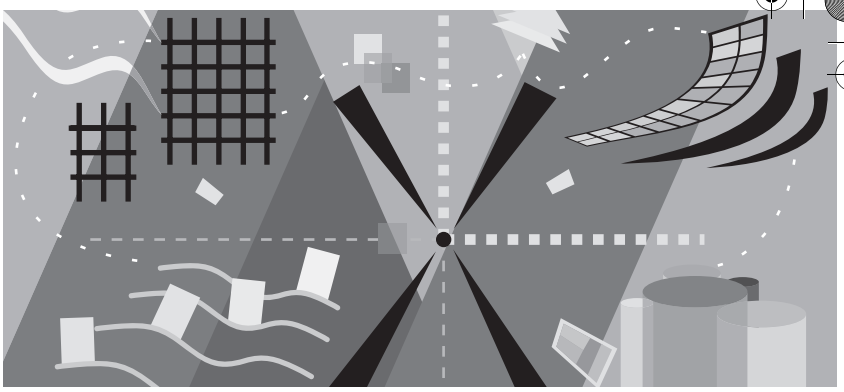


6

Data Access in .NET

An' home again, the Rio run: it's no child's play to go
Steamin' to bell for fourteen days o' snow an' floe an' blow—
The bergs like kelpies overside that girn an' turn an' shift
Whaur, grindin' like the Mills o' God, goes by the big South drift.
(Hail, snow an' ice that praise the lord: I've met them at their work,
An' wished we had anither route or they anither kirk.)

—Rudyard Kipling, writing on the perils of data access, "McAndrew's Hymn," 1894.

Problem Background

All Internet applications access remote data stores.

Unlike single desktop programs, which usually deal with documents on a user's local hard disk, essentially all distributed programs access remote data stores in some way. Remote data access is the main engine driving the phenomenal growth of the Internet—the incredible potential of easy access to data from anyone who wants to make it available. Sometimes the owner of that data charges money for the data itself. Pornographers were the first who made this business model really sing, as few users would fork over the bucks for any other type of content. Other businesses, such as the *Wall Street Journal* (porn for a different audience, some say) and the *Oxford English Dictionary*, are enjoying limited success with this model today, and mainstream music companies may eventually figure it out if they ever get their heads screwed on right. More often today, the owner of the data makes money by using the Internet's easy access to that data to lower the friction of existing

business processes, such as removing human employees from airline reservation systems or overnight package delivery tracking. Accessing remote data over the Internet is primarily why you have a PC today.

Data stores live in many different programs in many different locations.

Once you realize that the goals of most Internet applications differ radically from those of desktop programs, you won't be surprised to learn that we encounter different design problems when we write Internet apps. (Are you starting to see a pattern in this book?) First, the data that we want to see and perhaps change resides in many different locations and many different types of containers. I purposely selected the term *data stores* in this chapter's opening sentence instead of the more narrow *databases*. Certainly an enormous amount of data resides in large relational database programs such as Microsoft SQL Server or Oracle9*i*, but the data that an Internet app uses can and often does reside in many other locations. Some of these sources will be familiar to you, and only the notion of easy remote access will be new. For example, the financial data for my current house remodeling project lives in Microsoft Excel spreadsheets and Microsoft Money files on my hard disk. I'd like my architect and contractor to be able to read these files and update them with their latest cost overruns, and I'd like my banker to be able to read them and recoil in shock before handing over the money to cover the costs. Other data sources are new, and you might not have thought of them as data sources just a year ago. For example, the April 30, 2001, *Wall Street Journal* carried a story about a software product that reports the status of all the remote substations of an electric power utility company using the Web as a transport and display mechanism (oooh, baby, that feels SO good).

We want our many sources of data to look the same to a client program.

Naturally, the greater the number of different data sources, the more difficult becomes the task of writing client applications that access these different sources. We can't take the time to learn different programming models for every conceivable data store: one for SQL Server, a different one for Oracle, yet another for Excel—and heaven knows what programmatic interface those electric power guys are exposing to clients. This problem is especially bad for small-scale data providers because they don't have the clout to make clients learn their proprietary language, as some would argue that Microsoft and Oracle do. We need to have one basic programming model for accessing all types of data no matter where the data lives, otherwise we'll spend all our development budget dealing with different data access schemes and not have any resources left for writing any code that does useful work with the data once we've fetched it.

Our data access strategy needs to work well in the loosely coupled world of the Internet.

Internet data access programming is also difficult because of the heterogeneous and nondeterministic nature of the Internet environment. When a desktop PC accesses a database file on its own hard disk—say, in an application for a small dry-cleaning business looking for a missing garment—the

developer can depend on that access being fast because it uses the PC's internal bus. On the Internet, a similar request might have to travel over congested transmission lines and wait for the attention of overloaded servers. The request is slower and the speed varies from one access of data to another. A developer needs to write code to account for these various conditions. In addition, a data source and its client are coupled much more loosely over the Internet than they would be if they resided on the same PC. For example, it's relatively easy to write code that opens a database connection and keeps it open for the duration of the work session of the human user. While this might be reasonable on a single PC, it doesn't work well over the Internet because the server probably has (desperately hopes it has) many concurrent users and the server will buckle under the load of keeping open many simultaneous connections, even if most of them aren't doing anything. We want to be able to access data in a way that can deal with slow and varying response times and doesn't tie up server resources for long periods.

We need our data access strategy to work well with XML.

XML (eXtensible Markup Language) is quickly emerging as the lingua franca of the Internet. That Latin term is about 200 years old, and it literally means "French language," but figuratively it means "the language everyone speaks." Today, we'd probably call XML the English of the Internet. I like to call XML the tofu of the Internet because it doesn't have any flavor of its own—it takes on the flavor of whatever you cook it with—or occasionally the WD-40 of the Internet, because it drastically lowers the friction of crossing boundaries. XML makes an excellent wire format for transporting data from one computer system to another because it's widely supported and free of implementation dependencies. Our data access strategy needs to go into and out of XML easily.

We need our new data access strategy to keep working with what's been working.

Finally, we need to maintain backward compatibility with existing code and data. The installed base of data access code is enormous, written and tested at great expense, and we can't afford to jettison it. Any new architecture that doesn't provide a bridge from the current state of affairs, whatever it is, doesn't have much chance in the market, no matter how cool it is on its own.

Solution Architecture

OLE DB provided a single programmatic interface for all providers of data.

Microsoft's first attempt at solving the problem of universal data access from a single programming model was OLE DB, released around 1995. That's so long ago in geek years that COM was still called OLE (a MINFU; see Chapter 4), and the author of the *Microsoft Systems Journal* article about the technology provided a CompuServe number (70313,1455) as a contact address. In OLE DB, every data provider implemented a standard set of interfaces for

allowing external access that required no knowledge by clients of the data provider's internal implementation. This process is illustrated in Figure 6-1.

