Apress™
Books for Professionals by Professionals™

Real-World ASP.NET: Building a Content Management System

by Stephen R.G. Fraser
ISBN # 1-59059-024-4

CHAPTER 7

Database Development and ADO.NET

An amazing amount of functionality is built into Visual Studio .NET when it comes to Microsoft SQL Server database development. If you combine it with ADO.NET, you don't ever have to leave Visual Studio .NET when it comes to your database efforts.

Visual Studio .NET does not leave you in the cold when it comes to database-development utilities. It provides you with a way to create a database, table, view, and stored procedure. When combined, these cover almost everything you need to build your database.

ADO.NET takes over where the utilities leave off and enables you to seamlessly join your databases to your Web page code. With little effort, you can connect, extract data, and then load the data into your Web pages.

You don't have to take my word for it. Take a look for yourself.

Visual Studio .NET's Database Utilities

Visual Studio .NET is well equipped when it comes to the design and development of Microsoft SQL Server databases. It provides the functionality to create databases, tables, views, stored procedures, and many other features. This book will cover only these four features, however, because they are all you need to create the content management system that this book is building.

The starting point of all database utilities is the Server Explorer. Select Server Explorer from the View menu to open it; it looks similar to Figure 7-1. You will find your database in one of two places. If the database is Microsoft SQL Server 7.0 or higher, it will be located in the SQL Servers database folder inside your Servers folder. If a supported OLE DB connects the database, it will be found in the Data Connections folder just above the Servers folder.

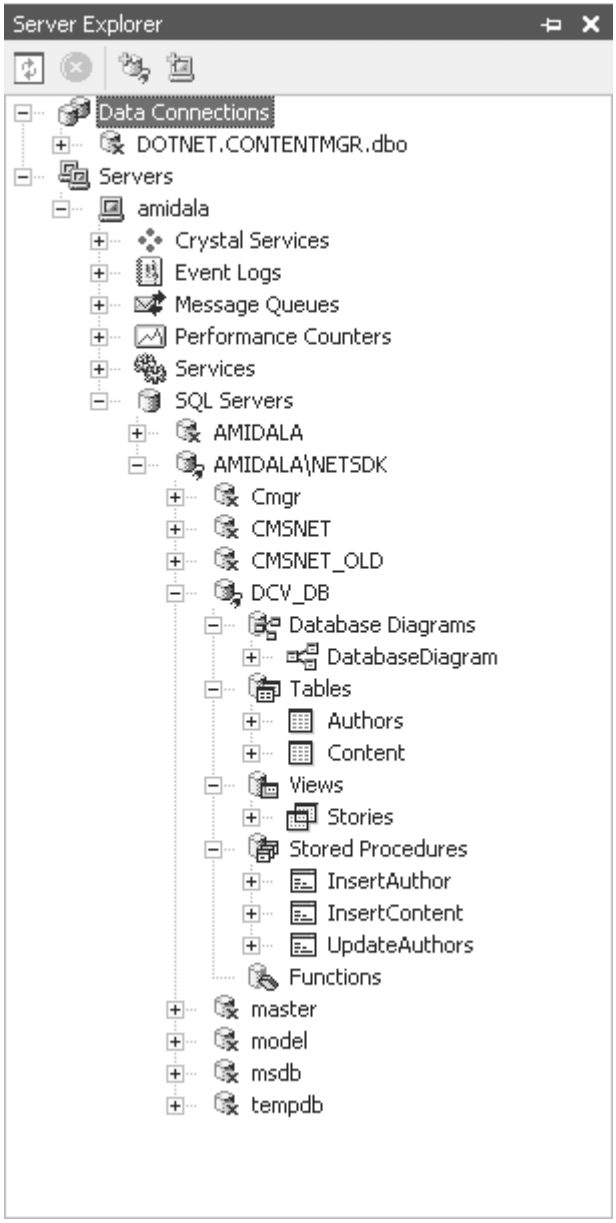


Figure 7-1. The Server Explorer

The functionality provided to OLE DB–connected databases is mostly restricted to viewing and editing records. On the other hand, Visual Studio .NET provides Microsoft SQL Server databases with much of the functionality that comes with SQL Enterprise Manager. This book focuses on Microsoft SQL Server and will cover the functionality provided by Visual Studio .NET. If you are developing using an OLE DB–connected database, much of the first part of this chapter will not help you because you will have to use the database maintenance tools provided by your database.

TIP *I recommend that you install the MSDE 2000 database server provided with the .NET Framework samples to get a feel for the functionality provided by Visual Studio .NET. You can always uninstall it later.*

There is nothing stopping you from building your Microsoft SQL Server databases outside of Visual Studio .NET and then adding the database to the Server Explorer. I will show you how to do this later when you walk through the process of adding the CMSNET database in Chapter 9. First, let's do it the hard way and build a simple content management database that you can integrate into the Dynamic Content Viewer you built in Chapter 6.

Creating a New Database

The first step in database development is not creating one. Obviously, creating the data model, designing the logic database, and designing the physical database should come first. But hey, I'm a programmer. I'll code first and then go ask questions. (I'm joking—really!)

Visual Studio .NET makes creating databases so easy that it's almost not worth explaining how to do it.

WARNING *Be sure you really want the database you are creating because there is no way to delete it once it's created in Visual Studio .NET. I had to go to the Microsoft SQL Enterprise Manager to delete my test databases. It is also possible to execute the* `DROP DATABASE` *command to remove the database.*

The following steps will create the database DCV_DB, which you will use throughout the chapter.

1. Select Server Explorer from the View menu.

2. Expand the SQL Servers database folder from within your Servers folder.

3. Right-click the database folder.

4. Select the New Database menu item, which will display the Create Database dialog box shown in Figure 7-2.

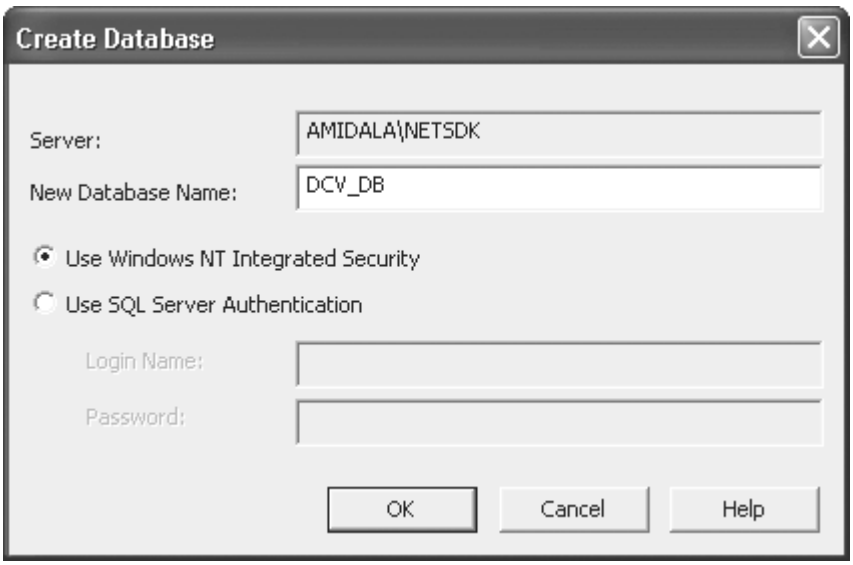


Figure 7-2. The Create Database dialog box

5. Enter **DCV_DB** in the New Database Name field.

6. Click OK.

Now you should have a new database called DCV_DB in your database folder. You can expand it and see all the default folders built. If you click these folders, however, you will see that there is nothing in them. That's your next job.

Adding and Loading Tables and Views to a Database

An empty database is really quite useless, so let's add a couple of tables to the database to provide a place to store your content.

NOTE *The tables and views you are using in this chapter are purposely very simple (you might even call them minimal) and are not the best schema around. I did this so that you don't get bogged down with the details of the database and so it doesn't take much effort or time for you to build them yourself.*

The first table is for storing authors and information about them, and the second table is for storing headlines and stories. The two databases are linked together by a common `AuthorID` key. Figure 7-3 presents a data diagram of the database.

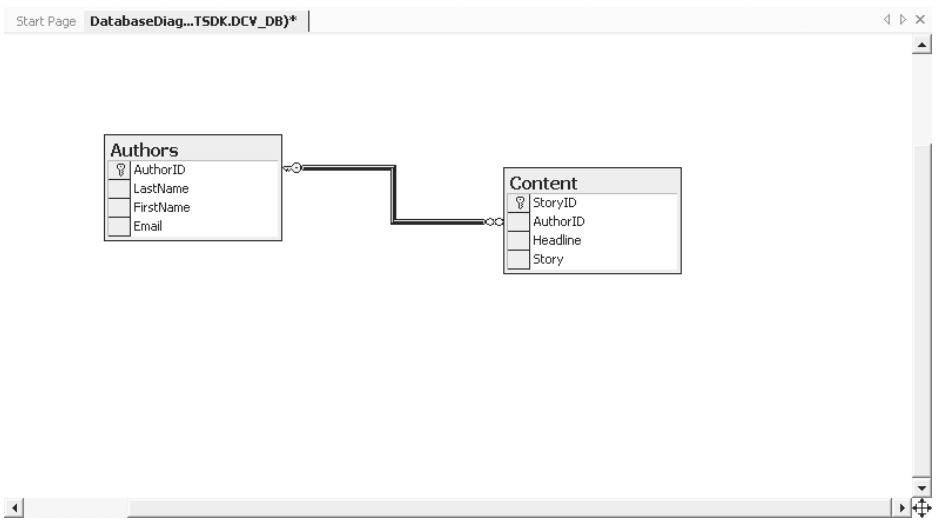


Figure 7-3. The DCV_DB data diagram

Having this separation means you only have to store one copy of the author information, even though the author may have written many stories. If you had created only one table to contain all the information, a lot of duplicated author information would have to be rekeyed each time a story is added to maintain the database. It also, conveniently, enables me to show you how to create a relationship between tables.

The process of building a new table is only slightly more difficult than creating a database. The hard part is figuring out what columns are needed and the format for each table in the database. It's nice to know you can spend most of your time designing the ultimate database schema instead of figuring out how to implement it.

