Using Direct Data Access

The most straightforward way to interact with a database is to use *direct data access*. When you use direct data access, you're in charge of building an SQL command (like the ones you considered earlier in this chapter) and executing it. You use commands to query, insert, update, and delete information.

When you query data with direct data access, you don't keep a copy of the information in memory. Instead, you work with it for a brief period of time while the database connection is open, and then close the connection as soon as possible. This is different from disconnected data access, where you keep a copy of the data in the DataSet object so you can work with it after the database connection has been closed.

The direct data model is well suited to ASP.NET web pages, which don't need to keep a copy of their data in memory for long periods of time. Remember, an ASP.NET web page is loaded when the page is requested and shut down as soon as the response is returned to the user. That means a page typically has a lifetime of only a few seconds (if that).

■ **Note** Although ASP.NET web pages don't need to store data in memory for ordinary data management tasks, they just might use this technique to optimize performance. For example, you could get the product catalog from a database once, and keep that data in memory on the web server so you can reuse it when someone else requests the same page. This technique is called *caching*, and you'll learn to use it in Chapter 23.

To query information with simple data access, follow these steps:

1. Create Connection, Command, and DataReader objects.
2. Use the DataReader to retrieve information from the database, and display it in a control on a web form.
3. Close your connection.
4. Send the page to the user. At this point, the information your user sees and the information in the database no longer have any connection, and all the ADO.NET objects have been destroyed.

To add or update information, follow these steps:

1. Create new Connection and Command objects.
2. Execute the Command (with the appropriate SQL statement).

This chapter demonstrates both of these approaches. Figure 14-8 shows a high-level look at how the ADO.NET objects interact to make direct data access work.

1

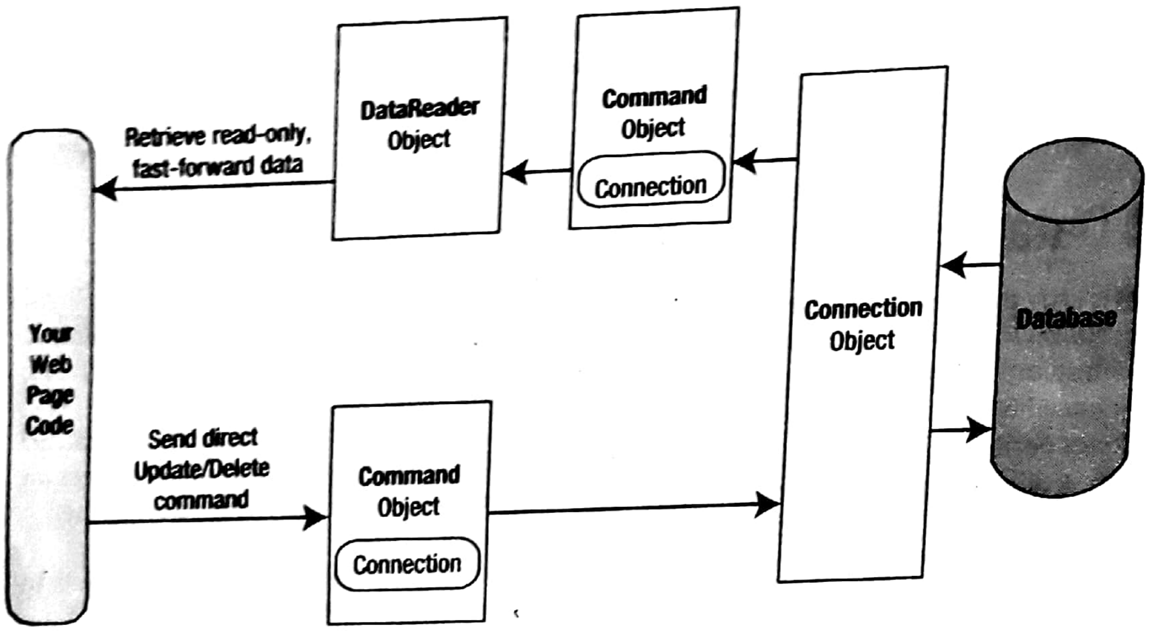

Figure 14-8. *Direct data access with ADO.NET*

Before continuing, make sure you import the ADO.NET namespaces. In this chapter, we assume you're the SQL Server provider, in which case you need these two namespace imports:

```
using System.Data;
using System.Data.SqlClient;
```

