
07:   Dim myDataSet As New DataSet
08:   myDataAdapter = New SqlDataAdapter("SELECT * FROM Customers",
➥     "server=localhost; database=Northwind; uid=sa; pwd=;")
09:   myDataAdapter.Fill(myDataSet, "Customers")
```

LISTING 3.8 Continued

```
10:   myDataGrid.DataSource = myDataSet.Tables("Customers").DefaultView
11:   myDataGrid.DataBind()
12:  End Sub
13: </script>
```

```
[C#]
```

```
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  void Page_Load(Object sender, EventArgs e){
06:   SqlDataAdapter myDataAdapter;
07:   DataSet myDataSet = new DataSet();
08:   myDataAdapter = new SqlDataAdapter("SELECT * FROM Customers",
➥     "server=localhost; database=Northwind; uid=sa; pwd=;");
09:   myDataAdapter.Fill(myDataSet, "Customers");
10:   myDataGrid.DataSource = myDataSet.Tables["Customers"].DefaultView;
11:   myDataGrid.DataBind();
12:  }
13: </script>
```

```
[VB & C#]
```

```
14: <html>
15: <body>
16: <form runat="server" method="post">
17:  <asp:DataGrid runat="server" id="myDataGrid" />
18: </form>
19: </body>
20: </html>
```

In Listing 3.8 you create an instance of the `SqlDataAdapter` class. When instantiating the class, on line 8 you pass in the command and connection values as text. The `DataAdapter` uses these values to create `SqlCommand` and `SqlConnection` objects behind the scenes. These objects are used to connect to the database and retrieve the appropriate data.

In Listing 3.9 you achieve the same result as in Listing 3.8, using explicit `SqlCommand` and `SqlConnection` objects.

LISTING 3.9 Creating an `SqlDataAdapter` with Connection and Command Objects

```
[VB]
```

```
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
```

LISTING 3.9 Continued

```
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myConnection As SqlConnection
  Dim myCommand As SqlCommand
  Dim myDataAdapter As SqlDataAdapter
  Dim myDataSet As New DataSet
  myConnection = New SqlConnection("server=localhost;
➥    database=Northwind; uid=sa; pwd=;")
  myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
  myDataAdapter = New SqlDataAdapter(myCommand)
  myDataAdapter.Fill(myDataSet, "Customers")
  myDataGrid.DataSource = myDataSet.Tables("Customers")
  myDataGrid.DataBind()
 End Sub
</script>
```

[C#]

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlConnection myConnection;
  SqlCommand myCommand;
  SqlDataAdapter myDataAdapter;
  DataSet myDataSet = new DataSet();
  myConnection = new SqlConnection("server=localhost;
➥    database=Northwind; uid=sa; pwd=;");
  myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
  myDataAdapter = new SqlDataAdapter(myCommand);
  myDataAdapter.Fill(myDataSet, "Customers");
  myDataGrid.DataSource = myDataSet.Tables["Customers"];
  myDataGrid.DataBind();
 }
</script>
```

[VB & C#]

```
<html>
<body>
<form runat="server" method="post">
 <asp:DataGrid runat="server" id="myDataGrid" />
</form>
</body>
</html>
```

In Listing 3.9 you explicitly create `SqlCommand` and `SqlConnection` objects. The `SqlConnection` object is used when constructing the `SqlCommand` object, and the `SqlCommand` object is passed into the `SqlDataAdapter` when it is instantiated.

While this might seem logical, it is more efficient to create the `DataAdapter` using the text values. The `DataAdapter` will manage the creation and destruction of the connection and command objects it requires. The explicit creation of these objects is only useful if you will be using either or both of them again, separate of the `DataAdapter`.

`DataAdapter.Fill()` Method

In Listings 3.8 and 3.9, you used various techniques and languages to create an instance of the Managed Provider `DataAdapter`. In one fashion or another, you created the `DataAdapter` and passed it values for the `SelectCommand` and `Connection` properties (either as inline values or as objects). Once the `DataAdapter` was created, the `Fill()` method of the `DataAdapter` was called.

