



ADO.NET Fundamentals

At the beginning of this book, you learned that ASP.NET is just one component in Microsoft's ambitious .NET platform. As you know, .NET also includes modern languages and a toolkit of classes that allows you to do everything from handling errors to analyzing XML documents. In this chapter, you'll explore another one of the many features in the .NET Framework: the ADO.NET data access model.

Quite simply, ADO.NET is the technology that .NET applications use to interact with a database. In this chapter, you'll learn about ADO.NET and the family of objects that provides its functionality. You'll also learn how to put these objects to work by creating simple pages that retrieve and update database records. However, you won't learn about one of the most interesting ways to access a database—using a code-generation and data-modeling tool called LINQ to Entities. Although LINQ to Entities is a powerful and practical way to generate a data model for your database, it may be overkill for your application, it may be unnecessarily complex, or it may not give you all the control you need (for example, if you want to perform unusual data tasks or implement elaborate performance-optimizing techniques). For these reasons, every ASP.NET developer should start by learning the ADO.NET fundamentals that are covered in this chapter.

■ **Note** The LINQ to Entities feature is a *higher-level* model. That means it uses the ADO.NET classes you'll learn about in this chapter to do its dirty work. After you've mastered the essentials of ADO.NET, you'll be ready to explore LINQ to Entities in Chapter 24.

Understanding Databases

Almost every piece of software ever written works with data. In fact, a typical web application is often just a thin user interface shell on top of sophisticated data-driven code that reads and writes information from a database. Often website users aren't aware (or don't care) that the displayed information originates from a database. They just want to be able to search your product catalog, place an order, or check their payment records.

The most common way to manage data is to use a database. Database technology is particularly useful for business software, which typically requires sets of related information. For example, a typical database for a sales program consists of a list of customers, a list of products, and a list of sales that draws on information from the other two tables. This type of information is best described by using a *relational model*, which is the philosophy that underlies all modern database products, including SQL Server, Oracle, and even Microsoft Access.

As you probably know, a relational model breaks information down to its smallest and most concise units. For example, a sales record doesn't store all the information about the products that were sold. Instead, it stores just a product ID that refers to a full record in a product table, as shown in Figure 14-1.

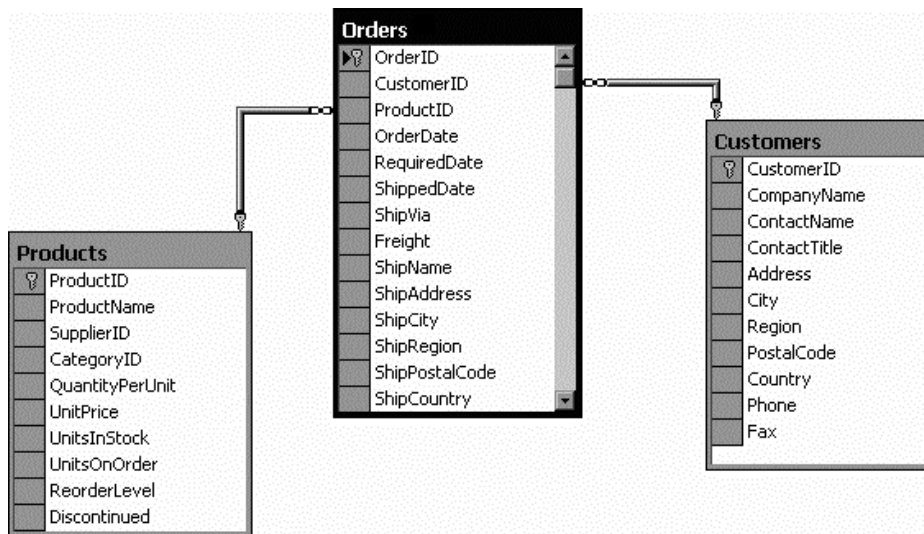


Figure 14-1. Basic table relationships

Although it's technically possible to organize data into tables and store it on the hard drive in one or more files (perhaps using a standard such as XML), this approach wouldn't be very flexible. Instead, a web application needs a full *relational database management system* (RDBMS), such as SQL Server. The RDBMS handles the data infrastructure, ensuring optimum performance and reliability. For example, the RDBMS takes the responsibility of providing data to multiple users simultaneously, disallowing invalid data, and using transactions to commit groups of actions at once.

In most ASP.NET applications, you'll need to use a database for some tasks. Here are some basic examples of data at work in a web application:

- E-commerce sites (for example, Amazon.com) use detailed databases to store product catalogs. They also track orders, customers, shipment records, and inventory information in a huge arrangement of related tables.
- Search engines (for example, Google) use databases to store indexes of page URLs, links, and keywords.
- Knowledge bases (for example, Microsoft Support) use less-structured databases that store vast quantities of information or links to various documents and resources.
- Media sites (for example, The New York Times) store their articles in databases.

You probably won't have any trouble thinking about where you need to use database technology in an ASP.NET application. What web application couldn't benefit from a guest book that records user comments or a simple e-mail address submission form that uses a back-end database to store a list of potential customers or contacts? This is where ADO.NET comes into the picture. ADO.NET is a technology designed to let an ASP.NET program (or any other .NET program, for that matter) access data.

■ **Tip** If you're a complete database novice, you can get up to speed on essential database concepts by using the video tutorials at www.asp.net/sql-server/videos. There you'll find more than nine hours of instruction that describes how to use SQL Server Express. (The content was originally prepared with SQL Server Express 2005, but it remains relevant for SQL Server 2012.) The tutorials move from absolute basics—covering topics such as database data types and table relationships—to more-advanced subject matter such as full-text search, reporting services, and network security.

Configuring Your Database

Before you can run any data access code, you need a database server to take your command. Although there are dozens of good options, all of which work equally well with ADO.NET (and require essentially the same code), a significant majority of ASP.NET applications use Microsoft SQL Server.

This chapter includes code that works with SQL Server 7 or later, although you can easily adapt the code to work with other database products. If you don't have a full version of SQL Server, there's no need to worry—you can use the free SQL Server Express (as described in the next section). It includes all the database features you need to develop and test a web application.

■ **Note** This chapter (and the following two chapters) use examples drawn from the pubs and Northwind databases. These databases aren't preinstalled in modern versions of SQL Server. However, you can easily install them by using the scripts provided with the online samples. See the `readme.txt` file for full instructions, or refer to the "Using the `sqlcmd` Command-Line Tool" section later in this chapter.

Using SQL Server Express

If you don't have a test database server handy, you may want to use SQL Server 2012 Express Edition. It's a scaled-down version of SQL Server that's free to distribute. SQL Server Express has certain limitations—for example, it can use only one CPU, four internal CPU cores, and a maximum of 1GB of RAM. Also, SQL Server Express databases can't be larger than 10GB. However, it's still remarkably powerful and suitable for many midscale websites. Even better, you can easily upgrade from SQL Server Express to a paid version of SQL Server if you need more features later.