Creating Tables

To create the first table, follow these steps:

1. Navigate down to the database folder as you did in the previous section.
2. Expand the database folder.
3. Expand the DCV_DB folder.
4. Right-click the Tables folder.
5. Select the New Table menu item. You should now have an entry form in which to enter the database columns found in Table 7-1.

Table 7-1. Authors Database Column Descriptions

COLUMN NAME	DATA TYPE	LENGTH	DESCRIPTION	IDENTITY	KEY
AuthorID	int	4	Autogenerated ID number for author	Yes	Yes
LastName	char	32	Last name of the author	No	No
FirstName	char	32	First name of the author	No	No
Email	char	64	E-mail address of the author	No	No

6. Select Save Table1 from the File menu.
7. Enter **Authors** into the edit field in the dialog box.
8. Click OK.

Go ahead and repeat this for the second table, but use the information in Table 7-2 and save the table as **Content**.

Table 7-2. Content Database Column Descriptions

COLUMN NAME	DATA TYPE	LENGTH	DESCRIPTION	IDENTITY	KEY
StoryID	int	4	Autogenerated ID number for author	Yes	Yes
AuthorID	int	4	Foreign key to author database	No	No
Headline	char	64	Headline for the content	No	No
Story	text	16	Story portion of the content	No	No

This book will not go into what all the data types mean, but if you are interested, many good books on Microsoft SQL Server and SQL cover this topic in great detail.

The Identity field, when set to Yes, will turn on autonumber generation for the column. Why you call the field Identity (instead of Autonumber) is a mystery to me. I'm an application programmer, though, and not a database person. It's probably some special database term.

Okay, you now have your tables. The next step is to build a relationship between them. In this database, it is fairly obvious—`AuthorID` is the column that should link these two tables.

Creating a Relationship

To create a relationship between your tables, follow these steps:

1. Right-click the Content table in the Server Explorer.
2. Select Design Table from the menu items.
3. Right-click anywhere on the Table Designer.
4. Select Relationships from the menu items. This will bring up a Relationships property page similar to Figure 7-4.

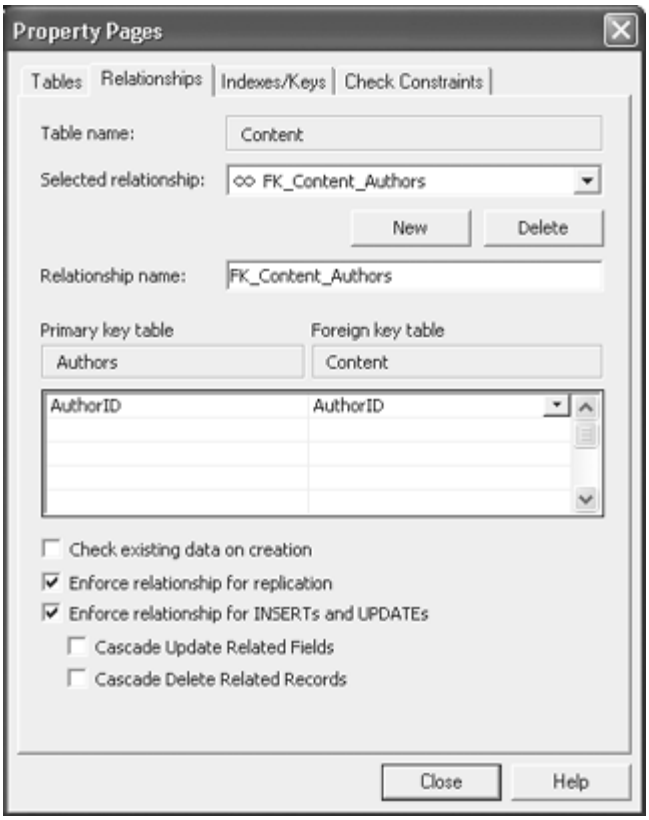


Figure 7-4. The Relationships property page

5. Click the New button.

6. Select Authors as the primary key side of the relationship from the Primary Key Table drop-down list.

7. Select AuthorID as the primary key in the grid beneath the Primary Key Table drop-down list.

8. Select Content as the foreign key side of the relationship from the Foreign Key Table drop-down list.

9. Select AuthorID as the foreign key in the grid beneath the Foreign Key Table drop-down list.

10. Click Close.

Now you have two tables and a relationship between them. Quite often, when you want to get data from a database, you need information from multiple tables. For example, in this case, you want to get all stories with each author's first and last name. As mentioned previously, you could have created the `Content` table that way, but then you would have a lot of duplicate data floating around. There is nothing stopping you from executing a SQL statement that gets this information, as shown in Listing 7-1.

Listing 7-1. Getting Data from Two Tables

```
SELECT      FirstName,
            LastName,
            Headline,
            Story
FROM        Authors,
            Content
WHERE       Authors.AuthorID = Content.AuthorID
ORDER BY    StoryID ASC
```

Personally, I prefer to be able to write something like this instead:

```
SELECT * FROM Stories
```

This is exactly what you can do with database views. Basically, you might think of a view as a virtual table without any data of its own, based on a predefined query. If you know you are going to use the same set of data based on a query, you might consider using the view instead of coding.

NOTE *Those of you who are knowledgeable about SQL and views might have noticed the* `ORDER BY` *clause. Microsoft SQL Server supports the* `ORDER BY` *clause in its views, unlike some other database systems.*

Creating a View

Follow these steps to create a view:

1. Right-click the Views table from within the DCV_DB folder in the Server Explorer.

2. Select New View from the menu items. This will bring up an Add Table dialog box similar to Figure 7-5.

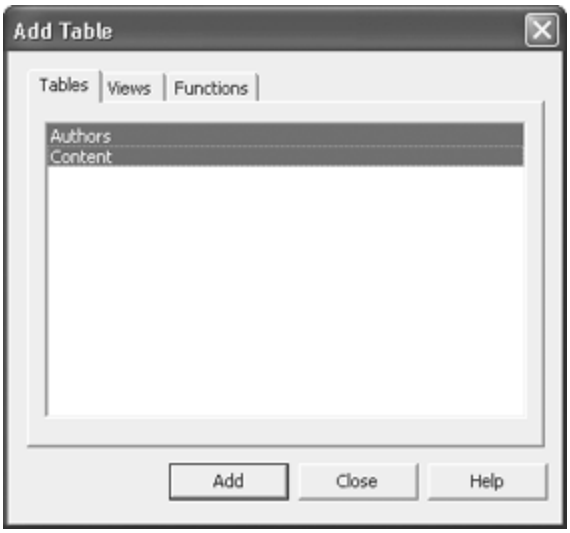


Figure 7-5. The Add Table dialog box

3. Select both Authors and Content and click the Add button.

4. Click Close. This should generate a window similar to Figure 7-6.

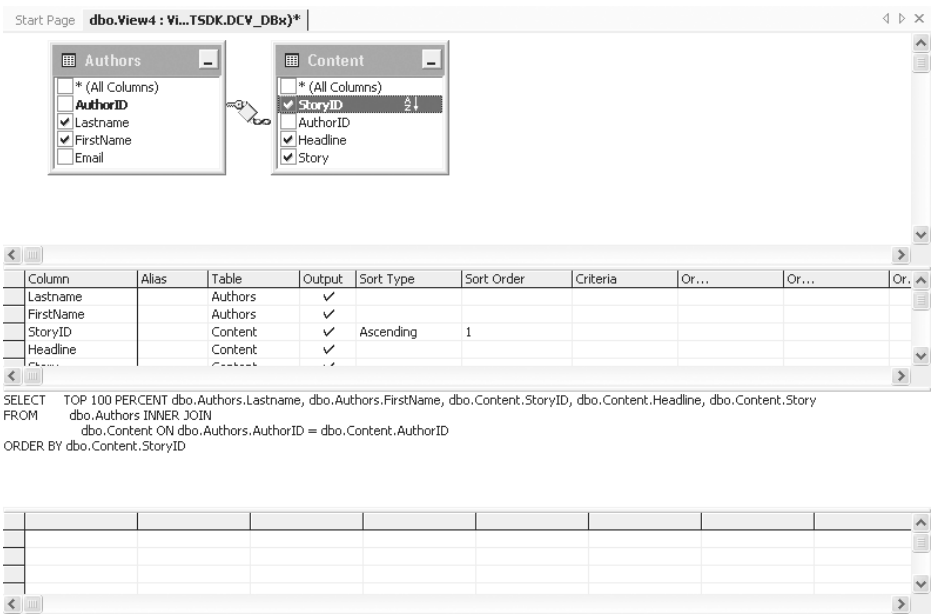


Figure 7-6. The View Design window

5. Click the check boxes for FirstName and LastName in the Authors table.

6. Click the check boxes for StoryID, Headline, and Story in the Content table.

7. Right-click StoryID and select Sort Ascending from the menu items.

8. Select Save View1 from the File menu.

9. Enter **Stories** into edit field.

10. Click OK.

Pretty painless, don't you think? You have the option of testing your view right there, too. Click the Run Query button on the main toolbar. (It is the button that has an exclamation point on it.) The View Design window is pretty powerful. If you play with it for a while, you will see what I mean.

Did you try the Run Query button and get nothing? Oops . . . I forgot to tell you to load some data into the database. You can do this with Visual Studio .NET as well. Simply double-click either of the tables you created, and an editable table will appear.

First enter the data for the authors. If you don't, you will not have an Author ID to enter into the `AuthorID` column in the `Content` view. Enter the data found in Table 7-3. Notice that there are no Author IDs to enter; this field is automatically created. In fact, Visual Studio .NET will yell at you if you try to enter something in the `AuthorID` column.

Table 7-3. Author Data

LASTNAME	FIRSTNAME	EMAIL
Doors	Bill	bill@contentmgr.com
Ellidaughter	Larry	larry@contentmgr.com
Fraser	Stephen	stephen@contentmgr.com

Now enter the data found in Table 7-4. Notice that `StoryID` cannot be entered. It, too, is an autogenerated number. You do have to enter `AuthorID`, though, because it is not automatically generated in this table.