The `DataAdapter.`*`Fill()`* method is like the switch that makes it go. Up until the `Fill()` method is called, the `DataAdapter` is idle. When the `Fill()` method is called, the connection to the database is made, and the SQL statement is executed. The results from the execution are filled into a `DataSet`, specified as a parameter of the `Fill()` method.

Specifying only a `DataSet` to fill the result set will cause a new `DataTable` object to be created in the `DataSet`. The `DataTable` is then accessible by its index value.

```
DataAdapter.Fill([DataSet])
DataGrid.DataSource = DataSet.Tables(0)
```

Optionally, you can also pass in a string value representing the name you would like assigned to the `DataTable` that is created. This makes your code easier to follow and more readable, as you can then access the `DataTable` by name rather than by its index value.

```
DataAdapter.Fill([DataSet], "[Table Name]")
DataGrid.DataSource = DataSet.Tables("[Table Name]")
```

On line 13 of Listing 3.9 you invoke the `Fill()` method of the `DataAdapter` and fill the results of the executed SQL statement into an empty `DataSet`, creating a new `DataTable` named “Customers”.

When the `Fill()` method is invoked, the bridge to the data store is extended. Then the data is retrieved and brought back to the calling application in the form of an XML file. This XML file is materialized as a `DataTable` in the `DataSet` you specified. The `DataTable` schema (table/column/primary key definitions) will be created automatically based on the schema of the database. Figure 3.5 illustrates the process when you’re invoking the `DataAdapter.Fill()` method.

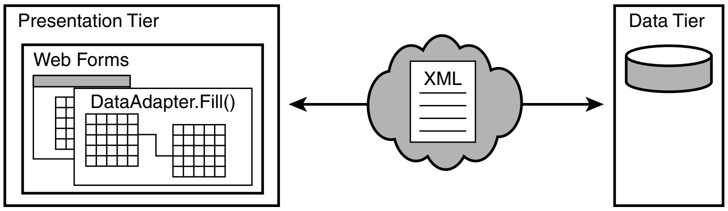


FIGURE 3.5
Invoking the `Fill()` *method of the* `DataAdapter` *class causes a bridge to be extended to the data store and the results to be returned to the calling application in the form of an XML file. The XML file is materialized as a* `DataTable` *in a* `DataSet`.

While the `Fill()` method will create the `DataTable` dynamically, it can also fill an existing `DataTable` that you create explicitly, as shown in Listing 3.10.

LISTING 3.10 Using the DataAdapter with an Existing DataTable

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myDataAdapter As SqlDataAdapter
  Dim myDataSet As New DataSet

  myDataSet.Tables.Add(New DataTable("Customers"))
  myDataSet.Tables("Customers").Columns.Add("CompanyName",
➥    System.Type.GetType("System.String"))
  myDataSet.Tables("Customers").Columns.Add("ContactName",
➥    System.Type.GetType("System.String"))
  myDataSet.Tables("Customers").Columns.Add("Region",
➥    System.Type.GetType("System.String"))

  myDataAdapter = New SqlDataAdapter("SELECT
➥    CompanyName, ContactName, Region FROM Customers",
➥    "server=localhost; database=Northwind; uid=sa; pwd=;")
  myDataAdapter.Fill(myDataSet, "Customers")
  myDataGrid.DataSource = myDataSet.Tables("Customers")
  myDataGrid.DataBind()
 End Sub
</script>
```

LISTING 3.10 Continued

```
[C#]

01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  void Page_Load(Object sender, EventArgs e){
06:   SqlDataAdapter myDataAdapter;
07:   DataSet myDataSet = new DataSet();
08:
09:   myDataSet.Tables.Add(new DataTable("Customers"));
10:   myDataSet.Tables["Customers"].Columns.Add("CompanyName",
➥      System.Type.GetType("System.String"));
11:   myDataSet.Tables["Customers"].Columns.Add("ContactName",
➥      System.Type.GetType("System.String"));
12:   myDataSet.Tables["Customers"].Columns.Add("Region",
➥      System.Type.GetType("System.String"));
13:
14:   myDataAdapter = new SqlDataAdapter("SELECT
➥      CompanyName, ContactName, Region FROM Customers",
➥      "server=localhost; database=Northwind; uid=sa; pwd=;");
15:   myDataAdapter.Fill(myDataSet, "Customers");
16:   myDataGrid.DataSource = myDataSet.Tables["Customers"];
17:   myDataGrid.DataBind();
18:  }
19: </script>