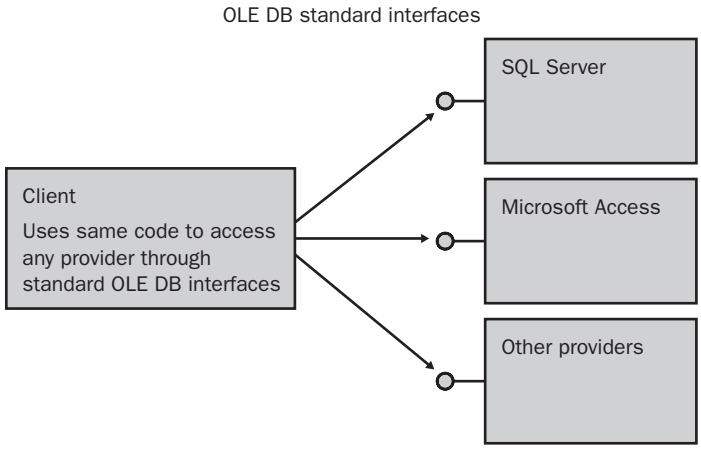


Figure 6-1 OLE DB abstracts away the differences between different data providers.

ADO made OLE DB easy for programmers to use but worked well only in a Microsoft-only environment.

It was a good idea and a good first try, but OLE DB[1] was hard for clients to program against, particularly in Microsoft Visual Basic. Microsoft next released ActiveX Data Objects (this was during the brief period when all things COM were called ActiveX, another MINFU), which Visual Basic programmers seized with cries of delight because it was so much easier (well, relatively) for them to program. ADO's front end provided an easier interface for clients to program against, and its back end spoke OLE DB to the provider, as shown in Figure 6-2. ADO worked fairly well on a Microsoft-only intranet and on the middle tier of a three-tier system accessed by Web clients. But ADO doesn't scale well to the open Internet. It uses DCOM to cross

1. You can see that as long ago as seven years, Microsoft foresaw the exhaustion of even the 456,976 unique FLAPs (Four-Letter Acronym Packages, FLAP itself is a FLAP, see Chapter 4) and started experimenting with alternatives. They've dusted off the FLEAP (Five Letter Extended Acronym Package, FLEAP itself is a FLEAP), which has a long and honorable history of military uses. Dwight David Eisenhower, for example, commanded SHAEF, the Supreme Headquarters Allied Expeditionary Force in the Second World War. While the use of FLEAPs hasn't crossed into the civilian sector, probably because there hasn't been any real need for them, 11,881,376 unique FLEAPs exist, so that ought to hold us for a while. If users balk at another increase in word length, the alternative would be to combine FLAPs, for example, CLOS, the Common LISP Object System. I call these FIAFs, which stands for FLAP Inside Another FLAP. And naturally, FIAF itself is a FIAF. When we exhaust the supply of FLEAPs, we can go to what I call the SLEAPE (pronounced "sleepy"), which stands for "Six Letter Extended Acronym Package, Eh?" (I coined this one just after a gig in Canada.) There are roughly 309 million SLEAPEs in a 26 letter alphabet, and the only one I've ever seen used is PCMCIA (People Can't Memorize Computer Industry Acronyms).

machine boundaries, which means that it can't easily work with non-Microsoft systems and has trouble getting through firewalls at Microsoft-only systems. ADO supported a limited amount of disconnected operation, but it was designed for and worked best in the connected case. It was a good solution for the problem it aimed at, but programmers' needs have changed in the modern Internet world.

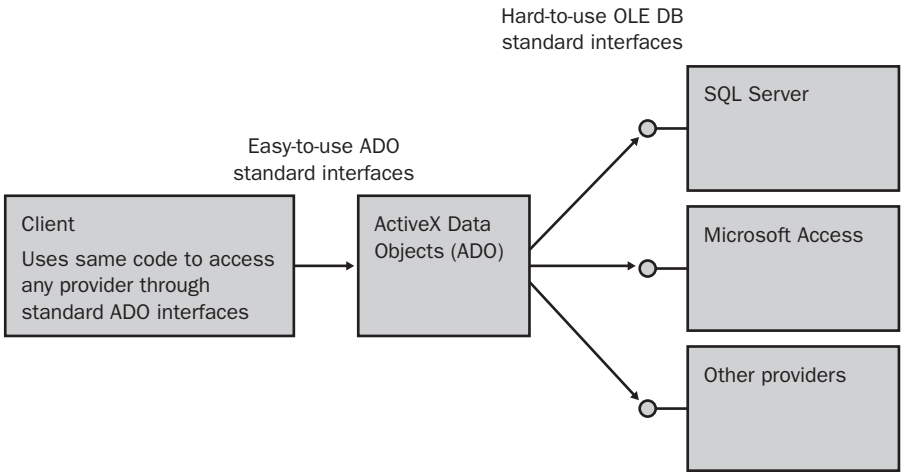


Figure 6-2 ADO object using OLE DB.

.NET provides data access via ADO.NET.

Microsoft .NET introduces ADO.NET, Microsoft's architecture for transferring the ideas of universal data access and easy programming from the COM-based world of ADO into the .NET world. ADO.NET is conceptually similar to ADO in the sense that it is a data abstraction layer that smooths over differences between data providers and includes prefabricated objects and functions for easy access to data.

The .NET common language runtime provides objects that access SQL Server and also any OLE DB provider.

A data provider that wants to make its data available to .NET clients via ADO.NET implements a standard set of .NET objects that connect the data provider to interested clients. These objects are *Connection*, *DataAdapter*, *Command*, and *DataReader*. A full description of these objects and their functionality requires examples, so I discuss them in the next section of this chapter. The developer of a data source can write his own implementation of these objects that is optimized for that particular data source, in the same manner as each data source today provides its own implementation of the OLE DB objects. Microsoft has provided an implementation of these objects for SQL Server, and these classes are part of the .NET common language runtime. In addition, the common language runtime provides an implementation of these objects that works with any OLE DB provider, so any current data provider that speaks OLE DB speaks ADO.NET automatically as well, as

shown in Figure 6-3. Version 1.1 of the .NET Framework added an implementation of these objects customized for Oracle databases. Other data providers can choose whether to write their own managed implementation of these objects or whether to write an OLE DB interface and use the compatibility layer.