There are actually two versions of SQL Server 2012 Express, which makes life slightly confusing. You can download the standalone edition, with or without database tools, from www.microsoft.com/express/sql. This is the version you want if you're installing it on a real web server (and you don't have the full version of SQL Server). The second version is the awkwardly named SQL Server 2012 Express LocalDB, which is included with all versions of Visual Studio. It has almost exactly the same functionality, but it's intended to be a testing tool for development purposes only (and it doesn't include any extra tools).

In this chapter, we refer to both editions as SQL Server Express, unless we need to highlight a difference between the two.

For a comparison between SQL Server Express and other editions of SQL Server, refer to www.microsoft.com/sqlserver/en/us/editions.aspx. SQL Server Express is included with the Visual Studio installation, but if you need to download it separately or you want to download the free graphical management tools that work with it, go to www.microsoft.com/express/database.

Browsing and Modifying Databases in Visual Studio

As an ASP.NET developer, you may have the responsibility of creating the database required for a web application. Alternatively, the database may already exist, or it may be the responsibility of a dedicated database administrator. If you're using a full version of SQL Server, you'll probably use a graphical tool such as SQL Server Management Studio to create and manage your databases.

■ **Tip** SQL Server Express doesn't include SQL Server Management Studio in the download that you use to install it. However, you can download it separately from www.microsoft.com/express/database. Click the Download SQL Server 2012 Express button to show a list of downloadable packages. Then click the Download button next to SQL Server Management Studio Express (Tools Only).

If you don't have a suitable tool for managing your database, or you don't want to leave the comfort of Visual Studio, you can perform many of the same tasks by using Visual Studio's Server Explorer window. (Confusingly enough, the Server Explorer window is called the Database Explorer window in Visual Studio Express for Web.)

You may see a tab for the Server Explorer on the right side of the Visual Studio window, grouped with the Toolbox and collapsed. If you do, click the tab to expand it. If not, choose View ► Server Explorer to show it (or View ► Database Explorer in Visual Studio Express for Web).

Using the Data Connections node in the Server Explorer, you can connect to existing databases or create new ones. Assuming you've installed the pubs database (see the readme.txt file for instructions), you can create a connection to it by following these steps:

1. Right-click the Data Connections node and choose Add Connection.
2. When the Choose Data Source window appears, select Microsoft SQL Server and then click Continue.
3. If you're using a full version of SQL Server, enter **localhost** as your server name. This indicates that the database server is the default instance on the local computer. (Replace this with the name of a remote computer if needed.) SQL Server Express is a bit different. If you're using SQL Server Express LocalDB (the version that's included with Visual Studio), you need to enter **(localdb)\v11.0** instead of **localhost**, as shown in Figure 14-2. If you've downloaded the full edition of SQL Server Express, you need to enter **localhost\SQLEXPRESS** instead.

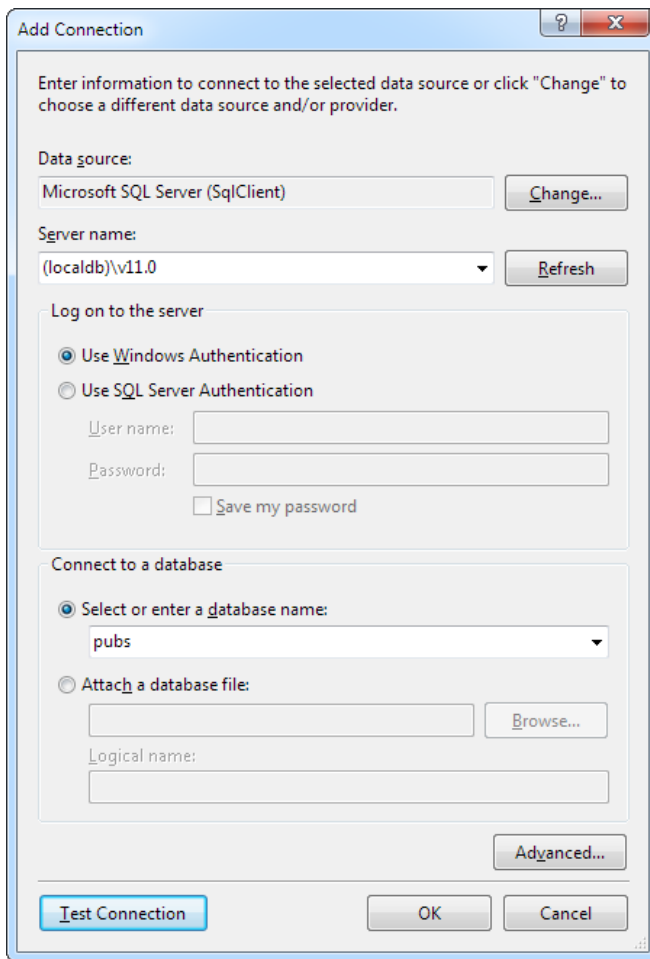


Figure 14-2. Creating a connection in Visual Studio

4. Click the Test Connection button to verify that this is the location of your database. If you haven't installed a database product yet, this step will fail. Otherwise, you'll know that your database server is installed and running.
5. In the Select or Enter a Database Name list, choose the pubs database. (In order for this to work, the pubs database must already be installed. You can install it by using the database script that's included with the sample code, as explained in the following section.) If you want to see more than one database in Visual Studio, you'll need to add more than one data connection.

■ **Tip** Alternatively, you can choose to create a new database by right-clicking the Data Connections node and choosing Create New SQL Server Database.

- Click OK. The database connection appears in the Server Explorer window. You can now explore its groups to see and edit tables, stored procedures, and more. For example, if you right-click a table and choose Show Table Data, you'll see a grid of records that you can browse and edit, as shown in Figure 14-3.