Table 7-4. Content Data

AUTHORID	HEADLINE	STORY
1	.NET Is the Best	According to my research, the .NET product has no competition, though I am a little biased.
2	Oracle Is #1	Research suggests that it is the best database on the market, not that I have any biases in that conclusion.
3	Content Management Is Expensive	Not anymore. It now costs the price of a book and a little work.
1	SQL Server Will Be #1	This database has no real competition. But then again, I am a little biased.

Building Stored Procedures

It is important to cover this one last utility because you will use it quite frequently throughout the rest of this book. Truthfully, you don't have to use stored procedures because anything you can run using stored procedures you can run using standard SQL. So, why cover this utility at all?

There are two main reasons: First, stored procedures let a software developer call database code using function calls with parameters. Second, and more important, the utility is compiled before it gets loaded. This makes the calls to the database faster and more efficient because it has already been optimized.

Because you haven't covered ADO.NET yet, you will not be able to do much with the stored procedure you will create. Fortunately, Visual Studio .NET provides an option so that it can be tested.

Unlike the previous utilities, you have to actually code stored procedures. If you don't know SQL, don't worry because the coding is short and, I think, pretty self-explanatory. As always, there are many good books you can read to get a better understanding of it.

You will create a stored procedure to insert data into the Authors table. You already did this process manually, so you should have a good idea of what the stored procedure needs to do.

Creating a Stored Procedure

To create a stored procedure, follow these steps:

1. Right-click the Stored Procedures table from within the DCV_DB folder in the Server Explorer.

2. Select New Stored Procedure from the menu items. This will bring up an editing session with the default code shown in Listing 7-2.

Listing 7-2. Default Stored Procedure Code

```
CREATE PROCEDURE dbo.StoredProcedure1
/*
    (
        @parameter1 datatype = default value,
        @parameter2 datatype OUTPUT
    )
*/
AS
    /* SET NOCOUNT ON */
    RETURN
```

First you have to set up the parameters that will be passed from the program. Obviously, you need to receive all the mandatory columns that make up the row. In the `Authors` table's case, that's the entire column except `AuthorID`, which is autogenerated. Listing 7-3 shows the changes that need to be made to the default code provided in order to add parameters. Note that the comments `/* . . . */` are removed.

Listing 7-3. Setting the Parameters

```
CREATE PROCEDURE dbo.StoredProcedure1

    (
        @LastName NVARCHAR(32) = NULL,
        @FirstName NVARCHAR(32) = NULL,
        @Email NVARCHAR(64)    = NULL
    )

AS
```

Next, you turn on the `SET NOCOUNT`. This option prevents the message about the number of rows affected by the stored procedure from being returned to the calling program every time it is called.

```
SET NOCOUNT ON
```

Finally, you code the actual `insert` command. The key to this stored procedure is that instead of hard-coding the values to be inserted, you use the parameters you previously declared. Listing 7-4 is the final version of the stored procedure. Note that you rename the stored procedure to `InsertAuthor`.

Listing 7-4. InsertAuthor Stored Procedure

```
CREATE PROCEDURE dbo.InsertAuthor

    (
        @LastName NVARCHAR(32) = NULL,
        @FirstName NVARCHAR(32) = NULL,
        @Email NVARCHAR(64)    = NULL
    )

AS
    SET NOCOUNT ON

    INSERT INTO   Authors ( LastName,  FirstName,  Email)
    VALUES                (@LastName, @FirstName, @Email)

    RETURN
```

All that's left is to save the stored procedure. If you made a mistake while coding, the save will fail and an error message will tell you where the error is.

To run or debug the stored procedure, just right-click the newly created stored procedure and select Run Stored Procedure or Debug.

What Is ADO.NET?

You have tables, views, relationships, and stored procedures, so what's the big deal? The answer is one word: ADO.NET (or is that two words?). Gone are things that have been plaguing developers for some time now such as the variant, firewalls, and COM marshalling. If you don't know what these things are or why they were a problem, don't worry about it; just be glad they are gone.

ADO.NET is a set of classes that encompasses all aspects of accessing data sources within the .NET architecture. It is designed to provide full support for disconnected data access, while using an Extensible Markup Language (XML) format for transmitting data when data transfer is required. Chapter 8 contains more details about XML, so let's not worry about it for now. Just think of ADO.NET as a programmer's window into a data source, in our case the DCV_DB database.

The Benefits of Disconnected Data Access

Disconnected data access is a key feature of ADO.NET. Basically, it means that most of the time when you are accessing a database, you aren't getting the data from the database at all. Instead, you are accessing a copy of the data that was moved earlier to your client computer. Don't worry about all the technical issues surrounding this; just be glad that it works because it provides two major benefits:

- Less congestion on the database server because users are spending less time connected to it
- Faster access to the data because the data is already on the client

It also offers one benefit (associated with a disconnect access) that is less obvious: Data does not have to be stored in a databaselike format. Realizing this, Microsoft decided to implement ADO.NET using a strong typed XML format. A bonus that comes along with this is that data can be transmitted by means of XML and standard HTTP. This causes a further benefit: Firewall problems disappear. An HTTP response within the body of XML flows freely through a firewall (see Chapter 5 for information about HTTP), unlike the pre-ADO.NET technology's system-level COM marshalling requests.

Strong Data Types

Strong typed data saves the programmer the headache of having to use the variant data type and continually having to convert it to the data type desired. With ADO.NET, if a column in a database is an integer, you work, store, and retrieve it as an integer. The only time you have to do a data conversion is when you don't want it to be an integer.

Not having to do the type conversion will speed things up. It seems that speeding things up is a very common theme with ADO.NET.

Most Common ADO.NET Classes

If you spend a lot of time working with ADO.NET, you may have an opportunity to work with almost all of ADO.NET's classes. For the purpose of this book, however, I've trimmed these classes down to the following:

- Two managed providers
- `DataSet`
- `TablesCollection`
- `DataTable`
- `DataRow`
- `DataColumn`
- `RelationsCollection`
- `DataRelation`

All of these classes interact with each other in some way. Figure 7-7 shows the flow of the interaction. Essentially, the managed provider connects the data store to the `DataSet`. The `DataSet` stores the data in a `TablesCollection` made up of one or more `DataTables`. Each `DataTable` is made up of `DataRows` and `DataColumns`. All of the `DataTables` store their relationships in a `RelationsCollection` made up of `DataRelations`. Finally, all these classes can be affected by constraints. Simple, no?

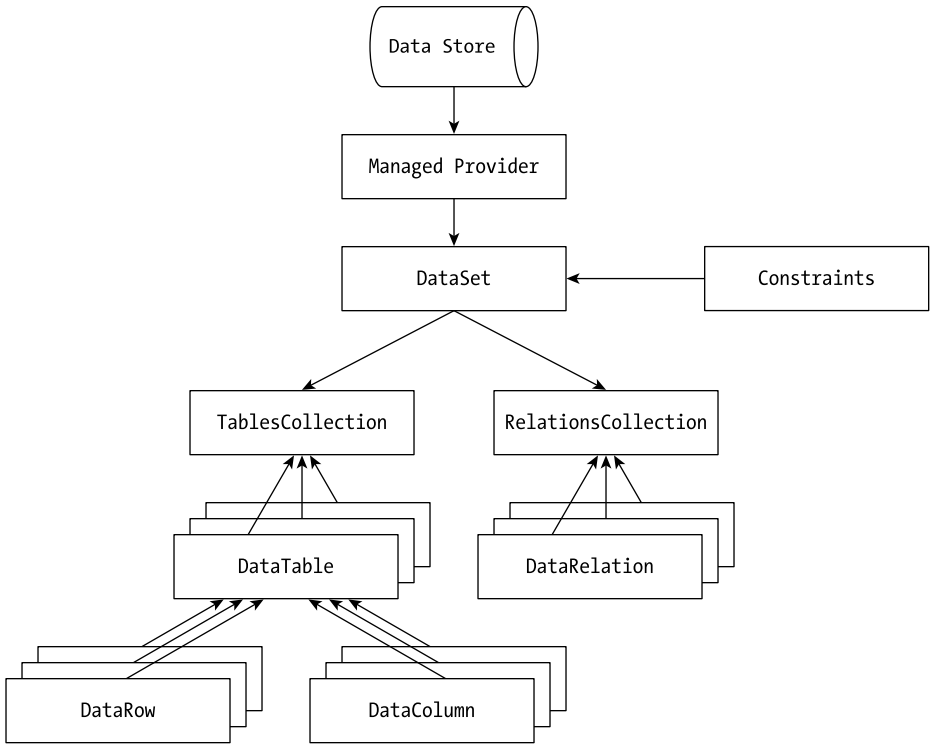


Figure 7-7. The ADO.NET class interaction

Managed Providers

Managed providers provide ADO.NET with the capability to connect and access data sources. Their main purpose, as far as most developers are concerned, is to provide support for the `DataAdapter` class. This class is essentially for mapping between the data store and the `DataSet`.

Currently only two managed providers exist for ADO.NET:

- *SQL managed provider:* Connects to Microsoft SQL Server version 7.0 or higher
- *OLE DB managed provider:* Connects to several supported OLE DB data sources

The managed provider you use determines which namespace you need to use. If you are using the SQL managed provider, you will need to use the `System.Data.SqlClient` namespace, which will include classes such as `SqlDataAdapter`, `SqlConnection`, and `SqlCommand`. On the other hand, if you are

using the OLE DB managed provider, you will need to use the `System.Data.OleDb` namespace, which will include classes such as `OleDbDataAdapter`, `OleDbConnection`, and `OleDbCommand`.

Once you have learned one managed provider, you have pretty much learned both because they are nearly the same, except for the `Sql` and `OleDb` prefixes and a few other small differences.

The most noticeable difference is that the OLE DB managed provider is slower than the SQL managed provider. This is mainly because the SQL managed provider accesses the Microsoft SQL Server directly, whereas the OLE DB managed provider must go through OLE DB first. As time goes on, Microsoft hopes that more database providers will create a managed provider for its database, thus removing the issue of slow access.

Because this book uses Microsoft SQL Server 2000, you will use the SQL managed provider and thus the namespace associated with it.