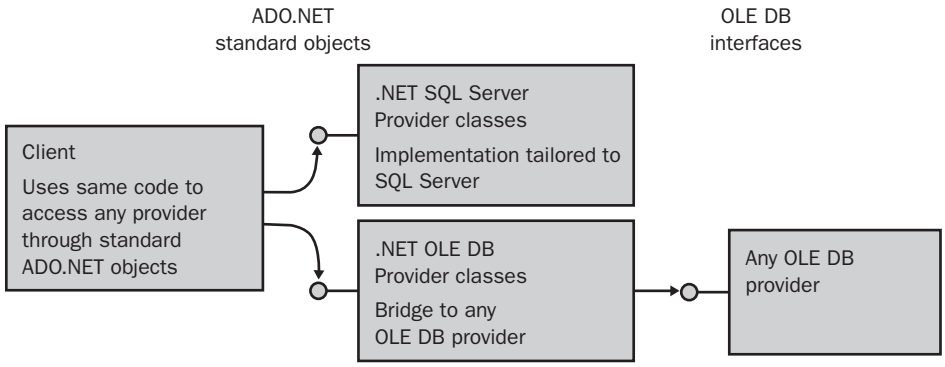


Figure 6-3 ADO.NET architecture and objects.

The server provides an ADO.NET *DataSet* object containing the result of a query.

ADO.NET provides its actual data in the form of a *DataSet* object, as shown in Figure 6-4. *DataSet* is a .NET class that represents a collection of data that results from one or more queries. It contains internal tables and provides methods that allow access to the tables' rows and columns. It also contains a schema describing its internal structure. A *DataSet* object can be untyped (the default), in which case a client asks for items by specifying their names in the form of coded strings. Alternatively, you can create a typed *DataSet* object that contains member variables tied to each individual field, which is easier to write good code for. Either type of *DataSet* object is compatible with .NET's XML serialization capability described in Chapter 7. This means that it knows how to convert itself into and out of XML so that it can be transmitted across process or machine boundaries.

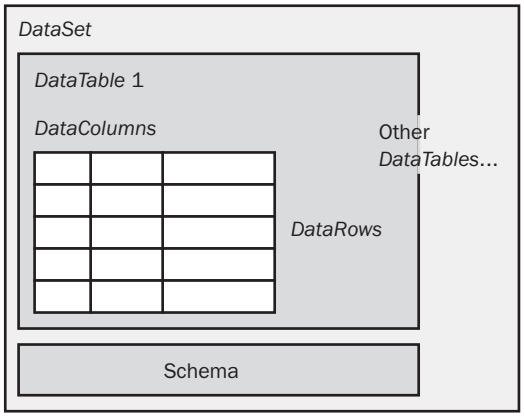


Figure 6-4 The ADO.NET *DataSet* object.

Windows Forms and Web Forms contain controls for displaying *DataSet* objects.

One of the main operations that programmers might want to do with a *DataSet* object when they get one is to display its contents to the user. Both Windows Forms (see Chapter 5) and Web Forms (see Chapter 3) contain controls that know how to take a *DataSet* object and render its contents for display to humans.

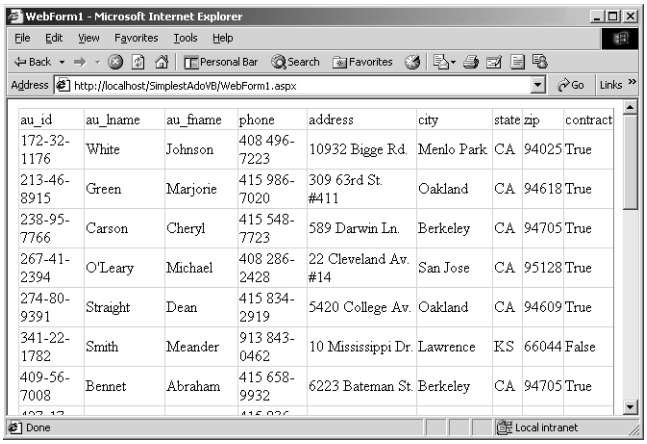
If they need it for backward compatibility, .NET programs can still use original ADO via the COM compatibility feature discussed in Chapter 2.

This chapter provides the briefest glimpse into the features in ADO.NET. Like Windows Forms, in fact, and like every chapter of this book, ADO.NET needs a book of its own. Microsoft Press has published *Microsoft ADO.NET (Core Reference)*, by David Sceppa, in May 2002 and *Building Web Solutions with ASP.NET and ADO.NET*, by Dino Esposito (February 2002).

Simplest Example

An ADO.NET example starts here.

As always, I started my exploration of ADO.NET with the simplest example I could think of that demonstrated anything useful. You can find this sample program on this book's Web site, *http://www.introducingmicrosoft.net*. The application is an ASP.NET page that performs a canned database query when you request the page. It uses *DataConnection* and *DataAdapter* objects to request all the entries in the *Authors* table in the *pubs* database in the Duwamish Books sample program distributed with the .NET Framework SDK. The query produces an ADO.NET *DataSet* object, which I display to the user in a Web Forms DataGrid control on the Web page. The page itself is shown in Figure 6-5 and the sample code in Listing 6-1.



au_id	au_lname	au_fname	phone	address	city	state	zip	contract
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

Figure 6-5 Web page from the simplest ADO.NET sample.

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

    ' Create Connection object containing connection string

    Dim Connection As New _
        SqlConnection("server=(local);uid=sa;pwd='';database=pubs")

    ' Create DataAdapter object containing query string

    Dim Adapter As New _
        SqlDataAdapter("select * from Authors", Connection)

    ' Create new empty DataSet object

    Dim DS = New DataSet()

    ' Fill DataSet object with results of query

    Adapter.Fill(DS, "Authors")

    ' Place data set into DataGrid control for user to look at

    DataGrid1().DataSource = DS.Tables("Authors").DefaultView

    ' Tell DataGrid control to display its contents
```

Listing 6-1 The *Page_Load* event handler of the simplest ADO.NET sample.

```
    DataGrid1().DataBind()

    ' Clean up database connection

    Connection.Close()