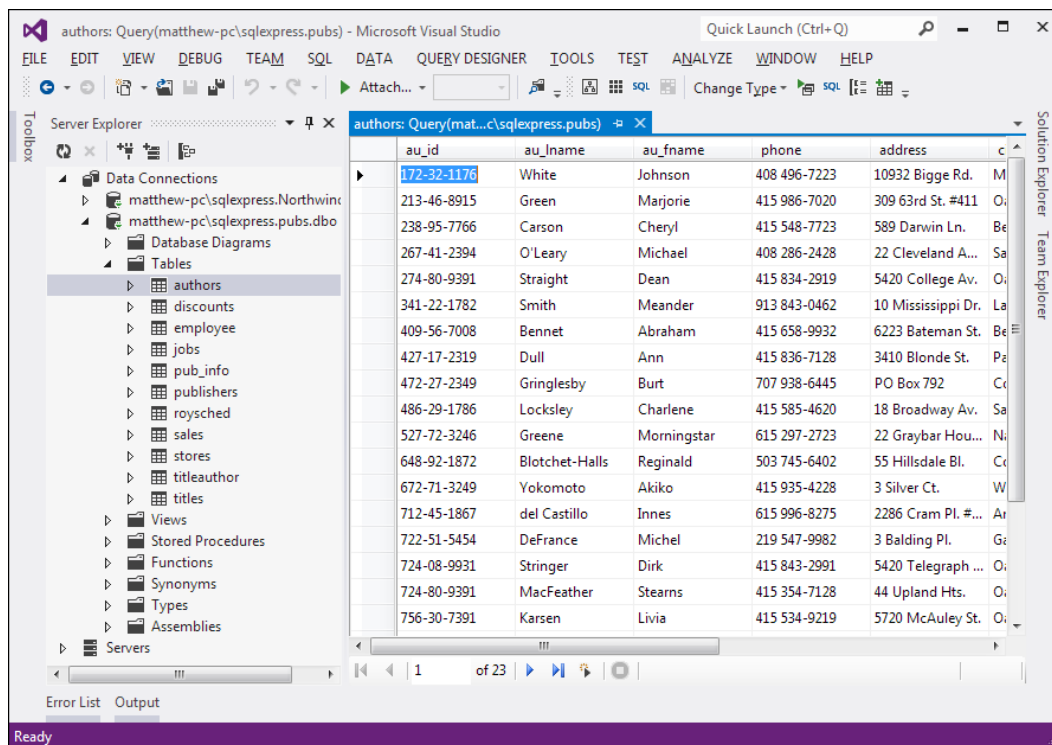


Figure 14-3. Editing table data in Visual Studio

■ **Tip** The Server Explorer window is particularly handy if you're using SQL Server Express, which gives you the ability to place databases directly in the App_Data folder of your web application (instead of placing all your databases in a separate, dedicated location). If Visual Studio finds a database in the App_Data folder, it automatically adds a connection for it to the Data Connections group. To learn more about this feature, check out the "Making User Instance Connections" section later in this chapter.

Using the sqlcmd Command-Line Tool

SQL Server includes a handy command-line tool named *sqlcmd.exe* that you can use to perform database tasks from a Windows command prompt. Compared to a management tool such as SQL Server Management Studio, sqlcmd doesn't offer many frills. It's just a quick-and-dirty way to perform a database task. Often sqlcmd is used in a batch file—for example, to create database tables as part of an automated setup process.

The sqlcmd tool is found in the directory `c:\Program Files\Microsoft SQL Server\110\Tools\Binn`. To run it, you need to open a Command Prompt window in this location. (One easy way to do that is to browse to the version folder in Windows Explorer, hold down Shift, and right-click the Binn folder. Then, choose "Open command menu window here" from the menu.) Once you're in the right folder, you can use sqlcmd to connect to your database and execute scripts.

When running sqlcmd, it's up to you to supply the right parameters. To see all the possible parameters, type this command:

```
sqlcmd -?
```

Two commonly used sqlcmd parameters are `-S` (which specifies the location of your database server) and `-i` (which supplies a script file with SQL commands that you want to run). For example, the downloadable code samples include a file named `InstPubs.sql` that contains the commands you need to create the pubs database and fill it with sample data. If you're using SQL Server Express LocalDB, and you've installed the code in a folder named `c:\Beginning ASP.NET`, you can run the `InstPubs.sql` script like this:

```
sqlcmd -S (localdb)\v11.0 -i "c:\Beginning ASP.NET\InstPubs.sql"
```

If you're using a full version of SQL Server on the local computer, you don't need to supply the server name at all:

```
sqlcmd -i "c:\Beginning ASP.NET\InstPubs.sql"
```

And if your database is on another computer, you need to supply that computer's name with the `-S` parameter (or just run sqlcmd on that computer).

■ **Note** The parameters you use with sqlcmd are case sensitive. For example, if you use `-s` instead of `-S`, you'll receive an obscure error message informing you that sqlcmd couldn't log in.

Figure 14-4 shows the feedback you'll get when you run `InstPubs.sql` with sqlcmd.

```

D:\Code\Beginning ASP.NET>sqlcmd -S localhost\SQLEXPRESS -i InstPubs.sql
Changed database context to 'master'.
Beginning InstPubs.SQL at 25 Jun 2012 23:53:51:137 ....
Creating pubs database....
Changed database context to 'pubs'.
Now at the create table section ....
Now at the create trigger section ...
Now at the inserts to authors ....
Now at the inserts to publishers ....
Now at the inserts to pub_info ....
Now at the inserts to titles ....
Now at the inserts to titleauthor ....
Now at the inserts to stores ....
Now at the inserts to sales ....
Now at the inserts to roysched ....
Now at the inserts to discounts ....
Now at the inserts to jobs ....
Now at the inserts to employee ....
Now at the create index section ....
Now at the create view section ....
Now at the create procedure section ....
Changed database context to 'master'.
Ending InstPubs.SQL at 25 Jun 2012 23:53:52:017 ....

D:\Code\Beginning ASP.NET>

```

Figure 14-4. Running an SQL script with *sqlcmd.exe*

In this book, you'll occasionally see instructions about using *sqlcmd* to perform some sort of database configuration. However, you can usually achieve the same result (with a bit more clicking) by using the graphical interface in a tool such as SQL Server Management Studio. For example, to install a database by running an SQL script, you simply need to start SQL Server Management Studio, open the SQL file (using the File ► Open ► File command), and then run it (using the Query ► Execute command).

Understanding SQL Basics

When you interact with a data source through ADO.NET, you use Structured Query Language (SQL) to retrieve, modify, and update information. In some cases, ADO.NET will hide some of the details for you or even generate required SQL statements automatically. However, to design an efficient database application with a minimal amount of frustration, you need to understand the basic concepts of SQL.

SQL is a standard data access language used to interact with relational databases. Different databases differ in their support of SQL or add other features, but the core commands used to select, add, and modify data are common. In a database product such as SQL Server, it's possible to use SQL to create fairly sophisticated SQL scripts for stored procedures and triggers (although they have little of the power of a full object-oriented programming language). When working with ADO.NET, however, you'll probably use only the following standard types of SQL statements:

- A Select statement retrieves records.
- An Update statement modifies existing records.
- An Insert statement adds a new record.
- A Delete statement deletes existing records.

If you already have a good understanding of SQL, you can skip the next few sections. Otherwise, read on for a quick tour of SQL fundamentals.

■ **Tip** To learn more about SQL, use one of the SQL tutorials available on the Internet, such as the one at www.w3schools.com/sql. If you're working with SQL Server, you can use its thorough Books Online help to become a database guru.