[VB & C#]

20: <html>
21: <body>
22: <form runat="server" method="post">
23:  <asp:DataGrid runat="server" id="myDataGrid" />
24: </form>
25: </body>
26: </html>
```

In Listing 3.10 you create a `DataTable` explicitly on lines 9–12. Using the `Fill()` method of the `DataAdapter` you fill this newly created `DataTable` with the results from the SQL statement execution. This is done by calling the `Fill()` method and passing in the `DataSet` and `DataTable` name for the `DataTable` you just created. If data already exists in the `DataTable`, then the `Fill()` method will update, or add rows to the `DataTable`. You can use the same `DataAdapter`, change its `SelectCommand.CommandText` property, and invoke the `Fill()` method again. In Listing 3.11 you use the `DataAdapter` to return two different result sets from similar SQL statements. You use the `Fill()` method to add the records to the same `DataTable` in the `DataSet`.

LISTING 3.11 Using the `Fill()` Method to Add Records to a `DataTable`

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myDataAdapter As SqlDataAdapter
  Dim myDataSet As New DataSet

  myDataSet.Tables.Add(New DataTable("Customers"))
  myDataSet.Tables("Customers").Columns.Add("CompanyName",
➥   System.Type.GetType("System.String"))
  myDataSet.Tables("Customers").Columns.Add("ContactName",
➥   System.Type.GetType("System.String"))
  myDataSet.Tables("Customers").Columns.Add("Region",
➥   System.Type.GetType("System.String"))

  myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",
➥   "server=localhost; database=Northwind; uid=sa; pwd=;")
  myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure
  myDataAdapter.SelectCommand.Parameters.Add(new
➥   SqlParameter("@country", SqlDbType.VarChar, 50))
  myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
  myDataAdapter.Fill(myDataSet, "Customers")

  myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
  myDataAdapter.Fill(myDataSet, "Customers")

  myDataGrid.DataSource = myDataSet.Tables("Customers")
  myDataGrid.DataBind()
 End Sub
</script>

[C#]

<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlDataAdapter myDataAdapter;
  DataSet myDataSet = new DataSet();

  myDataSet.Tables.Add(new DataTable("Customers"));
```

LISTING 3.11 Continued

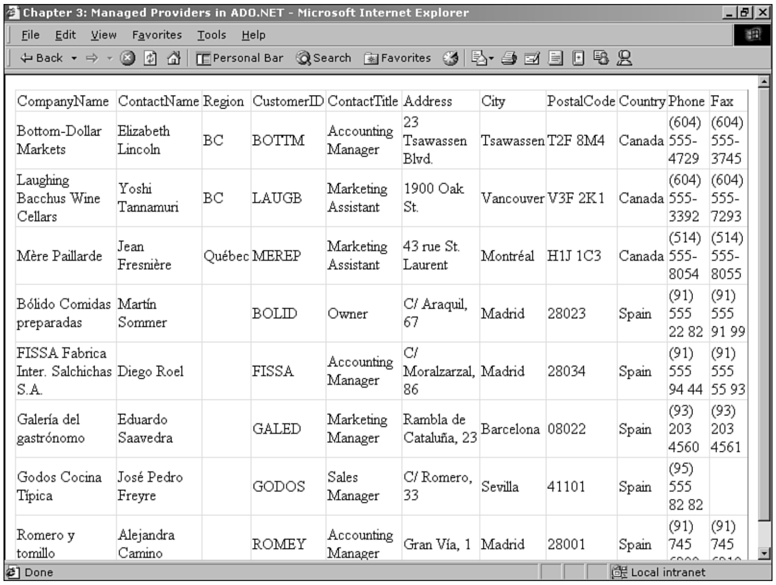
```
10:   myDataSet.Tables["Customers"].Columns.Add("CompanyName",
➥       System.Type.GetType("System.String"));
11:   myDataSet.Tables["Customers"].Columns.Add("ContactName",
➥       System.Type.GetType("System.String"));
12:   myDataSet.Tables["Customers"].Columns.Add("Region",
➥       System.Type.GetType("System.String"));
13:
14:   myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",
➥       "server=localhost; database=Northwind; uid=sa; pwd=;");
15:   myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
16:   myDataAdapter.SelectCommand.Parameters.Add(new
➥       SqlParameter("@country", SqlDbType.VarChar, 50));
17:   myDataAdapter.SelectCommand.Parameters["@country"].Value = "Canada";
18:   myDataAdapter.Fill(myDataSet, "Customers");
19:
20:   myDataAdapter.SelectCommand.Parameters["@country"].Value = "Spain";
21:   myDataAdapter.Fill(myDataSet, "Customers");
22:
23:   myDataGrid.DataSource = myDataSet.Tables["Customers"];
24:   myDataGrid.DataBind();
25:  }
26: </script>
```

```
[VB & C#]
```

```
27: <html>
28: <body>
29: <form runat="server" method="post">
30:  <asp:DataGrid runat="server" id="myDataGrid" />
31: </form>
32: </body>
33: </html>
```