End Sub
```

When the user requests the page in her browser, the request comes to Internet Information Services (IIS) and ASP.NET, which fires the *Page_Load* event on the page as part of the rendering process. All the interesting code in this example lives in the handler for this event.

You first create an ADO.NET *Connection* object representing the connection to your database.

The first thing we have to do is create the *Connection* object. This object represents the opening in the database program through which requests flow in and data flows out, roughly analogous to the Ethernet jack on your office wall. ADO.NET provides two different common language runtime classes that we can use for our database connection. The class *System.Data.SqlClient.SqlConnection*, which I use in this example, is optimized to work only with Microsoft SQL Server. ADO.NET also provides the class *System.Data.OleDb.OleDbConnection*, which is a generic *Connection* object that works with any OLE DB data provider, including SQL Server. Except for the names of the object classes and some slight differences in the connection string, the generic *Connection* object works the same from a client perspective as the dedicated SQL Server interface. Obviously, writing two different sets of data access objects was more work for Microsoft, but Microsoft probably figured that doing so was worth the effort to make SQL Server work better than generic databases, and they were probably right. I use the SQL-specific classes in this book.

In the constructor of the *SqlConnection* object, we pass it the connection string that we use to connect to the database, containing such items as the data provider's name and location, the database inside the provider to use, and the user ID and password that we use to connect to it. The values in this string are the same as they were in standard, pre-.NET ADO.

You next create a *DataAdapter* object, which uses the *Connection* object to make calls into the database.

Having created the *Connection* object, we now need to create the *DataAdapter* object, which mediates between the *Connection* object and the client application. Think of the *DataAdapter* object as the Ethernet card in your PC. Programs talk to the network card (the *DataAdapter* object), which in turn talks to the jack on the wall (the *Connection* object). Your client program issues commands to the *DataAdapter* object, which transmits them to the database through the *Connection* object and then accepts the results from the *Connection* object and returns them to your client program. In the *DataAdapter* class constructor, we pass it the command that we want it to execute in the database—in this case selecting all the entries from a table of authors—

and the *Connection* object for it to use in making that query. ADO.NET provides two *DataAdapter* classes, which are *System.Data.OleDb.OleDbDataAdapter* and *System.Data.SqlClient.SqlDataAdapter*. As was the case with *Connection* objects, the former is a generic class that works with any OLE DB–compliant data source, and the latter is optimized to work with SQL Server.

You create an empty *DataSet* object and use the *DataAdapter* object to fill it with data.

Now that we have our *DataAdapter* object, we want to use it to query the database and fetch some data for us to make money with. ADO.NET provides the class *System.Data.DataSet* as the fundamental holder for all types of data. A *DataSet* object contains its own internal tables that will contain the results of the queries that we will make on the data provider through the *DataAdapter* object and the *Connection* object. We start by creating an object of this class, which is empty when we first create it. We fill the *DataSet* object with data by calling the *DataAdapter* object's *Fill* method, passing the *DataSet* object itself and the name of the table inside the *DataSet* object that we want the data to live in. If, as in this case, the table doesn't currently exist in the *DataSet* object, it will be created as a result of this call. The table name need not match the table in the underlying database, as the name is used only within the *DataSet* object by client programs.

Once I have the data set, I use a DataGrid control to easily display it to the user.

Once I have my *DataSet* object filled, I want to display its contents to the user. I do this by placing it into a DataGrid control, a Web Forms control developed expressly for this purpose. The control lives on an .aspx page, and it knows how to eat a *DataSet* object and render its contents into HTML for display to the user. I tell it which *DataSet* object to eat by setting its *DataSource* property to the *DataSet* object I just got from my query. I then tell the control "make it so" by calling its *DataBind* method.

It's a good idea to explicitly close the database connection.

Finally, when I am finished with the database connection, it's a good idea to explicitly close it by using the *Connection* object's *Close* method. If I simply let the *Connection* object go out of scope, the object wouldn't be finalized and the underlying database connection that it wraps wouldn't be freed until the next garbage collection, whenever that is. Database connections are scarce resources, and I'd like to recover them as soon as possible. Therefore I call the *Close* method to tell the *Connection* object that I am finished with the database connection so that it should reclaim those resources. Enforcing this determinism in a garbage-collected memory management environment is obviously slightly harder to code, and therefore slightly easier to mess up, than was the automatic reference counting scheme used in Visual Basic 6.0, which would have released the *Connection* object immediately. However, garbage collection makes it impossible for you to permanently leak away resources, which reference counting allowed in certain cases. See my discussion of garbage collection in Chapter 2 for more details about the benefits of foolproofness vs. easy determinism.

This example required very little code.

This simple example required very little code, but it illustrates important concepts of ADO.NET and also demonstrates that it doesn't take a lot of programming to get a lot of stuff done.

Tips from the Trenches

One of the best ways to improve the performance of database applications is by pooling database connections. ADO.NET automatically provides this service by default. The first time a client creates a *Connection* object, it really is created. When the client calls *Close* or *Dispose* on this object, rather than shredding it, the pooling manager puts it into a pool that it maintains until the client application's process is terminated. Subsequent creations of a connection with the same connection string parameters cause the object to be fetched from the pool; a new object is not created. You can modify the behavior of the connection pool—for example, its maximum and minimum number of connections—by making entries in the connection string. Generally, however, the default behaviors (pooling enabled, minimum of zero objects, maximum of 100) give you good performance with no development effort. Still, for proper operation of the pool, you must remember to call *Close* or *Dispose* on your *Connection* object, as this example does. This would be a very good application of a *try-finally* block, as described in Chapter 2.

More Complex Example: Disconnected Operation

The previous example is very simple, therefore it only scratches the surface. It doesn't show your own code reading data from a *DataSet* object; it doesn't show marshaling data across machine boundaries with XML; and it doesn't show disconnected operations, such as making changes to data and posting them back. So I've written a different sample, whose operation is shown in Figure 6-6, to demonstrate these features. Instead of using a browser to display data, I wrote a rich client using Windows Forms (see Chapter 5). The client uses an XML Web service (see Chapter 4) to fetch a data set from the server machine. The client allows a user to edit the results of the query and post the changes back to the underlying database through the XML Web service.

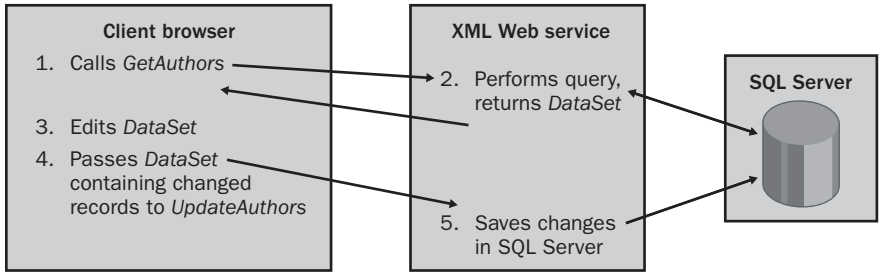


Figure 6-6 Operation of the DataSet sample program.

A sample showing disconnected operations starts here.

On the server, I've written a simple XML Web service that exposes the methods *GetAuthors* and *UpdateAuthors*. The first method's code is shown in Listing 6-2, and it's really quite simple.