Running Queries in Visual Studio

If you've never used SQL before, you may want to play around with it and create some sample queries before you start using it in an ASP.NET site. Most database products provide some sort of tool for testing queries. If you're using a full version of SQL Server, you can try SQL Server Management Studio. If you don't want to use an extra tool, you can run your queries by using the Server Explorer window described earlier. Just follow these steps in Visual Studio:

1. Right-click your connection and choose New Query.
2. Choose the table (or tables) you want to use in your query from the Add Table dialog box (as shown in Figure 14-5). Click Add and then click Close.

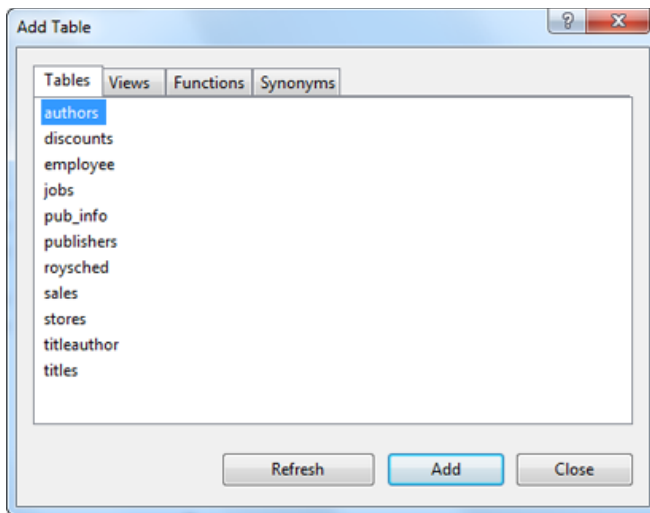


Figure 14-5. Adding tables to a query

3. You'll now see a handy query-building window. You can create your query by adding check marks next to the fields you want, or you can edit the SQL by hand in the lower portion of the window. Best of all, if you edit the SQL directly, you can type in anything—you don't need to stick to the tables you selected in step 2, and you don't need to restrict yourself to Select statements.
4. When you're ready to run the query, choose Query Designer ► Execute SQL from the menu. Assuming your query doesn't have any errors, you'll get one of two results. If you're selecting records, the results will appear at the bottom of the window. If you're deleting or updating records, a message box will appear, informing you of the number of records affected (see Figure 14-6).

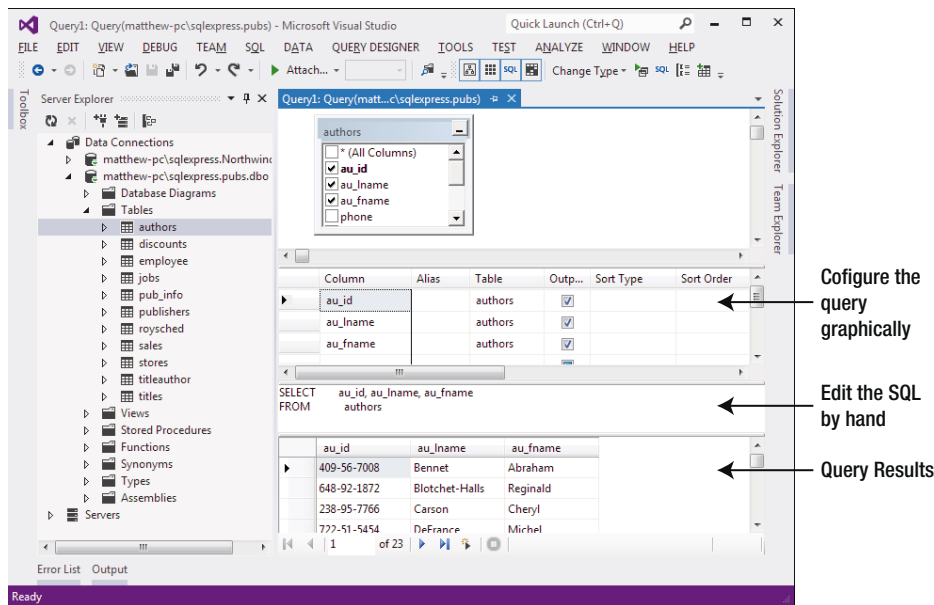


Figure 14-6. Executing a query

Tip When programming with ADO.NET, it always helps to know your database. If you have information on hand about the data types it uses, the stored procedures it provides, and the user account you need to use, you'll be able to work more quickly and with less chance of error.

Using the Select Statement

To retrieve one or more rows of data, you use a Select statement. A basic Select statement has the following structure:

```
SELECT [columns]
FROM [tables]
WHERE [search_condition]
ORDER BY [order_expression ASC | DESC]
```

This format really just scratches the surface of SQL. If you want, you can create more-sophisticated queries that use subgrouping, averaging and totaling, and other options (such as setting a maximum number of returned rows). By performing this work in a query (instead of in your application), you can often create far more efficient applications.

The next few sections present sample Select statements. After each example, a series of bulleted points breaks down the SQL to explain how each part of it works.

A Sample Select Statement

The following is a typical (and rather inefficient) Select statement for the pubs database. It works with the Authors table, which contains a list of authors:

```
SELECT * FROM Authors
```

- The asterisk (*) retrieves all the columns in the table. This isn't the best approach for a large table if you don't need all the information. It increases the amount of data that has to be transferred and can slow down your server.
- The From clause identifies that the Authors table is being used for this statement.
- The statement doesn't have a Where clause. This means all the records will be retrieved from the database, regardless of whether it has 10 or 10 million records. This is a poor design practice, because it often leads to applications that appear to work fine when they're first deployed but gradually slow down as the database grows. In general, you should always include a Where clause to limit the possible number of rows (unless you absolutely need them all). Often queries are limited by a date field (for example, including all orders that were placed in the last three months).
- The statement doesn't have an Order By clause. This is a perfectly acceptable approach, especially if order doesn't matter or you plan to sort the data on your own by using the tools provided in ADO.NET.

An Improved Select Statement

Here's another example that retrieves a list of author names:

```
SELECT au_lname, au_fname FROM Authors WHERE State='CA' ORDER BY au_lname ASC
```

- Only two columns are retrieved (au_lname and au_fname). They correspond to the first and last names of the author.
- A Where clause restricts results to those authors who live in the specified state (California). Note that the Where clause requires apostrophes around the value you want to match, because it's a text value.
- An Order By clause sorts the information alphabetically by the author's last name. The ASC part (for ascending) is optional, because that's the default sort order.

An Alternative Select Statement

Here's one last example:

```
SELECT TOP 100 * FROM Sales ORDER BY ord_date DESC
```

This example uses the Top clause instead of a Where statement. The database rows will be sorted in descending order by order date, and the first 100 matching results will be retrieved. In this case, it's the 100 most recent orders. You could also use this type of statement to find the most expensive items you sell or the best-performing employees.