Creating a Connection

Before you can retrieve or update data, you need to make a connection to the data source. Generally, connect are limited to some fixed number, and if you exceed that number (either because you run out of licenses or because your database server can't accommodate the user load), attempts to create new connections will fail. that reason, you should try to hold a connection open for as short a time as possible. You should also write yo database code inside a try/catch error-handling structure so you can respond if an error does occur, and make sure you close the connection even if you can't perform all your work.

When creating a Connection object, you need to specify a value for its ConnectionString property. This ConnectionString defines all the information the computer needs to find the data source, log in, and choose ar initial database. Out of all the details in the examples in this chapter, the ConnectionString is the one value you might have to tweak before it works for the database you want to use. Luckily, it's quite straightforward. Here's a example that uses a connection string to connect to SQL Server:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = "Data Source=localhost;" +
"Initial Catalog=Pubs;Integrated Security=SSPI";
```

If you're using SQL Server Express, your connection string will use the instance name, as shown here:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = @"Data Source=(localdb)\v11.0;" +
"Initial Catalog=Pubs;Integrated Security=SSPI";
```

Using Disconnected Data Access

When you use *disconnected data access*, you use the DataSet to keep a copy of your data in memory. You connect to the database just long enough to fetch your data and dump it into the DataSet, and then you disconnect immediately.

There are a variety of good reasons to use the DataSet to hold onto data in memory. Here are a few:

- You need to do something time-consuming with the data. By dumping it into a DataSet first, you ensure that the database connection is kept open for as little time as possible.
- You want to use ASP.NET data binding to fill a web control (such as a GridView) with your data. Although you can use the DataReader, it won't work in all scenarios. The DataSet approach is more straightforward.
- You want to navigate backward and forward through your data while you're processing it. This isn't possible with the DataReader, which goes in one direction only—forward.
- You want to navigate from one table to another. Using the DataSet, you can store several tables of information. You can even define relationships that allow you to browse through them more efficiently.
- You want to save the data to a file for later use. The DataSet includes two methods—WriteXml() and ReadXml()—that allow you to dump the content to a file and convert it back to a live database object later.
- You need a convenient package to send data from one component to another. For example, in Chapter 22 you'll learn to build a database component that provides its data to a web page by using the DataSet. A DataReader wouldn't work in this scenario, because the database component would need to leave the database connection open, which is a dangerous design.
- You want to store some data so it can be used for future requests. Chapter 23 demonstrates how you can use caching with the DataSet to achieve this result.

2

UPDATING DISCONNECTED DATA

The DataSet tracks the changes you make to the records inside. This allows you to use the DataSet to update records. The basic principle is simple. You fill a DataSet in the normal way, modify one or more records, and then apply your update by using a DataAdapter.

However, ADO.NET's disconnected update feature makes far more sense in a desktop application than in a web application. Desktop applications run for a long time, so they can efficiently store a batch of changes and perform them all at once. But in a web application, you need to commit your changes the moment they happen. Furthermore, the point at which you retrieve the data (when a page is first requested) and the point at which it's changed (during a postback) are different, which makes it very difficult to use the same DataSet object, and maintain the change-tracking information for the whole process.

For these reasons, ASP.NET web applications often use the DataSet to store data but rarely use it to make updates. Instead, they use direct commands to commit changes. This is the model you'll see in this book.

Selecting Disconnected Data

With disconnected data access, a copy of the data is retained in memory while your code is running. Figure 14-15 shows a model of the DataSet.