```
<WebMethod()> Public Function GetAuthors() As  Data.DataSet

        ' Create Connection object

        Dim Connection As SqlConnection
        Connection = New _
            SqlConnection("server=(local);uid=sa;pwd='';database=pubs")

        ' Create DataAdapter object

        Dim Adapter As SqlDataAdapter
        Adapter = New _
            SqlDataAdapter("select * from Authors", Connection)

        ' Create empty DataSet object
        Dim DS As Data.DataSet
        DS = New Data.DataSet()

        ' Fill DataSet object with data

        Adapter.Fill(DS, "Authors")

        ' Return DataSet object to caller

        Return DS

    End Function
```

Listing 6-2 The *GetAuthors* method.

When the client calls *GetAuthors*, the method creates a *Connection* object and a *DataAdapter* object and uses these to create a *DataSet* object, as shown in the previous example. I could easily have added additional parameters for the client to pass to the XML Web service that the service could use in performing the query—say, authors whose first name is "John"—but I didn't want to complicate the example. The difference between this example and the previous one is that, instead of displaying the data set on a Web page for a human user, the XML Web service returns the *DataSet* object to the client program that calls it. This causes the *DataSet* object to be serialized into XML and transmitted over the wire to the client. You can see the *DataSet* object layout in XML by using the XML Web service's built-in test capability, as shown in Figure 6-7.

The XML Web service method simply returns a *DataSet* object, which causes it to be transmitted in XML.

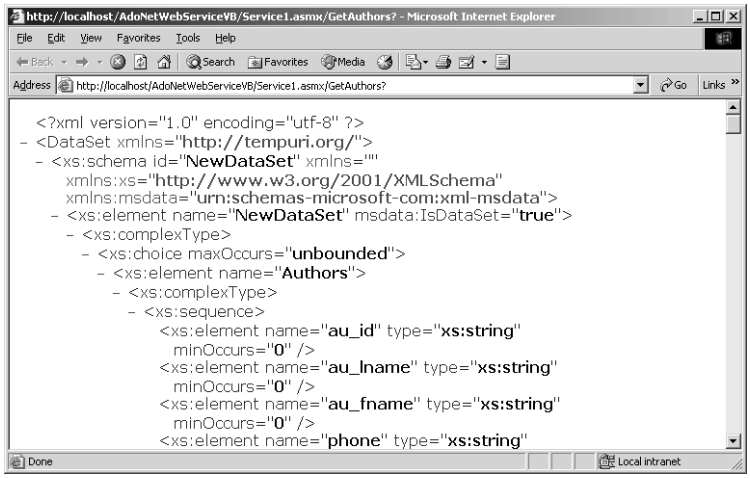


Figure 6-7 The XML layout for the *DataSet* object.

You can run the XML Web service through the sample client app. When the user clicks the Fill button, the app fetches the data set containing all the authors from the XML Web service, as shown in Figure 6-8. The code is shown in Listing 6-3.

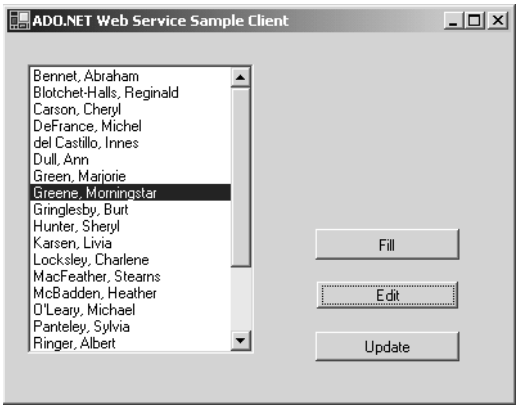


Figure 6-8 Our sample client application.

```
Dim MyDataSet As DataSet

Private Sub Button1_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) _
                         Handles Button1.Click

    ListBox1.Items.Clear()

    ' Create proxy object for accessing  Web service

    Dim Server As New localhost.Service1()

    ' Fetch DataSet object from proxy

    MyDataSet = Server.GetAuthors

    ' Populate initial list box with author's names

    Dim ThisAuthorRow As DataRow

    For Each ThisAuthorRow In MyDataSet.Tables("Authors").Rows

        ' Create my object that holds the author's name and
        ' the author's data row

        Dim ThisGuy As New MyOwnListItem(ThisAuthorRow("au_lname") + _
            ", " + ThisAuthorRow("au_fname"), ThisAuthorRow)
        ListBox1.Items.Add(ThisGuy)
    Next
End Sub
```

Listing 6-3 Code from the application.

The client creates an object of the XML Web service class and calls the *GetAuthors* method, which returns the *DataSet* object as I've just described. When the client assigns this return value to a variable, it takes the XML stream sent by the server and rehydrates it into a functioning *DataSet* object. The *DataSet* object has been transmitted using XML and HTTP, which clearly illustrates the fact that it can be sent to any type of client, even a non-Microsoft system.

The client automatically rehydrates the XML into a *DataSet* object.

Once I've gotten the *DataSet* object from server to client, I want to access it on the client side. I fetch the table in the *DataSet* object that I know contains the records of authors by using the *DataSet* object's *Tables* collection, passing the name of the table that I want to access, in this case *Authors*. This call returns an object of class *System.Data.DataTable*. This table contains the author records that I want, each represented by an object called *System.Data.DataRow*. I step through each record sequentially by accessing the collection called *Rows* in the *DataTable* object.

The *DataSet* object contains .NET properties representing tables and rows.

I'd like to get each author's first name and last name, assemble them into a string, and display the string in the *ListBox* control. Getting the data from the *DataSet* object is easy. Each *DataRow* object contains a collection of columns that represent the fields in the database that actually contain individual values. I access a column by using its name, in this case *au_lname* and *au_fname*, as shown at the end of the code listing in Listing 6-3.

You access an individual column through its name in a *DataRow* object.

Since I want to enable the user to edit the *DataRow* later, I need to associate a *DataRow* with its line in the *ListBox* control. In Visual Basic 6.0, I'd have used the *ListBox* control's *ItemData* property to hold an integer key identifying the row in a separate collection I'd have to somehow manage. But I can't do that in .NET because the *ItemData* property has been removed. Instead, the *ListBox* control can hold a .NET object of any class, but it won't hold two separate items (the string and the key) as it did before. The *ListBox* control displays the string returned by the object's *ToString* method (described in Chapter 2). So to make this app work the way I wanted, I needed to roll my own class that contained all the information I wanted for each line of the *ListBox* control to hold. You'll find that information in the class *MyOwnListItem*. It holds a *DataRow* object and a string, both of which it accepts in its constructor. The code is shown in Listing 6-4. It sounds complicated, but it really isn't. It saves me having to manage my own collection of *ListBox* items, which is a net gain even if you use it in only one place. For each row, I create an object of this class, passing it the full name string I want to display in the *ListBox* control and the corresponding *DataRow* object.

ListBox controls now require a .NET object because the *ItemData* property has been removed.