The Where Clause

In many respects, the Where clause is the most important part of the Select statement. You can find records that match several conditions by using the And keyword, and you can find records that match any one of a series of conditions by using the Or keyword. You can also specify greater-than and less-than comparisons by using the greater-than (>) and less-than (<) operators.

The following is an example with a different table and a more sophisticated Where statement:

```
SELECT * FROM Sales WHERE ord_date < '2000/01/01' AND ord_date > '1987/01/01'
```

This example uses the international date format to compare date values. Although SQL Server supports many date formats, yyyy/mm/dd is recommended to prevent ambiguity.

If you were using Microsoft Access, you would need to use the US date format, mm/dd/yyyy, and replace the apostrophes around the date with the number (#) symbol.

String Matching with the Like Operator

The Like operator allows you to perform partial string matching to filter records in order to find a particular field that starts with, ends with, or contains a certain set of characters. For example, if you want to see all store names that start with *B*, you could use the following statement:

```
SELECT * FROM Stores WHERE stor_name LIKE 'B%'
```

To see a list of all stores *ending* with *S*, you would put the percent sign *before* the *S*, like this:

```
SELECT * FROM Stores WHERE stor_name LIKE '%S'
```

The third way to use the Like operator is to return any records that contain a certain character or sequence of characters. For example, suppose you want to see all stores that have the word *book* somewhere in the name. In this case, you could use an SQL statement like this:

```
SELECT * FROM Stores WHERE stor_name LIKE '%book%'
```

By default, SQL is not case sensitive, so this syntax finds instances of *BOOK*, *book*, or any variation of mixed case.

Finally, you can indicate one of a set of characters, rather than just any character, by listing the allowed characters within square brackets. Here's an example:

```
SELECT * FROM Stores WHERE stor_name LIKE '[abcd]%'
```

This SQL statement will return stores with names starting with *A*, *B*, *C*, or *D*.

Aggregate Queries

The SQL language also defines special *aggregate functions*. Aggregate functions work with a set of values but return only a single value. For example, you can use an aggregate function to count the number of records in a table or to calculate the average price of a product. Table 14-1 lists the most commonly used aggregate functions.

Table 14-1. *SQL Aggregate Functions*

Function	Description
Avg(fieldname)	Calculates the average of all values in a given numeric field
Sum(fieldname)	Calculates the sum of all values in a given numeric field
Min(fieldname) and Max(fieldname)	Finds the minimum or maximum value in a number field
Count(*)	Returns the number of rows in the result set
Count(DISTINCT fieldname)	Returns the number of unique (and non-null) rows in the result set for the specified field

For example, here's a query that returns a single value—the number of records in the Authors table:

```
SELECT COUNT(*) FROM Authors
```

And here's how you could calculate the total quantity of all sales by adding together the qty field in each record:

```
SELECT SUM(qty) FROM Sales
```

Using the SQL Update Statement

The SQL Update statement selects all the records that match a specified search expression and then modifies them all according to an update expression. At its simplest, the Update statement has the following format:

```
UPDATE [table] SET [update_expression] WHERE [search_condition]
```

Typically, you'll use an Update statement to modify a single record. The following example adjusts the phone column in a single author record. It uses the unique author ID to find the correct row.

```
UPDATE Authors SET phone='408 496-2222' WHERE au_id='172-32-1176'
```

This statement returns the number of affected rows, which in this case is 1. Figure 14-7 shows this example in Visual Studio.

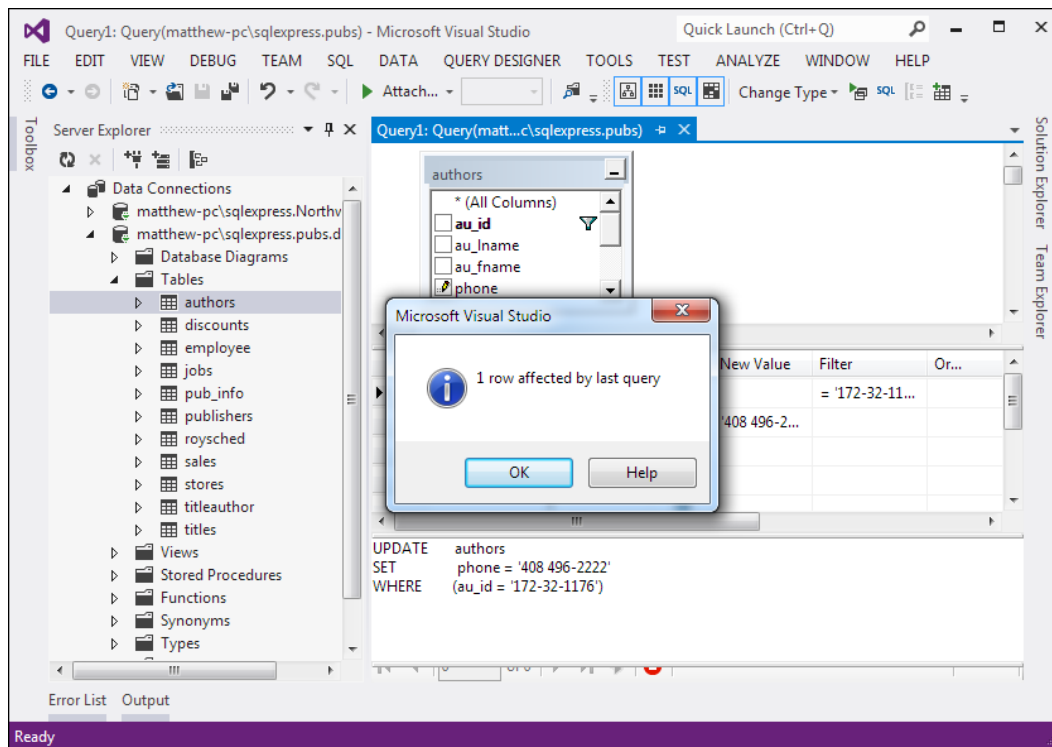


Figure 14-7. Executing an update query in Visual Studio

An Update statement won't display the change that you've made. To do that, you need to request the row by performing another Select statement:

```
SELECT phone FROM Authors WHERE au_id='172-32-1176'
```

As with a Select statement, you can use an Update statement with several criteria:

```
UPDATE Authors SET au_lname='Whiteson', au_fname='John'
WHERE au_lname='White' AND au_fname='Johnson'
```

You can even use the Update statement to update an entire range of matching records. The following example increases the price of every book in the Titles table that was published in 1991 by one dollar:

```
UPDATE Titles SET price=price+1
WHERE pubdate >= '1991/01/01' AND pubdate < '1992/01/01'
```

Using the SQL Insert Statement

The SQL Insert statement adds a new record to a table with the information you specify. It takes the following form:

```
INSERT INTO [table] ([column_list]) VALUES ([value_list])
```

You can provide the information in any order you want, as long as you make sure the list of column names and the list of values correspond exactly:

```
INSERT INTO Authors (au_id, au_lname, au_fname, zip, contract)
VALUES ('998-72-3566', 'Khan', 'John', 84152, 0)
```

This example leaves out some information, such as the city and address, in order to provide a simple example. However, it provides the minimum information that's required to create a new record in the Authors table.