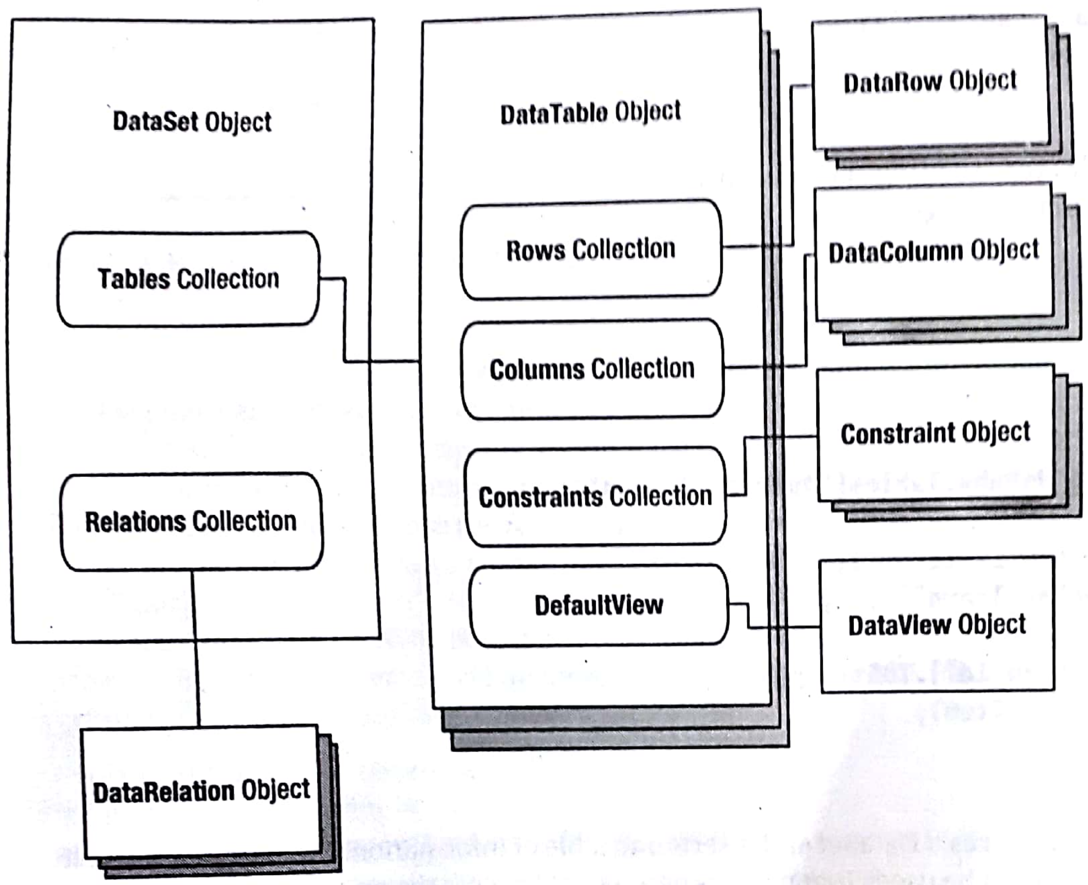

Figure 14-15. *The DataSet family of objects*

You fill the DataSet in much the same way that you connect a DataReader. However, although the DataReader holds a live connection, information in the DataSet is always disconnected.

The following example shows how you could rewrite the FillAuthorList() method from the earlier example t use a DataSet instead of a DataReader. The changes are highlighted in bold.

```
private void FillAuthorList()
{
    lstAuthor.Items.Clear();

    // Define ADO.NET objects.
    string selectSQL;
    selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);
    DataSet dsPubs = new DataSet();

    // Try to open database and read information.
    try
    {
        con.Open();

        // All the information in transferred with one command.
        // This command creates a new DataTable (named Authors)
        // inside the DataSet.
        adapter.Fill(dsPubs, "Authors");
    }
    catch (Exception err)
    {
        lblStatus.Text = "Error reading list of names. ";
        lblStatus.Text += err.Message;
    }
    finally
    {
        con.Close();
    }

    foreach (DataRow row in dsPubs.Tables["Authors"].Rows)
    {
        ListItem newItem = new ListItem();
        newItem.Text = row["au_lname"] + ", " +
          row["au_fname"];
        newItem.Value = row["au_id"].ToString();
        lstAuthor.Items.Add(newItem);
    }
}
```

The DataAdapter.Fill() method takes a DataSet and inserts one table of information. In this case, the table is named Authors, but any name could be used. That name is used later to access the appropriate table in the DataSet.