```
Public Class MyOwnListItem

    Public m_FullName As String
    Public m_DataRow As Data.DataRow

    ' Class constructor that accepts a name for display and a DataRow
    ' to hold

    Public Sub New(ByVal FullName As String, _
                   ByVal MyDataRow As Data.DataRow)
        m_FullName = FullName
        m_DataRow = MyDataRow
    End Sub

    ' Override System.Object.ToString. The displaying ListBox control
    ' will call this method to get the string to display

    Public Overrides Function ToString() As String
        Return m_FullName
    End Function

End Class
```

Listing 6-4 Code from my own class *MyOwnListItem*.

Now I want to edit an individual entry. When the user selects an entry from the *ListBox* and clicks Edit, I pop up a dialog box showing the status of that author's contract, as shown in Figure 6-9. You can see the code in Listing 6-5.

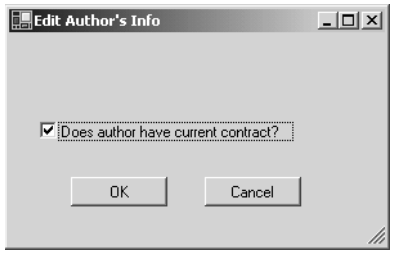


Figure 6-9 The Edit Author's Info dialog box.

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button2.Click

    ' Get selected author's data row from ListBox control

    Dim SelectedListItem As MyOwnListItem
    SelectedListItem = ListBox1.Items(ListBox1.SelectedIndex)

    Dim AuthorsDataRow As Data.DataRow
    AuthorsDataRow = SelectedListItem.m_DataRow

    ' Get state of author's contract from data row

    Dim contract As Boolean
    contract = AuthorsDataRow("contract")

    ' Set control in editing form according to current state
    ' of author's contract

    Dim EditForm As New Form2()
    EditForm.CheckBox1.Checked = contract

    ' Set editing form's text and show to user.
    ' If user clicked OK, then change value in data row
    ' and enable Update button

    If (EditForm.ShowDialog() = DialogResult.OK) Then
        AuthorsDataRow("contract") = EditForm.CheckBox1.Checked
        Button3.Enabled = True
    End If

End Sub
```

Listing 6-5 Code allowing editing of author info.

I first fetch the *DataRow* object representing the user's selection from the *ListBox*. (See how much easier it is than a separate collection?) I look at the *DataRow* object's *contract* column and set the dialog box's *CheckBox* control to the column's value. If the user clicks OK, I fetch the state of the *CheckBox* control from the dialog box and set the value in the *DataRow* object's *contract* column. You can see that I'm simply treating the *contract* column of a row like a standard variable.

You read and write columns in the *DataRow* object as if they were simple variables.

When the user clicks Update, I need to send whatever changes he's made back to the server to update the server's master database tables. I could send the entire data set back to the server and let the server figure out which rows have changed, but this would be a waste of network bandwidth. It

You can easily select only the changed rows to be sent back to the server for updating.

would be better to send only the changed rows. I can easily do this by using the method *DataSet.GetChanges*, which returns another *DataSet* object containing only the rows in the original data set to which changes have been made. I send this *DataSet* object back to the XML Web service using the service's *UpdateAuthors* method. You can see the code for *UpdateAuthors* in Listing 6-6.

```
<WebMethod()> Public Function UpdateAuthors( _
    ByVal ChangedItemsDS As System.Data.DataSet) As Integer

    ' Create new Connection object

    Dim Connection As SqlConnection
    Connection = New _
        SqlConnection("server=(local);uid=sa;pwd='';database=pubs")

    ' Create DataAdapter object

    Dim Adapter As SqlDataAdapter
    Adapter = New SqlDataAdapter()

    ' Create and set properties of Command object

    Dim MyUpdateCommand As New _
        Data.SqlClient.SqlCommand( _
        "UPDATE Authors SET contract = @contract WHERE au_id = @au_id", _
        Connection)
    Adapter.UpdateCommand = MyUpdateCommand
    Adapter.UpdateCommand.Parameters.Add("@contract", SqlDbType.Bit, _
                                         1, "contract")
    Adapter.UpdateCommand.Parameters.Add("@au_id", SqlDbType.VarChar, _
                                         11, "au_id")

    ' Send update command to database via DataAdapter object,
    ' specifying the changed records DataSet object.
    ' The Update method returns an integer, which we return to
    ' the client

    Return Adapter.Update(ChangedItemsDS, "Authors")

End Function
```

Listing 6-6 Updating author information.

The *DataAdapter* object uses a *Command* object to make changes to the underlying database.

On the server side, my XML Web service catches the incoming *DataSet* object containing changes that the client has made. It creates a *Connection* object and a *DataAdapter* object as before. In this case, we are going to be updating data that's already in the database, so we also need a *Command*

object, which represents a command that you use to tell the database to do something. The common language runtime provides two classes of *Command* object, *System.Data.OleDb.OleDbCommand*, which is the generic *Command* object available to any OLE DB provider, and *System.Data.SqlClient.SqlCommand*, which is the version specific to SQL Server. You create one of these objects as shown, passing in its constructor the SQL string that you want executed. You then plug the *Command* object into the *DataAdapter* object by assigning it to the *DataAdapter* object's *UpdateCommand* property. This assignment tells the *DataAdapter* object which SQL command to run when data is updated. You'll see that I also have to add parameter objects to tell the command which variables map to which columns. (The *DataAdapter* object also contains *InsertCommand* and *DeleteCommand* properties, which accept a similar *Command* object used during inserts and deletes, respectively, but I don't use these properties in this example.) Finally, I call the *DataAdapter* object's *Update* method, telling it to take the update command and run it against the database, using the *DataSet* object that I received from the client. This call returns the number of rows updated, which I return to the client.

Design of databases used by ADO.NET needs to take into account its loosely coupled nature.

The loosely coupled nature of ADO.NET *DataSet* objects requires careful thought in database design. Since you don't know how long a client is going to keep a *DataSet* object, you can't afford to keep locks on all your data to prevent conflicts; you'd tie the system into knots very quickly. Instead, you can design your database to use some form of optimistic concurrency. If I had done that, my sample XML Web service would contain code that would check before saving updates to see whether the data it is saving had been changed by someone else in the interim—for example, by checking a timestamp column. If the data had been changed, the unsaved, edited values might be bad, so the XML Web service would throw an error back to the client, and the client would somehow inform the user of this and make the user do it again. We call this type of concurrency *optimistic* because we're hoping that this somewhat painful process won't happen very often. This approach works well in systems that experience low contention rates. For higher contention systems, such as buying tickets online to the latest Harry Belafonte concert (he's still got his stuff, by the way, even at age 76), you might use compensating transactions—remove a specific pair of tickets from the theater database when the user first asks what's available and then perform the opposite operation to put the tickets back in the pool if the user doesn't buy them within ten minutes.

Tips from the Trenches

Most developers I know don't like optimistic concurrency, but it can be very efficient if you do it right. The key is to ruthlessly pare away the potential situations in which contention can arise, which this example doesn't even try to do. For example, a production app might again request the data record for an author when the user opens the editing dialog box to work on that author so that the user would be looking at the latest data. The app might automatically save the record back to the database when the user clicks OK so that the changes can be reflected immediately. These strategies greatly reduce the amount of time during which two users might be messing each other up. If your contention is low, it's the way to go.