Remember, database tables often have requirements that can prevent you from adding a record unless you fill in all the fields with valid information. Alternatively, some fields may be configured to use a default value if left blank. In the Authors table, some fields are required, and a special format is defined for the ZIP code and author ID.

One feature the Authors table doesn't use is an automatically incrementing identity field. This feature, which is supported in most relational database products, assigns a unique value to a specified field when you perform an insert operation. When you insert a record into a table that has a unique incrementing ID, you shouldn't specify a value for the ID. Instead, allow the database to choose one automatically.

AUTO-INCREMENT FIELDS ARE INDISPENSABLE

If you're designing a database, adding an auto-incrementing identity field to every table is considered good design. This is the fastest, easiest, and least error-prone way to assign a unique identification number to every record. Without an automatically generated identity field, you'll need to go to considerable effort to create and maintain your own unique field. Often programmers fall into the trap of using a data field for a unique identifier, such as a Social Security number (SSN) or a name. This almost always leads to trouble at some inconvenient time far in the future, when you need to add a person who doesn't have an SSN (for example, a foreign national) or you need to account for an SSN or name change (which will cause problems for other related tables, such as a purchase order table that identifies the purchaser by the name or SSN field). A much better approach is to use a unique identifier and have the database engine assign an arbitrary unique number to every row automatically.

If you create a table without a unique identification column, you'll have trouble when you need to select that specific row for deletion or updates. Selecting records based on a text field can also lead to problems if the field contains special embedded characters (such as apostrophes). You'll also find it extremely awkward to create table relationships.

Using the SQL Delete Statement

The Delete statement is even easier to use. It specifies criteria for one or more rows that you want to remove. Be careful: after you delete a row, it's gone for good!

```
DELETE FROM [table] WHERE [search_condition]
```

The following example removes a single matching row from the Authors table:

```
DELETE FROM Authors WHERE au_id='172-32-1176'
```

■ **Note** If you attempt to run this specific Delete statement, you'll run into a database error. The problem is that this author record is linked to one or more records in the TitleAuthor table. The author record can't be removed unless the linked records are deleted first. (After all, it wouldn't make sense to have a book linked to an author that doesn't exist.)

The Delete and Update commands return a single piece of information: the number of affected records. You can examine this value and use it to determine whether the operation is successful or executed as expected.

The rest of this chapter shows how you can combine SQL with the ADO.NET objects to retrieve and manipulate data in your web applications.

Understanding the Data Provider Model

ADO.NET relies on the functionality in a small set of core classes. You can divide these classes into two groups: those that are used to contain and manage data (such as DataSet, DataTable, DataRow, and DataRelation) and those that are used to connect to a specific data source (such as Connection, Command, and DataReader).

The data container classes are completely generic. No matter what data source you use, after you extract the data, it's stored using the same data container: the specialized DataSet class. Think of the DataSet as playing the same role as a collection or an array—it's a package for data. The difference is that the DataSet is customized for relational data, which means it understands concepts such as rows, columns, and table relationships natively.

The second group of classes exists in several flavors. Each set of data interaction classes is called an ADO.NET *data provider*. Data providers are customized so that each one uses the best-performing way of interacting with its data source. For example, the SQL Server data provider is designed to work with SQL Server. Internally, it uses SQL Server's tabular data stream (TDS) protocol for communicating, thus guaranteeing the best possible performance. If you're using Oracle, you can use an Oracle data provider (which is available at <http://tinyurl.com/2wbsjp6>) instead.

Each provider designates its own prefix for naming classes. Thus, the SQL Server provider includes SqlConnection and SqlCommand classes, and the Oracle provider includes OracleConnection and OracleCommand classes. Internally, these classes work quite differently, because they need to connect to different databases by using different low-level protocols. Externally, however, these classes look quite similar and provide an identical set of basic methods because they implement the same common interfaces. This means your application is shielded from the complexity of different standards and can use the SQL Server provider in the same way the Oracle provider uses it. In fact, you can often translate a block of code for interacting with an SQL Server database into a block of Oracle-specific code just by editing the class names in your code.

In this chapter, you'll use the SQL Server data provider. However, the classes you'll use fall into three key namespaces, as outlined in Table 14-2.

Table 14-2. ADO.NET Namespaces for SQL Server Data Access

Namespace	Purpose
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database and execute commands (such as SqlConnection and SqlCommand).
System.Data.SqlTypes	Contains structures for SQL Server-specific data types such as SqlMoney and SqlDateTime. You can use these types to work with SQL Server data types without needing to convert them into the standard .NET equivalents (such as System.Decimal and System.DateTime). These types aren't required, but they do allow you to avoid any potential rounding or conversion problems that could adversely affect data.
System.Data	Contains fundamental classes with the core ADO.NET functionality. These include DataSet and DataRelation, which allow you to manipulate structured relational data. These classes are totally independent of any specific type of database or the way you connect to it.

In the rest of this chapter, you'll consider how to write web page code that uses the classes in these namespaces. First you'll consider the most straightforward approach—direct data access. Then you'll consider disconnected data access, which allows you to retrieve data in the `DataSet` and cache it for longer periods of time. Both approaches complement each other, and in many web applications you'll use a combination of the two.

Using Direct Data Access

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

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

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

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

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

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

To add or update information, follow these steps:

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

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

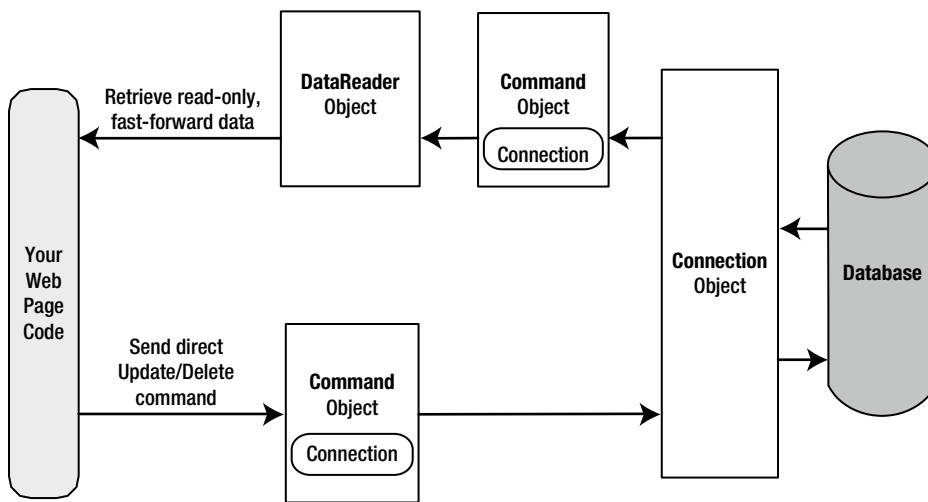


Figure 14-8. Direct data access with ADO.NET

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

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

Creating a Connection

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

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

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

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

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

■ **Note** Ordinarily, the C# compiler assumes that every backslash designates the start of a special character sequence. This can cause issues when you use connection strings in code. There are two ways to deal with this problem. You can use two backslashes instead of one, as in `(localdb)\\v11.0`. Or you can put the `@` character before the entire string, as in `@(localdb)\\v11.0`. However, if you define the connection string in a configuration file, as described in the next section, you don't need to take either of these steps. That's because you're no longer dealing with pure C# code, so a single backslash is fine.

Exploring the Connection String

The connection string is actually a series of distinct pieces of information separated by semicolons (;). Each separate piece of information is known as a *connection string property*.

The following list describes some of the most commonly used connection string properties, including the three properties used in the preceding example:

Data source: This indicates the name of the server where the data source is located. If the server is on the same computer that hosts the ASP.NET site, **localhost** is sufficient. But if you're using SQL Server Express, the connection string changes. For SQL Server Express LocalDB (the test version included with Visual Studio), the data source is **(localdb)\\v11.0**. For the full version of SQL Server Express, which you might use in a deployed application, it's **localhost\\SQLEXPRESS** (or **ServerName\\SQLEXPRESS** if the database is running on another computer). You'll also see this written with a period, as **.\\SQLEXPRESS**, which is equivalent.

Initial catalog: This is the name of the database that this connection will be accessing. It's only the "initial" database, because you can change it later by using the `Connection.ChangeDatabase()` method.

Integrated security: This indicates that you want to connect to SQL Server by using the Windows user account that's running the web page code, provided you supply a value of SSPI (which stands for *Security Support Provider Interface*). Alternatively, you can supply a user ID and password that's defined in the database for SQL Server authentication, although this method is less secure and generally discouraged.

ConnectionTimeout: This determines the number of seconds your code will wait before generating an error if it cannot establish a database connection. Our example connection string doesn't set the `ConnectionTimeout`, so the default of 15 seconds is used. You can use 0 to specify no limit, but this is a bad idea. This means that, theoretically, the code could be held up indefinitely while it attempts to contact the server.

You can set some other, lesser-used options for a connection string. For more information, refer to the Visual Studio Help. Look under the appropriate Connection class (such as `SqlConnection` or `OleDbConnection`), because there are subtle differences in connection string properties for each type of Connection class.

Using Windows Authentication

The previous example uses *integrated Windows authentication*, which is the default security standard for new SQL Server installations. You can also use *SQL Server authentication*. In this case, you will explicitly place the user ID and password information in the connection string. However, SQL Server authentication is disabled by default in modern versions of SQL Server, because it's not considered to be as secure.

Here's the lowdown on both types of authentication:

- With SQL Server authentication, SQL Server maintains its own user account information in the database. It uses this information to determine whether you are allowed to access specific parts of a database.
- With integrated Windows authentication, SQL Server automatically uses the Windows account information for the currently logged-in process. In the database, it stores information about what database privileges each user should have.

Tip You can set the type of authentication that your SQL Server uses by using a tool such as SQL Server Management Studio. Just right-click your server in the tree and select Properties. Choose the Security tab to change the type of authentication. You can choose either Windows Only (for the tightest security) or SQL Server and Windows, which allows both Windows authentication and SQL Server authentication. This option is also known as *mixed-mode authentication*.

For Windows authentication to work, the currently logged-on Windows user must have the required authorization to access the SQL database. This isn't a problem while you test your websites, because Visual Studio launches your web applications by using your user account. However, when you deploy your application to a web server running IIS, you might run into trouble. In this situation, all ASP.NET code is run by a more limited user account that might not have the rights to access the database. Although the exact user depends on your version of IIS (see the discussion in Chapter 26), the best approach is usually to grant access to the IIS_IUSRS group.

Making User Instance Connections

Every database server stores a master list of all the databases that you've installed on it. This list includes the name of each database and the location of the files that hold the data. When you create a database (for example, by running a script or using a management tool), the information about that database is added to the master list. When you connect to the database, you specify the database name by using the Initial Catalog value in the connection string.

Note If you haven't made any changes to your database configuration, SQL Server will quietly tuck the files for newly created databases into a directory such as c:\Program Files\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\Data (although the exact path depends on the version of SQL Server you're using). Each database has at least two files—an .mdf file with the actual data and an .ldf file that stores the database log. Of course, database professionals have a variety of techniques and tricks for managing database storage, and can easily store databases in different locations, create multiple data files, and so on. The important detail to realize is that ordinarily your database files are stored by your database server, and they aren't a part of your web application directory.

Interestingly, SQL Server Express has a feature that lets you bypass the master list and connect directly to any database file, even if it's not in the master list of databases. This feature is called *user instances*. Oddly enough, this feature isn't available in the full edition of SQL Server.

To use this feature, you need to set the User Instances value to True (in the connection string) and supply the file name of the database you want to connect to with the AttachDBFilename value. You don't supply an Initial Catalog value.

Here's an example connection string that uses this approach:

```
myConnection.ConnectionString = @"Data Source=(localdb)\v11.0;" +
    @"User Instance=True;AttachDBFilename=|DataDirectory|\Northwind.mdf;" +
    "Integrated Security=True";
```

There's another trick here. The file name starts with |DataDirectory|. This automatically points to the App_Data folder inside your web application directory. This way, you don't need to supply a full file path, which might not remain valid when you move the web application to a web server. Instead, ADO.NET will always look in the App_Data directory for a file named Northwind.mdf.

User instances are a handy feature if you have a web server that hosts many different web applications that use databases and these databases are frequently being added and removed. However, because the database isn't in the master list, you won't see it in any administrative tools (although most administrative tools will still let you connect to it manually, by pointing out the right file location). But remember, this quirky but interesting feature is available in SQL Server Express only—you won't find it in the full version of SQL Server.

VISUAL STUDIO'S SUPPORT FOR USER INSTANCE DATABASES

Visual Studio provides two handy features that make it easier to work with databases in the App_Data folder.