In Listing 3.11 you use the `DataAdapter` to select a small set of records from the database (line 14) with the stored procedure from Listing 3.6. On line 15 you specify that the `SelectCommand` property of the `DataAdapter` is a stored procedure. On line 16 you add a parameter for the expected “@country” parameter in the stored procedure. On line 17 you set the value of the parameter to “Canada.” Using the `Fill()` method you add those records to the Customers `DataTable` on line 18. On line 20 you change the parameter’s value to “Spain.” Since you are using the same `DataAdapter` to execute the second `Fill()` method, you do not need to set the `ConnectionString` property, or recreate the parameter you are using. Using the `Fill()` method, on line 18, you add the new result set to the existing records in the Customers table. Figure 3.6 shows the result of executing the code in Listing 3.11.

As you've learned, the `DataAdapter` works on a disconnected message-based model. You can reuse the `DataAdapter` to fill additional `DataTables` in the same `DataSet`, or in other `DataSets`, because there's no physical link between the `DataAdapter` and the `DataSet` or `DataTable`. You only need to change the `SelectCommand` property of the `DataSet` (if you want to use a new SQL statement), or change a parameter value (if you want to use the same SQL statement), and call the `Fill()` method, passing it the new `DataSet` and `DataTable` name (see Listing 3.12).


Chapter 3: Managed Providers in ADO.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Done
Local intranet

CompanyName	ContactName	Region	CustomerID	ContactTitle	Address	City	PostalCode	Country	Phone	Fax
Bottom-Dollar Markets	Elizabeth Lincoln	BC	BOTTM	Accounting Manager	23 Tsawassen Blvd.	Tsawassen	T2F 8M4	Canada	(604) 555-4729	(604) 555-3745
Laughing Bacchus Wine Cellars	Yoshi Tannamuri	BC	LAUGB	Marketing Assistant	1900 Oak St.	Vancouver	V3F 2K1	Canada	(604) 555-3392	(604) 555-7293
Mère Paillarde	Jean Fresnière	Québec	MEREP	Marketing Assistant	43 rue St. Laurent	Montréal	H1J 1C3	Canada	(514) 555-8054	(514) 555-8055
Bólido Comidas preparadas	Martín Sommer		BOLID	Owner	C/ Araquil, 67	Madrid	28023	Spain	(91) 555 22 82	(91) 555 91 99
FISSA Fabrica Inter. Salchichas S.A.	Diego Roel		FISSA	Accounting Manager	C/ Moralzarzal, 86	Madrid	28034	Spain	(91) 555 94 44	(91) 555 55 93
Galería del gastrónomo	Eduardo Saavedra		GALED	Marketing Manager	Rambla de Cataluña, 23	Barcelona	08022	Spain	(93) 203 4560	(93) 203 4561
Godos Cocina Típica	José Pedro Freyre		GODOS	Sales Manager	C/ Romero, 33	Sevilla	41101	Spain	(95) 555 82 82	
Romero y tomillo	Alejandra Camino		ROMEY	Accounting Manager	Gran Vía, 1	Madrid	28001	Spain	(91) 745	(91) 745

FIGURE 3.6
The `DataAdapter.Fill()` *method can be used to add records to a* `DataTable`*, or update existing records.*