Visual Studio Support and Typed *DataSet* Objects

Visual Studio provides good editor support for writing data applications.

In the two previous examples, I've written my own code for creating the various objects that I've needed for my data operations, such as the *Connection* and the *Adapter*. I've also had to write code for setting their properties, such as the connection string parameters in the *Connection* object and the query string parameters in the *Adapter* object. As any developer who's struggled with connection strings knows, this can become painful. I don't usually have to write code that creates buttons on a form or sets their properties; instead, Visual Studio provides an editor that generates that code for me and saves me lots of time. I'd really like some of that support for writing my database operation code, and Visual Studio gives it to me.

Working with standard *DataSet* objects could use some development time support.

Besides the *Connection* and *Adapter* objects, I'd also like some help with *DataSet*s. Working with a *DataSet* object as shown in the previous example is useful, but it's still somewhat unwieldy because you have to plug in strings to specify the names of tables and columns. Some programming cases require this flexibility, such as a generic data browsing tool that allows a human user to type in any sort of query that occurs to him. But the majority of data access programs perform the same operations on the same data sources over and over and over again—think of the concert ticket application, for example. In cases like these, it doesn't make sense to require the programmer to pass a string name to identify a table or column. The programmer

has to look up the name in a manual so that she knows which one to use and then make sure she types it in correctly every time—that she hasn't transposed a key and typed "Auhtors," for example. Mistakes like this are easy to make because raw *DataSet*s don't have development-time support to make sure that you type in the correct string. They are also difficult to debug because your eye isn't good at picking out close misspellings. It may not sound like a terrible problem, but if we could prevent it, we'd save some programmer time, some testing time, and probably some service calls. If you don't want those savings, send yours to me, OK?

We want a dedicated *DataSet* class tailored to the results of a specific data operation.

What we'd really like is a *DataSet* object that's tailored to the particular data that we expect to receive from a specific operation. It would have table and column names already wired into it as hard-coded variables. It would allow IntelliSense to show these names during programming so that we wouldn't have to reach for the paper manual. We wouldn't have to worry about misspellings because the compiler would catch us if we somehow ignored IntelliSense and got a name wrong. And we'd like good development tool support for generating them.

It turns out that all of our wishes have been granted by .NET and Visual Studio. Now that we've seen the nuts and bolts of ADO.NET, I'll show you the tools that make it easier. I've written a sample program that demonstrates it.

A database programming example demonstrating intelligent tool support starts here.

Visual Studio .NET supports developers writing data applications by providing its Server Explorer, shown in Figure 6-10. Server Explorer shows the various elements on a server for which it can generate .NET wrapper class objects, such as message queues and performance counters. The most interesting part of Server Explorer for our purposes is that it allows us to see the contents of our local SQL Server installation down to the table level. (It will actually go down to the individual column level, but I'm not using that for this example.)

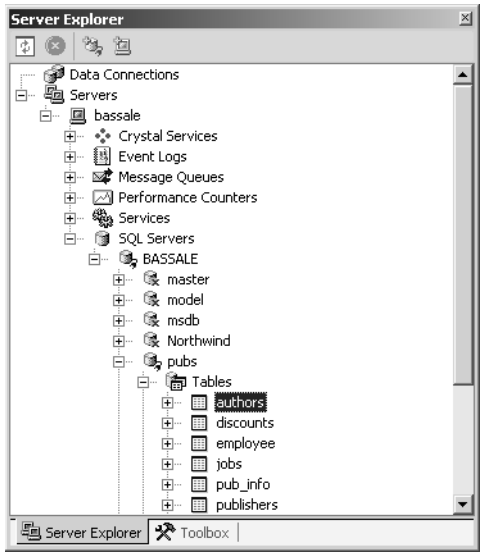


Figure 6-10 Server Explorer within the Visual Studio .NET environment.

Server Explorer in Visual Studio will automatically generate the correct *SqlConnection* and *SqlDataAdapter* objects for accessing a database.

When I click on a table (in this case, *authors*) and drag it onto my design surface, Visual Studio generates *SqlConnection* and *SqlDataAdapter* objects and sets their parameters to the proper values for accessing the selected table. For example, the *SqlDataAdapter* is set to use the created connection, and has appropriate SQL commands added, as shown in Figure 6-11. As with all objects generated by Visual Studio, the code for creating them is placed in the *InitializeComponent* method of the container, in this case my XML Web service. I haven't gained any run-time performance advantage by setting up my connection and adapter objects this way. In fact, I've probably lost a little because Visual Studio generates the Insert and Delete commands that this example doesn't use. But it saves a whole lot of developer time and prevents errors, which means it's usually a good trade-off. And I could manually remove the unneeded pieces if I really cared.

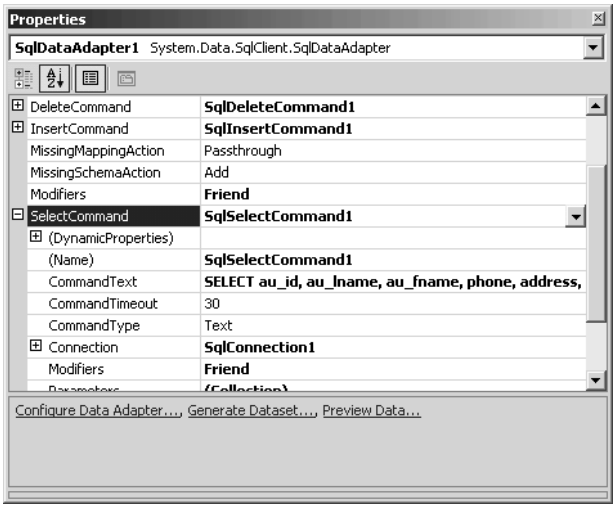


Figure 6-11 *SQLDataAdapter* object properties.

Tips from the Trenches

My clients report that the overhead of creating the unneeded pieces within these objects is not very high. It doesn't perform any external communication with the database, it's just allocating local memory and setting its values. Especially when used in conjunction with an XML Web Service as shown, there's so much other stuff happening that this overhead gets lost in the noise. However, if you find that it's taking more time than you can afford, you will probably still find it handy to use the designer to generate the code and then remove or modify the pieces that you don't care about in any particular method.

ADO.NET supports typed *DataSet* classes.

Now that I have my connection and adapter set up nicely, it's time to do something about my *DataSet*. ADO.NET provides the *typed DataSet* class. This is a custom class, derived from *System.Data.DataSet*, that provides named member variables for each specific table, row, and column. If we know at programming time which tables and columns a data set will contain, we can use utility programs to generate a typed *DataSet* class. This feature might not sound like a big deal, and I didn't think it would matter much until I tried it, but now I'm hooked. As long as you know during program development which data queries you are going to want to make, and you usually

will, you won't want to program any other way. I've rewritten the authors client example from the previous section to use a typed *DataSet* object. This approach was easy to generate and made my programming somewhat easier to accomplish and somewhat harder to get wrong. The cost is a small amount of extra code, which you don't have to write, in your application. Any time you can trade off larger code size for faster and better programming, you don't have an economic choice.