DataSet

The `DataSet` is the major controlling class of ADO.NET. A `DataSet` is a memory cache used to store all data retrieved from a data source, in most cases a database or XML file. The data source is connected to the `DataSet` using a managed provider. It also stores the format information about the data of which it is made up.

A `DataSet` consists of a `TablesCollection`, a `RelationsCollection`, and constraints. I will not use constraints in this book, but for the curious, two examples would be the `UniqueConstraint` and `ForeignKeyConstraint`, which ensure that the primary key value and foreign keys are unique.

A `DataSet` is data source independent. All it understands is XML. In fact, all data sent or received by the `DataSet` is in the form of an XML document. The `DataSet` has methods for reading and writing XML, and these are covered in Chapter 8.

TablesCollection

A `TablesCollection` is a standard collection class made up of one or more `DataTables`. Like any other collection class, it has methods such as `Add()`, `Remove()`, and `Clear()`. Usually, you will not use any of this functionality. Instead, you will use it to get access to the `DataTables` it stores. The method of choice in doing this will probably be to access the `TablesCollection` like an array.

DataTable

Put simply, a `DataTable` is one table of data stored in memory. A `DataTable` will also contain constraints, which help ensure the integrity of the data it is storing.

A `DataTable` is made up of zero or more `DataRows` because it is possible to have an empty table.

DataRow

The `DataRow` is where the data is actually stored. You will frequently access the data from the database using the `DataRow` as an array of columns.

DataColumn

The `DataColumn` is used to define the columns in a `DataTable`. Each `DataColumn` has a data type that determines the kind of data it can hold. A `DataColumn` also has properties similar to a database, such as `AllowNull` and `Unique`. If the `DataColumn` autoincrements, the `AutoIncrement` property is set. (Now, that makes more sense than Identity.)

RelationsCollection

A `RelationsCollection` is a standard collection class made up of one or more `DataRelations`. Like any other collection class, it has methods such as `Add()`, `Remove()`, and `Clear()`. Usually, as with the `TablesCollection`, you will not use any of this functionality. Instead, you will simply use it to get access to the `DataRelations` it stores.

DataRelation

A `DataRelation` is used to relate two `DataTables` together. It does this by matching `DataColumns` between two tables. You can almost think of it as the ADO.NET equivalent of the foreign-key relationship in a relational database (like you previously set).

One important thing you have to keep in mind is that the `DataColumns` must be the same data type. Remember that ADO.NET has strong data types, and when comparing different data types, one data type must be converted to the other. This conversion is not done automatically, thus the restriction of common data types.

If you are like me, you are probably saying, "Enough with theory, how do I code it?" Okay, let's have some more fun and play with the database you previously built.

Examples of ADO.NET Development

Two of the most common things you will do with ADO.NET are read data out of a database and insert data back into a database. Reading data out is so common that a special ASP.NET intrinsic control has been created to help display the data.

This section first shows how to read data out of a database the old-fashioned way and into a list. Then it shows how to read data from a database and into a DataGrid, the special ASP.NET intrinsic control. The next example shows how to load data into a database using stored procedures. Finally, you will finish by updating the Dynamic Content Viewer to get its data from a database.

Reading Data from a Database Using ADO.NET

Let's start pretty simple by getting some data out of the database and displaying it in a list. This will not look fancy, as you can see in Figure 7-8, but it will show you the basics of accessing data out of a database.

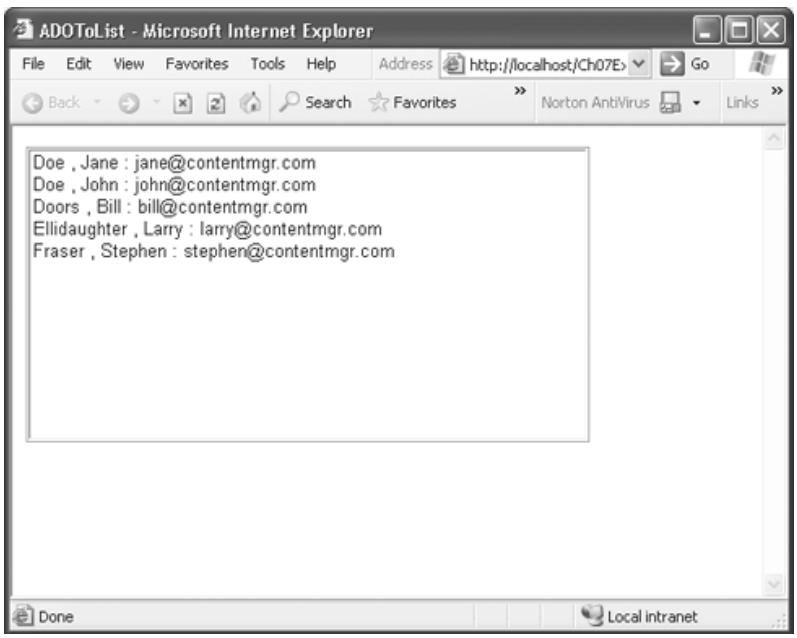


Figure 7-8. The ADOToList list

The first thing you are going to need is a new project in which to create your sample Web pages. If you need help doing this, Chapter 6 covers the process in detail. Name the project **Ch07Examples**. Then rename WebForm1.aspx to **ADOToList.aspx**.

Something you might have noticed is that the name of the class in the new ADOToList.aspx remains `WebForm1`. I recommend changing this to `ADOToList` to be consistent with the name of the ADOToList.cs file. The procedure is very simple.

Renaming WebForm1 to ADOToList

Follow these steps to rename `WebForm1` to `ADOToList`:

1. Change the edit window of ADOToList.aspx to HTML mode.

2. Find all occurrences of `WebForm1` and change them to `ADOToList`. There should be two. The first will be in the `Inherits` attribute in the first line of the HTML and the second will be within the `<title></title>` tag.

3. Select Save All from the main toolbar.

4. Open up the code for `ADOToList` in the edit window.

5. Find the occurrence (the name of the class) of `WebForm1` and change it to `ADOToList`.

6. Select Save All from the main toolbar to complete the procedure.

Now that you have a clean place to start your development from, the first thing you need to do is design your Web page. By the way, designing the Web page will almost always be the first thing you do in creating ASP.NET Web pages using Visual Studio .NET.

You should know how to drag controls from the Toolbox because this was covered in Chapter 6. For this example, all you need is to drag over a list box and resize it to be a little bigger. That's it for the design. (I'm sure you broke out in a sweat with that design effort.)

Now that you have someplace to put your data, let's go get some. Change the edit window to ADOToList.cs and let's proceed.

Coding the Reading of Data from a Database

The first thing you need to add is the namespace for the managed provider you will be using to access the data. This will give you access to all the classes needed to access the database using ADO.NET.

This book uses Microsoft SQL Server, so you'll use the SQLClient provider. If you are using a different database, just replace the prefix of every method starting with `Sql` with `OleDb`, and of course, you will have to change the connection string, but I'll get to that in a second.

When I add a namespace, I try to keep common namespaces together. In the case of ADOToList, add the highlighted code near the top of this listing:

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

For those of you using a database different than Microsoft SQL Server, use this highlighted code instead:

```
using System.Data;
using System.Data.OleDb;
using System.Drawing;
```

If you recall from Figure 7-7, the first piece in the puzzle of accessing a database using ADO.NET is the managed provider. You need to initialize your managed provider so that it can figure out where the database is and the privileges needed to access it. Then you tell it what SQL command to execute.

ADO.NET simplifies the process by allowing you to code this in one method. It is possible and necessary to break this method into its two parts, and this is shown in the section "Inserting Data Using a Stored Procedure" later in this chapter.

The hardest part of this piece of the coding (unless you don't know SQL) is figuring out what is the connection string. For SQL managed providers, this is fairly easy because it is made up of four parts:

- The location of the server: `server=localhost;`
- *The user ID:* `uid=sa;` (Note: This may vary with how you configured your database server.)
- *The user password:* `pwd=;` (In my case this is empty, but again this will depend on how you configured your database server.)
- The name of the database: `database=DCV_DB;`

It will look like this in the code:

```
"server=localhost;uid=sa;pwd=;database=DCV_DB"
```

The options are more varied for ADO managed providers, but it will probably look something like this:

```
"Provider=SQLOLEDB; Data Source=(local); Initial Catalog=DCV_DB; User ID=sa"
```

You will not be too elaborate when it comes to coding the SQL for this example because all you are doing is getting all the data out of the Authors table and sorting it by last name. If you are not happy with this simple SQL, you are welcome to experiment with it to your heart's content.

Here is my simple SQL:

```
"SELECT * FROM Authors ORDER BY LastName"
```

The only thing left to do now is call the managed provider's constructor with the SQL command and the connection string. Here is how the code looks:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        string ConnectStr = "server=localhost;uid=sa;pwd=;database=DCV_DB";
        string Cmd = "SELECT * FROM Authors ORDER BY LastName";

        SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);
```

The next piece of the puzzle is to create the `DataSet` and fill it with the data collected by the managed provider. You can do this simply by creating an instance of a `DataSet` and then letting the managed provider fill it in by calling its method `Fill()`, which takes two parameters. The first parameter is the newly created `DataSet`, and the second is the name of the table you want to extract from the managed provider. Here is how it should look:

```
DataSet ds = new DataSet();
        DAdpt.Fill(ds, "Authors");
```

The last piece of the puzzle is to simply take the table out of the DataSet and add the data from the table row by row into the list box designed earlier. I think the code is pretty self-explanatory, so I'll just list it now:

```
DataTable dt = ds.Tables["Authors"];

        foreach (DataRow dr in dt.Rows)
        {
            ListBox1.Items.Add(dr["LastName"] + ", " +
                               dr["FirstName"] + ": " +
                               dr["Email"]);
        }
```

The only thing of note in the preceding code—if you come from the C++ and Java world as I do—is what appears to be a string inside of an array. This handy syntax makes arrays in ADO.NET much more understandable than the equivalent:

```
        DataTable dt = ds.Tables[0];

        foreach (DataRow dr in dt.Rows)
        {
            ListBox1.Items.Add(dr[1] + ", " +
                               dr[2] + ": " +
                               dr[3]);
```

You also have to know the order of the columns in the array. As you might have noticed, LastName is the second element in the array, not the first. The first is AuthorID, something you are not adding to the list box. Also, note that the array starts at zero, but of course, you already knew that.

Okay, you are done. Listing 7-5 shows ADOToList.cs in its entirety so that you can check to make sure you coded everything correctly. Go ahead and save it and execute it. You should see something like Figure 7-8, which was shown at the beginning of this example.

Listing 7-5. ADOToList.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
```

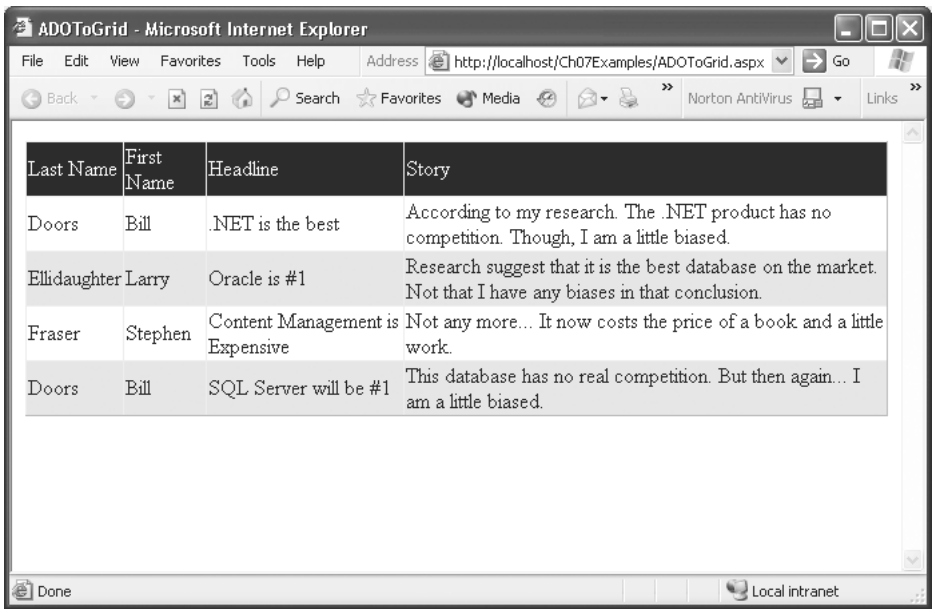
```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace Ch07Examples
{
    public class ADOToList : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.ListBox ListBox1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            if (!IsPostBack)
            {
                string ConnectStr =
                       "server=localhost;uid=sa;pwd=;database=DCV_DB";
                string Cmd = "SELECT * FROM Authors ORDER BY LastName";

                SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);

                DataSet ds = new DataSet();
                DAdpt.Fill (ds, "Authors");

                DataTable dt = ds.Tables["Authors"];

                foreach (DataRow dr in dt.Rows)
                {
                    ListBox1.Items.Add(dr["LastName"] + ", " +
                                       dr["FirstName"] + ": " +
                                       dr["Email"]);
                }
            }
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }
```

```
        private void InitializeComponent()
        {
            this.Load += new System.EventHandler (this.Page_Load);
        }
        #endregion
    }
}
```

Okay, let's move on to the next example.

Building an ASP.NET DataGrid

The previous example had a pretty boring display, don't you think? I think the authors of ADO.NET thought so, too, and decided to provide a standard intrinsic control called the DataGrid to provide a simple and elegant way of displaying data on a Web page. When you are finished, you will have a Web page that looks similar to Figure 7-9.



Last Name	First Name	Headline	Story
Doors	Bill	.NET is the best	According to my research. The .NET product has no competition. Though, I am a little biased.
Ellidaughter	Larry	Oracle is #1	Research suggest that it is the best database on the market. Not that I have any biases in that conclusion.
Fraser	Stephen	Content Management is Expensive	Not any more... It now costs the price of a book and a little work.
Doors	Bill	SQL Server will be #1	This database has no real competition. But then again... I am a little biased.

Figure 7-9. The ADOToGrid DataGrid

This example is actually simpler than the previous one when it comes to the code. The design is more difficult, but hey, it's an intrinsic control, how hard can that be?

Take my word for it, an intrinsic control can be tough, especially the DataGrid because it has quite a few different options available. In fact, this example will not even skim the surface. If you want more information about the DataGrid, you might want to check out the Microsoft documentation.

Let's not create a new project. Instead, you will add a new Web page to the existing project.

Adding a Web Page to a Project

Follow these steps to add a Web page to your project:

1. Right-click Ch07Examples.

2. Select Add Web Form from the Add submenu item. This will display a dialog box similar to the one shown in Figure 7-10.

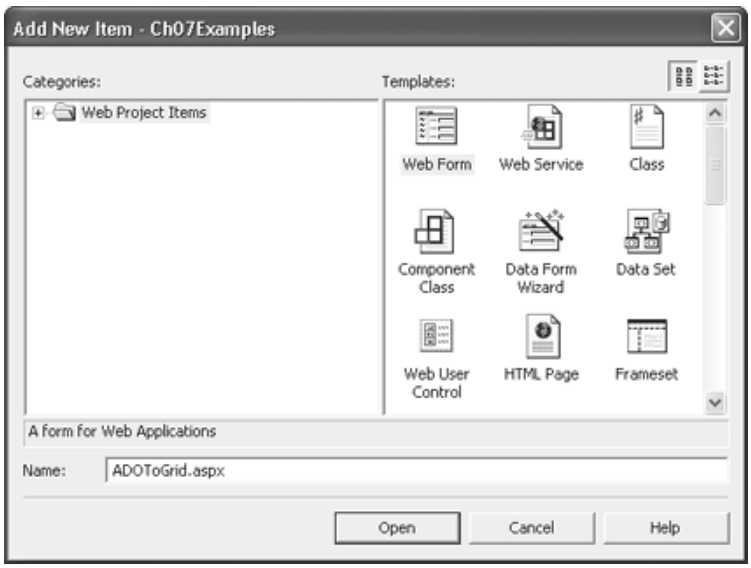


Figure 7-10. The Add New Item dialog box

3. Enter **ADOToGrid.aspx** in the Name field.

4. Click Open.

The best part is that everything has the name ADOToGrid, so now you don't have to do any renaming.

Coding the Connection of a Database to a DataGrid

Now that you have your new Web page, drag a DataGrid from the Toolbox to it. Leave it in its default state to start with and then come back and fix it up slightly after you have finished connecting it to the database. Save the design and then change the edit window to the Web page code ADOToGrid.cs.

First, remember to add the namespace of the managed provider you are using. As before, this book will use the SqlClient namespace, so the code should look like this:

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

As in the previous example, the next thing you have to do is connect to the managed provider. Note that once you know the connection string for the database to which you are connecting, you will be able to continually use it without having to do a single change ever again (unless you change your database server configuration).

Just for grins and giggles, you are going to get your data out of the database using the `Stories` view previously created. So, with this minor change, the code will look like this:

```
protected void Page_Load(object sender, EventArgs e)
{

    if (!IsPostBack)
    {
        string ConnectStr = "server=localhost;uid=sa;pwd=;database=DCV_DB";
        string Cmd = "SELECT * FROM Stories";

        SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);
```

Next, you create a `DataSet` and fill it with the data gathered by the managed provider. The only difference between this example and the previous one is that the second parameter of the `Fill()` method will change to the `Stories` view.

```
        DataSet ds = new DataSet();
        DAdpt.Fill (ds, "Stories");
```

The last thing you need to do is place the data from the `DataSet` into the DataGrid designed earlier. Do you remember from Chapter 6 that you can load a drop-down list using its `DataSource` property? You can do the exact same thing

with a DataGrid. All you need to find is an object that implements the IEnumerable interface. Fortunately, the DataTable has a property called `DefaultView` that provides a (what a coincidence!) default view of the table. This property also happens to implement the IEnumerable interface. After you load the `DataSource`, you must `DataBind()` it to the DataGrid, as shown in the following code:

```
DataGrid1.DataSource = ds.Tables["Stories"].DefaultView;
DataGrid1.DataBind();
```

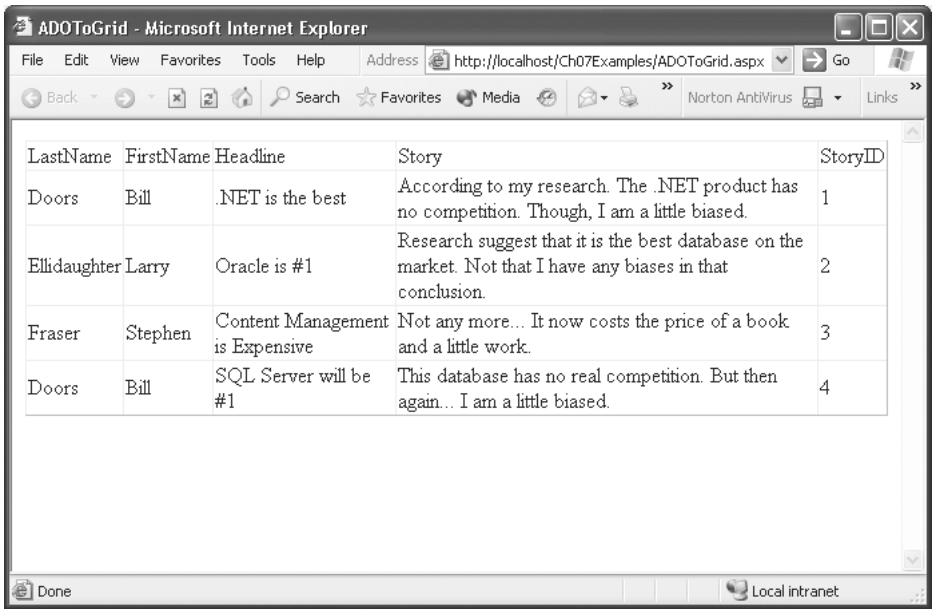
Just to make sure you coded correctly, Listing 7-6 shows the entire ADOToGrid.cs.