LISTING 3.12 Adding a Second `DataTable` to a `DataSet`

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myDataAdapter As SqlDataAdapter
  Dim myDataSet As New DataSet

  myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",
➥    "server=localhost; database=Northwind; uid=sa; pwd=;")
```

LISTING 3.12 Continued

```
  myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure
  myDataAdapter.SelectCommand.Parameters.Add(new
➥   SqlParameter("@country", SqlDbType.VarChar, 50))
  myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
  myDataAdapter.Fill(myDataSet, "Canada_Customers")

  myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
  myDataAdapter.Fill(myDataSet, "Spain_Customers")

  myDataGrid.DataSource = myDataSet.Tables("Canada_Customers")
  myDataGrid.DataBind()

  myOtherDataGrid.DataSource = myDataSet.Tables("Spain_Customers")
  myOtherDataGrid.DataBind()
 End Sub
</script>
```

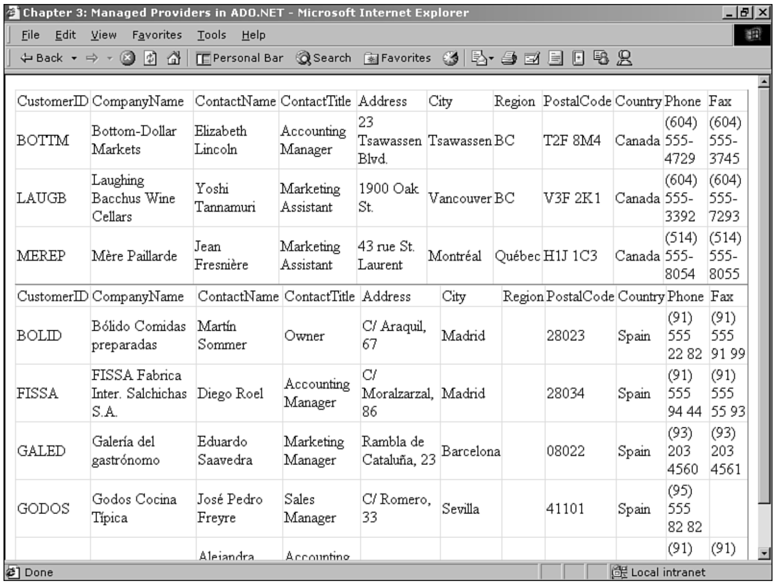
[C#]

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlDataAdapter myDataAdapter;
  DataSet myDataSet = new DataSet();

  myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",
➥   "server=localhost; database=Northwind; uid=sa; pwd=;");
  myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
  myDataAdapter.SelectCommand.Parameters.Add(new
➥   SqlParameter("@country", SqlDbType.VarChar, 50));
  myDataAdapter.SelectCommand.Parameters["@country"].Value = "Canada";
  myDataAdapter.Fill(myDataSet, "Canada_Customers");

  myDataAdapter.SelectCommand.Parameters["@country"].Value = "Spain";
  myDataAdapter.Fill(myDataSet, "Spain_Customers");

  myDataGrid.DataSource = myDataSet.Tables["Canada_Customers"];
  myDataGrid.DataBind();

  myOtherDataGrid.DataSource = myDataSet.Tables["Spain_Customers"];
  myOtherDataGrid.DataBind();
 }
</script>
```

LISTING 3.12 Continued

```
[VB & C#]

<html>
<form runat="server" method="post">
<body>
 <asp:DataGrid runat="server" id="myDataGrid" />
 <asp:DataGrid runat="server" id="myOtherDataGrid" />
</form>
</body>
</html>
```

In Listing 3.12 you use the same data access code from Listing 3.11. You use the `Fill()` method to create a new `DataTable` named `Canada_Customers`. Then, by resetting the parameter value you retrieve another set of data from the database. Using the `Fill()` method you create a second `DataTable` named `Spain_Customers`. Finally you bind each of these `DataTables` to a separate `DataGrid`. The resulting page is shown in Figure 3.7.