You generate a typed *DataSet* class using Visual Studio's wizards.

You generate a typed *DataSet* by selecting Generate Dataset from the Data menu of Visual Studio's main menu. (You can also generate it with the command-line utility XSD.exe, which requires an XML schema that describes your data set.) Visual Studio pops up the dialog box shown in Figure 6-12, asking for the name of the new class and the table that you want it to match. You make your selections and Visual Studio generates the code for the new class. In Class View, shown in Figure 6-13, you can see that the new class contains strongly typed classes to represent the table (*authorsDataTable*) and the row within the table (*authorsRow*).

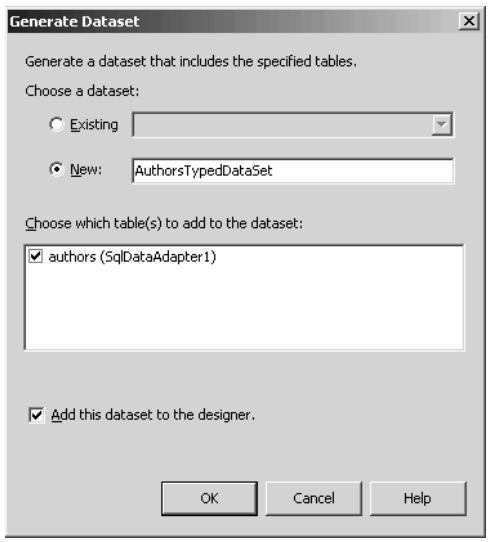


Figure 6-12 Generating a *DataSet* in Visual Studio .NET.

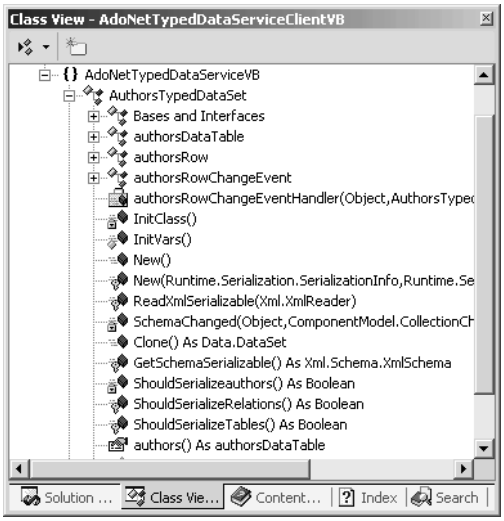


Figure 6-13 Class View showing strongly typed classes.

Writing the database access code is much easier with the objects that Visual Studio has created.

I rewrote my XML Web service to take advantage of the new objects that it contains. Listing 6-7 shows the code of my XML Web service method. It's much simpler because the connection, adapter, and typed data set have already been created. In my *GetAuthors* method, I simply tell the adapter to fill the data set and then return it. In my *UpdateAuthors* method, I simply tell the adapter to update the database with the new information. Again, I'm not saving any CPU cycles here; the objects are still being created exactly as if I had written the code myself, but I had to write much less code. When either method returns, ASP.NET automatically calls *Dispose* on the XML Web service object, which automatically disposes of all its components, including the connection and adapter.

```
<WebMethod()> Public Function GetAuthors() As AuthorsTypedDataSet

    ' Connection, adapter, and dataset objects have been
    ' added by designer instead of with our own code.

    ' Fill DataSet object with data

    Me.SqlDataAdapter1.Fill(Me.AuthorsTypedDataSet1)

    ' Return DataSet object to caller

    Return Me.AuthorsTypedDataSet1
```

Listing 6-7 XML Web service sample code for getting and updating author information.

```
End Function

<WebMethod()> Public Function UpdateAuthors(ByVal ChangedItemsDS _
    As System.Data.DataSet) As Integer

    ' Connection, adapter, and dataset objects have been
    ' added by designer instead of with our own code.

    ' Call Update method on adapter, which makes updates in records

    Return Me.SqlDataAdapter1.Update(ChangedItemsDS)

End Function
```

Writing the client code is much easier with the typed data set.

The client code, shown in Listing 6-8, is very similar to the previous example. You can see that instead of saying *MyDataSet.Tables("Authors").Rows*, I say *MyDataSet.authors*. To fetch an individual value, I say *ThisAuthorRow.au_lname* instead of *ThisAuthorRow("au_lname")*. These differences may not sound like much, but they remove a common source of errors (misspelling the string) and save programmer time by allowing IntelliSense support, as shown in Figure 6-14. If you still don't think it sounds useful, try using it for an hour and then give it up. You'll change your mind very quickly. The rest of the sample gets similarly easier. If you know your query set at development time, you do not have an economic choice.

```
Dim MyDataSet As localhost.AuthorsTypedDataSet

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles Button1.Click

    ListBox1.Items.Clear()

    ' Create proxy object for accessing Web Service

    Dim Server As New localhost.Service1()

    ' Fetch DataSet object from proxy

    MyDataSet = Server.GetAuthors

    ' Populate initial list box with author's names
```

Listing 6-8 Client code for typed XML Web service sample.

```
    Dim ThisAuthorRow As localhost.AuthorsTypedDataSet.authorsRow

    For Each ThisAuthorRow In MyDataSet.authors

    ' Create my object that holds the authors name and
    ' the author's data row

    Dim ThisGuy As New MyOwnListItem(ThisAuthorRow.au_lname + _
            ", " + ThisAuthorRow.au_fname, ThisAuthorRow)
        ListBox1.Items.Add(ThisGuy)
    Next
End Sub
```

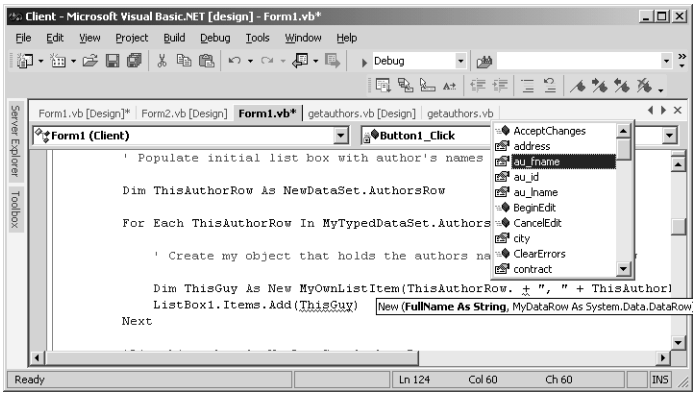


Figure 6-14 IntelliSense support in Visual Studio.