Listing 7-6. ADOToGrid.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace Ch07Examples
{
    public class ADOToGrid : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.DataGrid DataGrid1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            if (!IsPostBack)
            {
                string ConnectStr =
                  "server=localhost;uid=sa;pwd=;database=DCV_DB";
                string Cmd = "SELECT * FROM Stories";

                SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);

                DataSet ds = new DataSet();
                DAdpt.Fill(ds, "Stories");
```

```
                DataGrid1.DataSource = ds.Tables["Stories"].DefaultView;
                DataGrid1.DataBind();
            }
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }

        private void InitializeComponent()
        {
            this.Load += new System.EventHandler (this.Page_Load);
        }
        #endregion
    }
}
```

Now you can save, compile, and execute it. Notice in Figure 7-11 that the headings used by the DataGrid are the actual column names from the database and that the grid is quite boring. Let's change it to something more pleasant.



LastName	FirstName	Headline	Story	StoryID
Doors	Bill	.NET is the best	According to my research. The .NET product has no competition. Though, I am a little biased.	1
Ellidaughter	Larry	Oracle is #1	Research suggest that it is the best database on the market. Not that I have any biases in that conclusion.	2
Fraser	Stephen	Content Management is Expensive	Not any more... It now costs the price of a book and a little work.	3
Doors	Bill	SQL Server will be #1	This database has no real competition. But then again... I am a little biased.	4

Figure 7-11. The ADOToGrid DataGrid before facelift

Change the edit window back to the Web page ADOToGrid.aspx and select the HTML view.

Changing DataGrid Layout

Perform the following steps to change the DataGrid layout:

1. Select the DataGrid control in Design editor.

2. Select False in the AutoGenerateColumns property.

3. Click the ellipsis in the Columns property. This will display a DataGrid Properties dialog box, as shown in Figure 7-12.

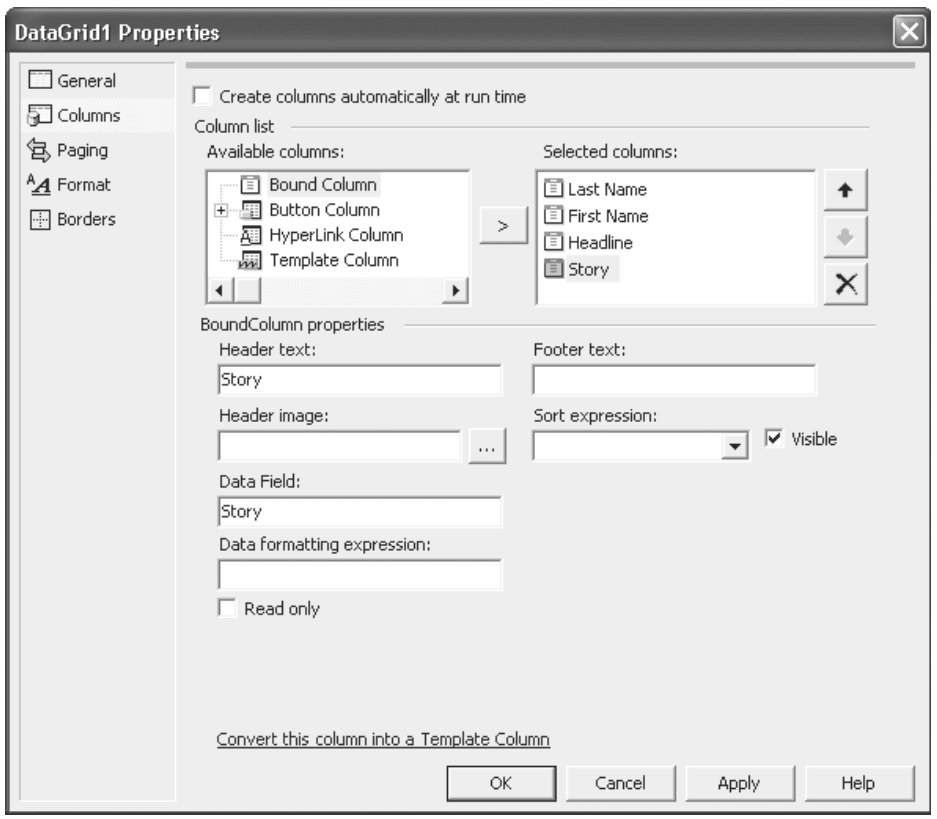


Figure 7-12. The DataGrid dialog box

4. Click Columns in left selection list.

5. Select Bound Column in the Available Columns list.

6. Click the Add arrow button to add a bound column to selected columns list.

7. Enter **Last Name** in the Header Text box.

8. Enter **LastName** in the Data Field box.

9. Repeat steps 5 through 8 for the following three columns:

HEADER TEXT	DATA FIELD
First Name	FirstName
Headline	Headline
Story	Story

10. Click the OK button.

11. Expand the AlternatingItemStyle property.

12. Enter **Gainsboro** in the BackColor property of AlternatingItemStyle.

13. Expand the HeaderStyle property.

14. Enter **Navy** in the BackColor property of HeaderStyle.

15. Enter **White** in the ForeColor property of HeaderStyle.

If you get sets of columns in the DataGrid—the first with the new heading and the second with the default heading—you forgot to set the `AutoGenerateColumns` property to False. (It took me a while to find it. Guess I should read the manual first before I use something.) After you do this, be warned: The DataGrid no longer generates all the columns in the table, only the ones you specify.

Listing 7-7 presents the finished HTML code of ADOToGrid.aspx, just in case you missed something when entering your own code.

Listing 7-7. ADOToGrid.aspx

```
<%@ Page language="c#" Codebehind="ADOToGrid.aspx.cs"
                        AutoEventWireup="false"
                        Inherits="Ch07Examples.ADOToGrid" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>ADOToGrid</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name=vs_defaultClientScript content="JavaScript">
    <meta name=vs_targetSchema
          content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body >
    <form id="ADOToGrid" method="post" runat="server">
      <asp:DataGrid id=DataGrid1 runat="server" AutoGenerateColumns="False">
        <HeaderStyle ForeColor="White" BackColor="Navy">
        </HeaderStyle>
        <AlternatingItemStyle BackColor="Gainsboro">
        </AlternatingItemStyle>
        <Columns>
          <asp:BoundColumn DataField="LastName" HeaderText="Last Name">
          </asp:BoundColumn>
          <asp:BoundColumn DataField="FirstName" HeaderText="First Name">
          </asp:BoundColumn>
          <asp:BoundColumn DataField="Headline" HeaderText="Head Line">
          </asp:BoundColumn>
          <asp:BoundColumn DataField="Story" HeaderText="Story">
          </asp:BoundColumn>
        </Columns>
      </asp:DataGrid>
    </form>
  </body>
</HTML>
```

Inserting Data Using a Stored Procedure

Do you remember from Chapter 1 the term "content management application (CMA)"? Figure 7-13 shows the building of a (very) rudimentary one for the Dynamic Content Viewer that you started in Chapter 6.

Figure 7-13. A Dynamic Content CMA

When I developed the Dynamic Content CMA Web page (which I shortened to DCCMA.aspx), I created it within the CH07Examples project. You can do this as well, or you can separate it out into its own project.

The first thing, as always, is to design your Web page. Figure 7-14 shows what I came up with as a simple design. Table 7-5 describes all the intrinsic controls so that you can build a similar Web page.

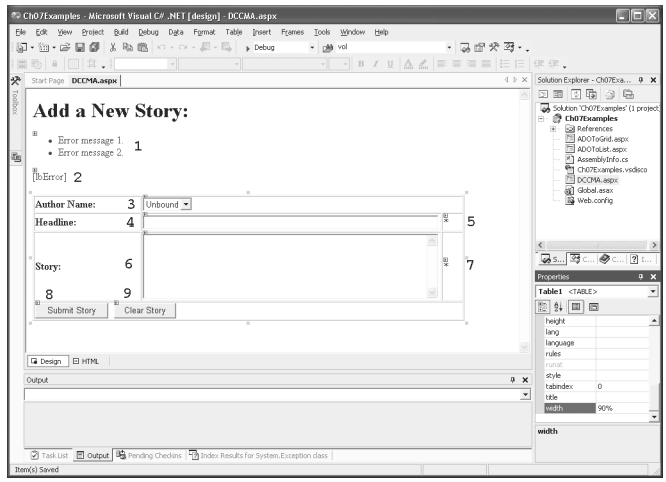


Figure 7-14. A Dynamic Content CMA design

Table 7-5. Dynamic Content CMA Design

NUMBER	CONTROL TYPE	ID	OTHER DETAILS
1	ValidationSummary	ValSum	Default values.
2	Label	lbError	Set ForeColor to Red and blank out all text.
3	DropDownList	ddlAuthor	Default values.
4	TextBox	tbHeadline	Set Width to 100%.
5	RequiredFieldValidator	rfvHeadline	Set Display to Dynamic, Error Message to **You must enter a Headline**, Text to *****, and Control To Validate to tbHeadline.
6	TextBox	tbStory	Set Rows to 6, Text Mode to MultiLine, and Width to 100%.
7	RequiredFieldValidator	rfvStory	Set Display to Dynamic, Error Message to **You must enter a Story**, Text to *****, and Control To Validate to tbStory.
8	Button	bnSubmit	Set Text to **Submit Story**.
9	Button	bnClear	Set Text to **Clear Story**.

Listing 7-8 shows the finished HTML code of DCCMA.aspx, just in case you missed something when entering your own code.

Listing 7-8. DCCMA.aspx

```
<%@ Page language="c#" Codebehind="DCCMA.cs"
                        AutoEventWireup="false"
                        Inherits="Ch07Examples.DCCMA" %>
<html>
  <head>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
          content="http://schemas.microsoft.com/intellisense/ie5">
  </head>
```