Chapter 3: Managed Providers in ADO.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Favorites
Done
Local intranet

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Tsawassen	BC	T2F 8M4	Canada	(604) 555-4729	(604) 555-3745
LAUGB	Laughing Bacchus Wine Cellars	Yoshi Tannamuri	Marketing Assistant	1900 Oak St.	Vancouver	BC	V3F 2K1	Canada	(604) 555-3392	(604) 555-7293
MEREP	Mère Paillarde	Jean Fresnière	Marketing Assistant	43 rue St. Laurent	Montréal	Québec	H1J 1C3	Canada	(514) 555-8054	(514) 555-8055

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid		28023	Spain	(91) 555 22 82	(91) 555 91 99
FISSA	FISSA Fabrica Inter. Salchichas S.A.	Diego Roel	Accounting Manager	C/ Moralzarzal, 86	Madrid		28034	Spain	(91) 555 94 44	(91) 555 55 93
GALED	Galería del gastrónomo	Eduardo Saavedra	Marketing Manager	Rambla de Cataluña, 23	Barcelona		08022	Spain	(93) 203 4560	(93) 203 4561
GODOS	Godos Cocina Típica	José Pedro Freyre	Sales Manager	C/ Romero, 33	Sevilla		41101	Spain	(95) 555 82 82	
		Alejandra	Accounting						(91)	(91)

FIGURE 3.7
The `DataAdapter` *can be used to create multiple* `DataTables` *in one or more* `DataSets`. *This is allowed because the* `DataAdapter` *is not explicitly tied to a single* `DataSet` *or* `DataTable`.

Table and Column Mappings

Table and column mappings enable you to alter the schema of the `DataTable` that's dynamically created in the `DataSet`. In the previous listings you filled the `DataSet`, taking in the schema provided to you by the data store. There are certainly instances when you'll want to change this. Often, database column names can be a bit cryptic, and changing them can make writing your code easier. Table and column mappings allow you to create a master mapping between the data returned from the data store and the `DataTable` in the `DataSet`. The `DataSet` maintains the table and column mappings and can translate them back to their original names when reconciling the data with the data store.

NOTE

In the previous listings, you've been filling a `DataSet` with the Customers table from the Northwind database. In the following listings, you'll create a new `TableMapping` for the Authors table in the Pubs database. Pubs is a sample database installed with Microsoft SQL Server. I chose to switch to it because of its notoriously cryptic naming conventions.

In the following example, you'll create table and column mappings for the Authors table in the Pubs database. When you add a new `TableMapping`, you must pass in the `TableMapping` source name and the `DataTable` name. If you specify "Table" (the default) as the source name, when you fill the `DataSet` you do not need to pass in the `DataTable` name. The data retrieved will use the `TableMappings` for "Table" by default, and a `DataTable` with the name you specified when you created the `TableMappings` will be created. This is demonstrated in Listing 3.13.

LISTING 3.13 Creating Table and Column Mappings for the Default Table

```
[VB]

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 Sub Page_Load(Sender As Object, E As EventArgs)
  Dim myDataAdapter As SqlDataAdapter
  Dim myDataSet As New DataSet

  myDataAdapter = new SqlDataAdapter("SELECT * FROM Authors",
➥    "server=localhost; database=Pubs; uid=sa; pwd=;")

```

LISTING 3.13 Continued

```
  myDataAdapter.TableMappings.Add("Table", "Authors")
  With myDataAdapter.TableMappings("Table").ColumnMappings
    .Add("au_id", "ID")
    .Add("au_lname", "Last Name")
    .Add("au_fname", "First Name")
    .Add("phone", "Phone")
    .Add("address", "Address")
    .Add("city", "City")
    .Add("state", "State")
    .Add("zip", "Zipcode")
    .Add("contract", "Contract")
  End With

  myDataAdapter.Fill(myDataSet)

  myDataGrid.DataSource = myDataSet.Tables("Authors")
  myDataGrid.DataBind()
 End Sub
</script>
```

[C#]

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
 void Page_Load(Object sender, EventArgs e){
  SqlDataAdapter myDataAdapter;
  DataSet myDataSet = new DataSet();

  myDataAdapter = new SqlDataAdapter("SELECT * FROM Authors",
➥   "server=localhost; database=Pubs; uid=sa; pwd=;");

  myDataAdapter.TableMappings.Add("Table", "Authors");