To access the individual DataRows, you can loop through the Rows collection of the appropriate table. Each piece of information is accessed by using the field name, as it was with the DataReader.

}

Data Binding:-

→ Basic principle of data binding is this: you tell a control where to find your data and how you want it displayed, and the control handles the rest of the details.

3

→ Data binding in the case of Desktop appli-cations involves creating data connection b/w data source and the control.

→ If user makes changes to the control on screen data, the changes are immediately reflected in the linked database

→ If you made changes to the database, those changes are reflected in the bounded UI control automatically.

→ Data binding with ASP.NET is more complicate-d (because of web connections)

→ ASP.NET data binding works in one direction only. information moves from a data object into a control. Then the data objects are thrown away, and the page is sent to the client.

→ if the user ~~information~~ modifies the data in a data-bound control, your program can update the corresponding database.

Types of Data Binding

Single values (or) simple Data binding

Repeated value (or) List Binding

1) you can use single-value data binding to add into anywhere on an ASP.NET Page. you can even place information into a control property or as plaintext inside an HTML Tag.

→ Instead, single-value data binding allows you to take a variable, a property, or an expression and insert it dynamically into a Page.

→ Repeated value (or) List Binding

Repeated-value data binding allows you to display an entire table (or just a single field from a table). Unlike single-value data binding, this type of data binding requires a special control that supports it.

→ control ~~that~~ used can be list box, checkbox list, Grid view etc.

→ Types of data binding → single value, & Repeated value

→ As with single value binding repeated value binding doesn't necessarily need to use data from a database, and it doesn't have to use the ADO.NET objects for example you can use repeted value binding to bind data from a collection (or) an array

→ To use single-value binding, you must insert a data binding expression into the markup in the .aspx file (not the code behind file). To use repeated-value binding you must set one (or) more properties of a data control

→ once you specify data binding, you need to activate it you accomplish this task by calling the DataBind() method. the DataBind() method performs repeated-value data binding.

→ you can also bind the whole page at once by calling the DataBind() method of the current page object once you call this method, all the data binding expressions in the page are evaluated and replaced with the specified value.

Eg:-

```
<asp: Label ID="label" runat="server"/>
        UserID: <%#UserID%>
        UserName: <%#UserName%>
        city: <%#city%>.
```

.cs:-

```
protected void page_Load(object sender,
                          Event Args e)
        UserID=1;
        username = "Venu";
          city = "Pone";
          this.DataBind();
```

Using single values:-

→ single value data binding is really just a different approach to dynamic text. To use it, you add special data-bind expr into your .aspx file.

```
<%#expression-goes-here%>
```

Eg:-

1. <%#County%> → Coonty is protected variable

→ when you call the DataBind() method for the page, this text will be replaced with the value for country

2. <%# Request . Browser. Browser %> → gives browser name.

3. <%# 1+ (2*20) %> → gives 41

4. <%# GetUserName(ID)%>

Problems:-

1. Violation of code separation from presentation concept of ASP.NET

2. fragmenting code : if you use data binding to fill a control and also
→ modify that control directly incode, data binding will not work properly
→ if the page code changes ; or a variable (or) function is removed (or) renamed, the corresponding data binding expression could stop providing valid information without any explanation (or) even an obvious error, resulting in maintenance problem.

using Repeated Value Binding:

-> to use repeated-value binding, you link one of these contrd to a datasource.

-> when you call DataBind(), the control automa-tically creates a full list by using all the corresponding values. This saves you from writing code that loops through the array (or) data table and manually adds elements to a control List controls used in Repeated value binding

> list box
> HTML select
> Grid view, detailsview, listview

multiple binding

-> binding the same data list object to multiple controls is called multipleBinding.

```
list <string> fruits = new List <string>();
        fruits. Add ("Apple");
        fruits. Add ("Gua");
 istItems . Datasource = fruits;
        stItems . DataBind();
```