```
<body>
  <form id="DCCMA" method="post" runat="server">
    <h1>Add a new story:</h1>
    <p>
      <asp:ValidationSummary id=ValSum runat="server">
      </asp:ValidationSummary>
    </p>
    <p>
      <asp:Label id=lbError runat="server" ForeColor="Red">
      </asp:Label>
    </p>
    <table cellspacing=1 cellpadding=1 width="90%" border=1>
      <tr>
        <td width="25%"><strong>Author Name:</strong></td>
        <td width="70%" colspan="2">
          <asp:DropDownList id=ddlAuthor runat="server">
          </asp:DropDownList>
        </td>
      </tr>
      <tr>
        <td width="25%"><strong>Headline:</strong></td>
        <td width="70%">
          <asp:TextBox id=tbHeadline width="100%" runat="server">
          </asp:TextBox>
        </td>
        <td width="5%">
          <asp:RequiredFieldValidator id=rfvHeadline
                                      runat="server"
                                      ErrorMessage="You must enter a Headline"
                                      Display="Dynamic"
                                      ControlToValidate="tbHeadline">
          *
          </asp:RequiredFieldValidator>
        </td>
      </tr>
      <tr>
        <td width="25%"><strong>Story:</strong></td>
        <td width="70%">
          <asp:TextBox id=tbStory runat="server" Rows="6"
                       TextMode="MultiLine" width="100%">
          </asp:TextBox>
        </td>
```

```
          <td width="5%">
            <asp:RequiredFieldValidator id=RequiredFieldValidator2
                                        runat="server"
                                        ErrorMessage="You must enter a story"
                                        Display="Dynamic"
                                        ControlToValidate="tbStory">
            *
            </asp:RequiredFieldValidator>
          </td>
        </tr>
        <tr>
          <td colspan="3">
            <asp:Button id=bnSubmit runat="server" Text="Submit Story">
            </asp:Button>
             
            <asp:Button id=bnClear runat="server" Text="Clear Story">
            </asp:Button>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Alarms should be going off right now in your head. There are two buttons; how does the Web page differentiate between the two? Taking a look at the HTML source doesn't help either:

```
 <tr>
  <td colspan="3">
    <asp:Button id=bnSubmit runat="server" Text="Submit Story">
    </asp:Button>
     
    <asp:Button id=bnClear runat="server" Text="Clear Story">
    </asp:Button>
  </td>
</tr>
```

Other than the `id`, there is no difference as far as the HTML is concerned. Believe it or not, this is a good thing. Remember that you are trying to keep HTML and code logic separate. Because this is logic you are dealing with, it only seems appropriate that all the code that handles logic would handle this. Guess what? It does.

Adding Code Specific to a Button

To add code specific to a button, perform these steps:

1. Select the Clear Story button.
2. Click the Events button in the toolbar of the Properties dialog box.
3. Enter **bnClear_Click** in the Click entry field.
4. Change the edit window to DCCMA.cs.

You should notice that the highlighted code automatically gets added:

```
private void InitializeComponent()
{
    this.bnClear.Click += new System.EventHandler (this.bnClear_Click);
    this.Load += new System.EventHandler (this.Page_Load);
}

private void bnClear_Click (object sender, System.EventArgs e)
{

}
```

By the way, you could also double-click the button instead of doing the preceding four steps, but then you can't give the button a custom name without additional effort. Personally, I prefer to double-click and use the method name provided.

Basically, the added code adds an event handler to watch for events received by the bnClear button. When the event occurs, the `bnClear_Click()` method is executed after the `Page_Load()` method. Read that again: The `Page_Load()` method is always run first.

So, how do you use these two buttons? First, you have to create events for all buttons by double-clicking them, and then you place the logic specific to each in its respective event handler methods. If you have any generic code, which all methods need to execute, put that in the `Page_Load()` method. You will see this in action in a second when you examine the Web page logic code.

The Dynamic Content CMA Code Logic

As always, the first thing you add is the namespace to the managed provider you are using. For the SQL managed provider, the code is as follows:

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

With the SqlClient namespace ready, you can now tackle the coding of the Web page. The first thing you need to do is populate the author drop-down list. There is nothing new here, as you can see in Listing 7-9.

Listing 7-9. Populating the Author Drop-Down List in DCCMA

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!IsPostBack)
    {
        string ConnectStr = "server=localhost;uid=sa;pwd=;database=DCV_DB";
        string Cmd =
        "SELECT AuthorID, LastName, FirstName FROM Authors ORDER BY LastName";

        SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);

        DataSet ds = new DataSet();
        DAdpt.Fill(ds, "Authors");

        DataTable dt = ds.Tables["Authors"];

        foreach (DataRow dr in dt.Rows)
        {
            ddlAuthor.Items.Add(new ListItem(dr["LastName"] + ", " +
                                             dr["FirstName"],
                                             dr["AuthorID"].ToString()));
        }
    }
}
```

Clearing the form is very straightforward. You need to set the text properties of both text boxes to an empty string in the new event method created by double-clicking the Clear Story button. Listing 7-10 shows the story clearing method.

Listing 7-10. Clearing a Web Form

```
private void bnClear_Click (object sender, System.EventArgs e)
{
    tbHeadline.Text = "";
    tbStory.Text = "";
}
```

Now, let's move on to the meat of this example: the inserting of data using a stored procedure.

The first thing you need is the stored procedure. I already covered building a stored procedure, and there is nothing new about this one, as you can see in Listing 7-11.

Listing 7-11. The InsertContent Stored Procedure

```
CREATE PROCEDURE dbo.InsertContent
    (
        @AuthorID INT      = NULL,
        @Headline CHAR(64) = NULL,
        @Story    TEXT     = NULL
    )
AS
    SET NOCOUNT ON

    INSERT INTO   Content ( AuthorID,  Headline,  Story)
    VALUES                (@AuthorID, @Headline, @Story)

    RETURN
```

The approach I use to execute a stored procedure is two-pronged. The first prong is located in the code portion of the Web page, and the second is located in a helper class. Both of these contain new code, so you will break them down to make following it easier.

All the code is located in the event method of the bnSubmit button. You put it there instead of in the `Load_Page()` method because, if you didn't, the Clear Story button would also run it. Thus, it would cause the headline and story to be stored in the database even though you just want it cleared. Putting it into its own method means it will only be called when the Submit Story button is clicked.

The first thing in the `bnSubmit_Click` method is a second way to set up the managed provider. As hinted before, you can set up a connection to the database without submitting a SQL command. Obviously, you don't want to send one because you will be using a stored procedure.

The code is easy to follow. All you need to do is create an instance of a `SqlConnection` and then set its `ConnectionString` property with a merged data

source, user ID, password, and database name string, just like the connection string you used with a `SqlDataAdapter`. Once you have the connection prepared, you create an instance of the helper class, content, passing it the `SqlConnection` in a parameter to its constructor.

```
private void bnSubmit_Click (object sender, System.EventArgs e)
{
    if (Page.IsValid)
    {
        SqlConnection myConnection = new SqlConnection();

        myConnection.ConnectionString =
            "server=localhost;uid=sa;pwd=;database=DCV_DB";

        Content content = new Content(myConnection);
```

A neat feature of ASP.NET is that exception handling works fine, no matter which language you are implementing it in. You use this to your advantage here because you call the `Insert()` method of the helper class, confident that if something goes wrong, the exception handling routine will catch it. The `Insert()` method takes three parameters: `AuthorID`, `Headline`, and `Story`. This is not coincidental because it corresponds directly to the parameters of the stored procedure created previously. Because ADO.NET is strong typed, you convert the Author drop-down list value to an integer type. After it is inserted, you clear the headline and story, so it is ready for the next entries.

```
    try
    {
        content.Insert(Convert.ToInt16(ddlAuthor.SelectedItem.Value),
                       tbHeadline.Text,
                       tbStory.Text);

        tbHeadline.Text = "";
        tbStory.Text = "";
    }
    catch (SqlException err)
    {
        lbError.Text = "Sorry, the following error occured: " + err.Message;
    }
```

Just to make sure you didn't miss anything, Listing 7-12 shows DCCMA.cs in its entirety.

Listing 7-12. DCCMA.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace Ch07Examples
{
    public class DCCMA : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Button bnClear;
        protected System.Web.UI.WebControls.Button bnSubmit;
        protected System.Web.UI.WebControls.RequiredFieldValidator
                  RequiredFieldValidator2;
        protected System.Web.UI.WebControls.TextBox tbStory;
        protected System.Web.UI.WebControls.RequiredFieldValidator
                  rfvHeadline;
        protected System.Web.UI.WebControls.TextBox tbHeadline;
        protected System.Web.UI.WebControls.DropDownList ddlAuthor;
        protected System.Web.UI.WebControls.Label lbError;
        protected System.Web.UI.WebControls.ValidationSummary ValSum;

        private void Page_Load(object sender, System.EventArgs e)
        {
            if (!IsPostBack)
            {
                string ConnectStr =
                       "server=localhost;uid=sa;pwd=;database=DCV_DB";
                string Cmd =
         "SELECT AuthorID, LastName, FirstName FROM Authors ORDER BY LastName";

                SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);

                DataSet ds = new DataSet();
                DAdpt.Fill(ds, "Authors");
```

```
            DataTable dt = ds.Tables["Authors"];

            foreach (DataRow dr in dt.Rows)
            {
                ddlAuthor.Items.Add(new ListItem(dr["LastName"] + ", " +
                                                 dr["FirstName"],
                                                 dr["AuthorID"].ToString()));
            }
        }
    }

    #region Web Form Designer generated code
    override protected void OnInit(EventArgs e)
    {
        InitializeComponent();
        base.OnInit(e);
    }

    private void InitializeComponent()
    {
        bnClear.Click += new System.EventHandler (this.bnClear_Click);
        bnSubmit.Click += new System.EventHandler (this.bnSubmit_Click);
        this.Load += new System.EventHandler (this.Page_Load);
    }
    #endregion

    private void bnSubmit_Click (object sender, System.EventArgs e)
    {
        if (Page.IsValid)
        {
            SqlConnection myConnection = new SqlConnection();

            myConnection.ConnectionString =
                "server=localhost;uid=sa;pwd=;database=DCV_DB";

            Content content = new Content(myConnection);

            try
            {
                content.Insert(Convert.ToInt16(
                                    ddlAuthor.SelectedItem.Value),
                               tbHeadline.Text,
                               tbStory.Text);
```

```
                    tbHeadline.Text = "";
                    tbStory.Text = "";
                }
                catch (SqlException err)
                {
                    lbError.Text =
                      "Sorry, the following error occured: " + err.Message;
                }
            }
        }

        private void bnClear_Click (object sender, System.EventArgs e)
        {
            tbHeadline.Text = "";
            tbStory.Text = "";
        }
    }
}
```

All that is left for this Web page is the helper class. This class is actually derived from a helper class used by the CMS system that this book will develop, so it is a little more feature-rich than it needs to be.