  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("au_id", "ID");
  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("au_lname", "Last Name");
  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("au_fname", "First Name");
  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("phone", "Phone");
  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("address", "Address");
  myDataAdapter.TableMappings["Table"].
➥   ColumnMappings.Add("city", "City");
```

LISTING 3.13 Continued

```
19:   myDataAdapter.TableMappings["Table"].
➥       ColumnMappings.Add("state", "State");
20:   myDataAdapter.TableMappings["Table"].
➥       ColumnMappings.Add("zip", "Zipcode");
21:   myDataAdapter.TableMappings["Table"].
➥       ColumnMappings.Add("contract", "Contract");
22:
23:
24:   myDataAdapter.Fill(myDataSet);
25:
26:   myDataGrid.DataSource = myDataSet.Tables["Authors"];
27:   myDataGrid.DataBind();
28:  }
29: </script>
```

```
[VB & C#]
```

```
30: <html>
31: <body>
32: <form runat="server" method="post">
33:  <asp:DataGrid runat="server" id="myDataGrid" />
34:  <asp:DataGrid runat="server" id="myOtherDataGrid" />
35: </form>
36: </body>
37: </html>
```

On line 9, you create a new `SqlDataAdapter`, selecting all of the fields in the Authors table of the Pubs database. On line 11, you add a new default table mapping by passing in the name “Table” as the first parameter. Any table created in the `DataSet` that doesn’t have a name specified in the `Fill()` method will use this table mapping and will be given the name “Authors”, as indicated on line 11. On lines 13–21 you create the column mappings for all of the fields in the Authors table. The first parameter passed into the `ColumnMappings` collection’s `Add()` method is the name of the field in the database. The second parameter is the name you’re giving the field in the `DataTable`.

In case you’d rather not alter the `DataSet`’s default table and column mappings, the `DataAdapter` allows you to add named table and column mappings. Rather than specifying “Table” as the source name, you can provide a new source name. When you call the `Fill()` method, you pass in the new source name as the table parameter. The `DataSet` checks for a table mapping for the source name passed in. If no table mapping exists, the schema is built on the fly based on the data store’s schema, the same as in earlier examples. If there’s a table mapping for the name passed in, it’s used.

```
11:  myDataAdapter.TableMappings.Add("BookAuthors", "BookAuthors");
```

```
24:  myDataAdapter.Fill(myDataSet, "BookAuthors");
```

Line 11 shows a table mapping named “BookAuthors”, specifying the name “BookAuthors” to be used when the `DataTable` is created. (Lines 12–23 did not change from Listing 3.13.) On line 24, you call the `Fill()` method, passing in the name of your table mapping (“BookAuthors”). The `DataSet` finds this table mapping and creates a `DataTable` named “BookAuthors”, using the schema you created.

Summary

Chapter 1 looked at what ASP.NET is, and Chapter 2 looked at what ADO.NET is. This chapter pulled the two pieces together.

In this chapter you learned that the ADO.NET Managed Providers are the bridge from a data store to your data-driven application. The Managed Providers come in two stock flavors: the SQL Managed Provider and the OleDb Managed Provider. The SQL Managed Provider is used to connect directly to a Microsoft SQL Server database (version 7.0 or higher), and bypasses OLEDB to provider better performance. The OleDb Managed Provider is used to connect to non-Microsoft SQL Server databases, such as Access, Oracle, and a host of others.

In this chapter you created connections to a database, built command objects to execute SQL statements on the database, and used the `DataReader` to iterate through data before binding it to a server control. You also learned about the `DataAdapter` class; a specific class for bridging between the Web application and the database to return the result set as a `DataTable` in a `DataSet`.

Throughout this chapter you built sample Web forms that connected to the database and returned records as either a data stream (`DataReader`), or a `DataTable` (`DataAdapter`).

By now you should be comfortable with the two Managed Providers and should be ready to start building data-driven Web applications. For the rest of this book you’ll be working with both the Managed Providers. You’ll be pulling data into an application with the Managed Command or the `DataAdapter`, and you’ll use the `DataSet` to persist the data, or the `DataReader` to iterate through the data.

If it felt as though you covered a lot in this chapter, you did. You’ll be using it repetitively throughout this book, so don’t worry. . . you’ll get lots of practice.