After creating the new class, you need to add the necessary namespaces to access the SQL managed provider and a couple of member variables. The first variable will hold the `SqlConnection` passed by the constructor. The second is a copy of the `SqlCommand` you will be building. This is so that after it has been built once, it doesn't have to be built again.

After the variables is the constructor, which only initializes the `SqlConnection` member variable.

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

namespace Ch07Examples
{
    public class Content
    {
        private SqlConnection m_Connection;
        private SqlCommand    m_InsertCommand;
```

```
        public Content(SqlConnection Connection)
        {
            m_Connection = Connection;
        }
        ...
    }
}
```

Now you come to the `Insert()` method. The method looks complex but it's not. First, you check whether the `Insert()` method has been run before. If it has, you can bypass the command-building code because you stored it off the first time you accessed the method.

You call a managed provider method, called `SqlCommand`, to set up the command to call the stored procedure `InsertContent` using the connection passed by the constructor. You set the `CommandType` property to `StoredProcedure`. Finally, you add parameter definitions to `SqlCommand` so that it can pass data to the stored procedure.

```
        public void Insert(int AuthorID, string Headline, string Story)
        {
            SqlParameterCollection Params;

            if ( m_InsertCommand == null )
            {
                // Only create Insert command the first time
                m_InsertCommand = new SqlCommand("InsertContent",
                                                 m_Connection);
                m_InsertCommand.CommandType  = CommandType.StoredProcedure;
                Params = m_InsertCommand.Parameters;

                Params.Add(new SqlParameter("@AuthorID", SqlDbType.Int));
                Params.Add(new SqlParameter("@Headline", SqlDbType.Char, 64));
                Params.Add(new SqlParameter("@Story",    SqlDbType.Text));
            }
```

Next, you load the `SqlCommand`'s parameters with the values passed in through the method's parameters.

```
            Params = m_InsertCommand.Parameters;

            Params["@AuthorID"].Value = AuthorID;
            Params["@Headline"].Value = Headline;
            Params["@Story"].Value    = Story;
```

You use a little trick of exception handling. By including the `finally` clause of the exception, you are guaranteed that the connection will be closed, even on error. The actual exception is caught in the method that calls this one.

Finally, you open a connection to the database and execute the stored procedure using the `ExecuteNonQuery()` method.

```
            try
            {
                m_Connection.Open();
                m_InsertCommand.ExecuteNonQuery();
            }
            finally
            {
                m_Connection.Close();
            }
        }
```

Not that bad, was it? Save, compile, and execute it. Try entering some HTML formatting into the stories. You will see in the next example that they come out formatted as you specified.

Now let's move on to the last example: revisiting the Dynamic Content Viewer.

Updating the Dynamic Content Viewer with ADO.NET

Let's finish off the chapter by revisiting the Dynamic Content Viewer you created in Chapter 6. As you can see in Figure 7-15, the Web page has become much simpler because all the personalization code was removed, thus removing all the validation intrinsic controls because they were no longer needed. A label called `lbAuthor` was also added in a smaller font so that you can see who wrote the story. All the code needed to create the design is in Listing 7-13.

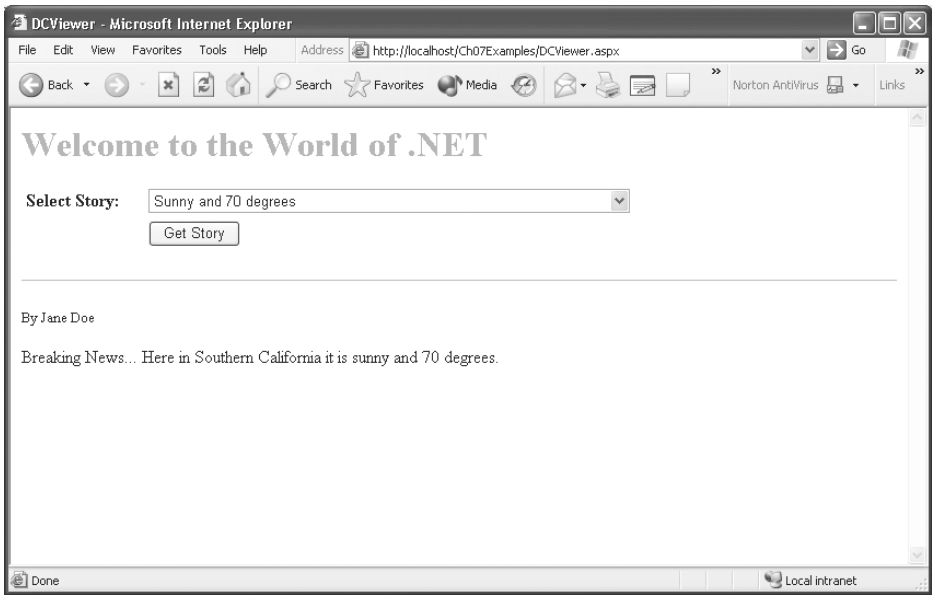


Figure 7-15. The updated Dynamic Content Viewer

Listing 7-13. DCViewer.aspx

```
<%@ Page language="c#" Codebehind="DCViewer.aspx.cs"
                        AutoEventWireup="false"
                        Inherits="Ch07Examples.DCViewer" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>DCViewer</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name=vs_defaultClientScript content="JavaScript">
    <meta name=vs_targetSchema
          content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body >
    <form id="DCViewer" method="post" runat="server">
      <H1> <FONT color="#66cc00">Welcome to the world of .NET </FONT></H1>
      <P>
        <TABLE id="Table1" cellSpacing=1 cellPadding=3 width="100%" border=0>
          <TR>
            <TD width="20%"><STRONG>Select Story:</STRONG></TD>
            <TD width ="80%">
```

```
                <asp:DropDownList id=ddlStories runat="server">
                </asp:DropDownList>
              </TD>
            </TR>
            <TR>
              <TD width ="20%"> </TD>
              <TD width ="80%">
                <asp:Button id=bnGetStory runat="server" Text="Get Story">
                </asp:Button>
              </TD>
            </TR>
          </TABLE>
        </P>
        <P>
          <HR width="100%" SIZE="1">
        <P>
          <asp:Label id=lbAuthor runat="server" Font-Size="X-Small">
          </asp:Label>
        </P>
        <P>
          <asp:Label id=lbStory runat="server">
          </asp:Label>
        </P>
      </form>
    </body>
</HTML>
```

As you can see in Listing 7-14, there is really nothing new when it comes to coding the functionality of the Web page. You load the drop-down list from the `Content` table the first time the page is loaded, and all subsequent times you check which value has been selected in the drop-down list and do a SQL select on the `Stories` view to find it. Then you display the author and story using their respective labels.

Listing 7-14. DCViewer.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
```

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace Ch07Examples
{
    public class DCViewer : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label lbStory;
        protected System.Web.UI.WebControls.Label lbAuthor;
        protected System.Web.UI.WebControls.Button bnGetStory;
        protected System.Web.UI.WebControls.DropDownList ddlStories;

        private void Page_Load(object sender, System.EventArgs e)
        {
            if (!IsPostBack)
            {
                string ConnectStr =
                       "server=localhost;uid=sa;pwd=;database=DCV_DB";
                string Cmd = "SELECT StoryID, Headline FROM Content";

                SqlDataAdapter DAdpt = new SqlDataAdapter(Cmd, ConnectStr);

                DataSet ds = new DataSet();
                DAdpt.Fill(ds, "Content");

                DataTable dt = ds.Tables["Content"];

                foreach (DataRow dr in dt.Rows)
                {
                    ddlStories.Items.Add(
                        new ListItem(dr["Headline"].ToString(),
                                     dr["StoryID"].ToString()));
                }
            }
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }
```

```
        private void InitializeComponent()
        {
            bnGetStory.Click += new System.EventHandler(this.bnGetStory_Click);
            this.Load += new System.EventHandler (this.Page_Load);
        }
        #endregion

        private void bnGetStory_Click(object sender, System.EventArgs e)
        {
            if (Page.IsValid)
            {
                string ConnectStr =
                    "server=localhost;uid=sa;pwd=;database=DCV_DB";
                string Cmd = "SELECT * FROM Stories WHERE StoryID = "
                    + ddlStories.SelectedItem.Value.ToString();

                SqlDataAdapter DAdpt =
                    new SqlDataAdapter(Cmd, ConnectStr);

                DataSet ds = new DataSet();
                DAdpt.Fill (ds, "Stories");

                DataRow dr = ds.Tables["Stories"].Rows[0];

                lbAuthor.Text = "By " + dr["FirstName"] + " " + dr["LastName"];
                lbStory.Text = dr["Story"].ToString();
            }
        }
    }
}
```

There are only two things of note in Listing 7-14. The first is how you populate the `DataRow`:

```
DataRow dr = ds.Tables["Stories"].Rows[0];
```

What is interesting is that because you know that only one row will be retrieved, you can directly populate the `DataRow` with the first row of the table. Remember that the `Rows` collection starts with zero.

The second thing is that you need to be careful about the use of the `Page.IsValid` variable. I had originally coded this example with the retrieving of the stories from the database within the `Page_Load()` method. Unfortunately, this causes an error because when the page is run an exception is thrown due to

`Page.IsValid` not being set. To fix this problem, `Page.IsValid` needs to be coded within an event handler with the control property `CausesValidation` set to its default `true` (as we have done previously) or the Web page code must call `Page.Validate` itself.

All that is left to do is save, compile, and execute. All the stories you entered in the previous example show up, and if you formatted them using HTML, the formatting appears as expected.

Summary

This chapter covered in detail the database utilities provided by Visual Studio .NET and ADO.NET.

It started by covering four common database utilities:

- Creating databases
- Adding tables
- Adding views
- Building stored procedures

Next, it discussed ADO.NET and explored some of its common classes. It finished with four example programs:

- Reading data from a database using ADO.NET
- Building an ASP.NET DataGrid
- Inserting data using a stored procedure
- Updating the Dynamic Content Viewer with ADO.NET

In the next chapter, you are going to have some fun with XML.