First, Visual Studio gives you a nearly effortless way to create new databases. Simply choose Website ► Add New Item. Then pick SQL Server Database from the list of templates, choose a file name for your database, and click OK. The .mdf and .ldf files for the new database will be placed in the App_Data folder, and you'll see them in the Solution Explorer. Initially, they'll be blank, so you'll need to add the tables you want. (The easiest way to do this is to right-click the Tables group in the Server Explorer and choose Add Table.)

Visual Studio also simplifies your life with its automatic Server Explorer support. When you open a web application, Visual Studio automatically adds a data connection to the Server Explorer window for each database that it finds in the App_Data folder. To jump to a specific data connection in a hurry, just double-click the .mdf file for the database in the Solution Explorer.

Using the Server Explorer, you can create tables, edit data, and execute commands, all without leaving the comfort of Visual Studio. (For more information about executing commands with the Server Explorer, refer to the "Understanding SQL Basics" section earlier in this chapter.)

Storing the Connection String

Typically, all the database code in your application will use the same connection string. Therefore, it usually makes the most sense to store a connection string in a class member variable or, even better, a configuration file.

You can also create a Connection object and supply the connection string in one step by using a dedicated constructor:

```
SqlConnection myConnection = new SqlConnection(connectionString);
// myConnection.ConnectionString is now set to connectionString.
```

You don't need to hard-code a connection string. The <connectionStrings> section of the web.config file is a handy place to store your connection strings. Here's an example:

```
<configuration>
  <connectionStrings>
    <add name="Pubs" connectionString=
"Data Source=localhost;Initial Catalog=Pubs;Integrated Security=SSPI"/>
  </connectionStrings>
  ...
</configuration>
```

You can then retrieve your connection string by name. First import the System.Web.Configuration namespace. Then you can use code like this:

```
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
```

This approach helps ensure that all your web pages are using the same connection string. It also makes it easy for you to change the connection string for an application, without needing to edit the code in multiple pages. The examples in this chapter all store their connection strings in the web.config file in this way.

Making the Connection

After you've created your connection (as described in the previous section), you're ready to use it.

Before you can perform any database operations, you need to explicitly open your connection:

```
myConnection.Open();
```

To verify that you have successfully connected to the database, you can try displaying some basic connection information. The following example writes some basic information to a Label control named lblInfo (see Figure 14-9).

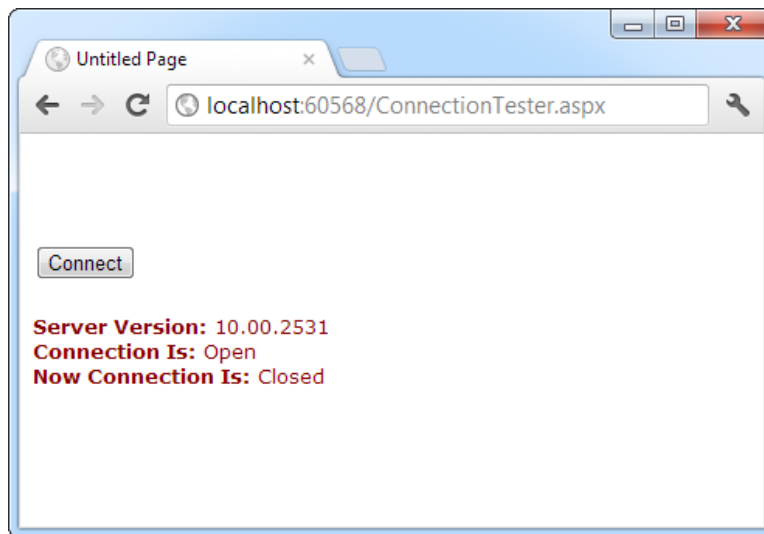


Figure 14-9. Testing your connection

Here's the code with basic error handling:

```
// Define the ADO.NET Connection object.
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection myConnection = new SqlConnection(connectionString);

try
{
    // Try to open the connection.
    myConnection.Open();
    lblInfo.Text = "<b>Server Version:</b> " + myConnection.ServerVersion;
    lblInfo.Text += "<br /><b>Connection Is:</b> " +
        myConnection.State.ToString();
}
catch (Exception err)
{
    // Handle an error by displaying the information.
    lblInfo.Text = "Error reading the database. ";
    lblInfo.Text += err.Message;
}
finally
{
    // Either way, make sure the connection is properly closed.
    // (Even if the connection wasn't opened successfully,
    // calling Close() won't cause an error.)
    myConnection.Close();
    lblInfo.Text += "<br /><b>Now Connection Is:</b> ";
    lblInfo.Text += myConnection.State.ToString();
}
```

After you use the `Open()` method, you have a live connection to your database. One of the most fundamental principles of data access code is that you should reduce the amount of time you hold a connection open as much as possible. Imagine that as soon as you open the connection, you have a live, ticking time bomb. You need to get in, retrieve your data, and throw the connection away as quickly as possible in order to ensure that your site runs efficiently.

Closing a connection is just as easy, as shown here:

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

Another approach is to use the `using` statement. The `using` statement declares that you are using a disposable object for a short period of time. As soon as you finish using that object and the `using` block ends, the common language runtime will release it immediately by calling the `Dispose()` method. Here's the basic structure of the `using` block:

```
using (object)
{
    ...
}
```

It just so happens that calling the `Dispose()` method of a connection object is equivalent to calling `Close()` and then discarding the connection object from memory. That means you can shorten your database code with the help of a `using` block. The best part is that you don't need to write a `finally` block—the `using` statement releases the object you're using even if you exit the block as the result of an unhandled exception.

Here's how you could rewrite the earlier example with a using block:

```
// Define the ADO.NET Connection object.
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection myConnection = new SqlConnection(connectionString);

try
{
    using (myConnection)
    {
        // Try to open the connection.
        myConnection.Open();
        lblInfo.Text = "<b>Server Version:</b> " + myConnection.ServerVersion;
        lblInfo.Text += "<br /><b>Connection Is:</b> " +
            myConnection.State.ToString();
    }
}
catch (Exception err)
{
    // Handle an error by displaying the information.
    lblInfo.Text = "Error reading the database. ";
    lblInfo.Text += err.Message;
}

lblInfo.Text += "<br /><b>Now Connection Is:</b> ";
lblInfo.Text += myConnection.State.ToString();
```

There's one difference in the way this code is implemented as compared to the previous example. The error-handling code wraps the using block. As a result, if an error occurs, the database connection is closed first, and *then* the exception-handling code is triggered. In the first example, the error-handling code responded first, and then the finally block closed the connection afterward. Obviously, this rewrite is a bit better, as it's always good to close database connections as soon as possible.

Using the Select Command

The Connection object provides a few basic properties that supply information about the connection, but that's about all. To actually retrieve data, you need a few more ingredients:

- An SQL statement that selects the information you want
- A Command object that executes the SQL statement
- A DataReader or DataSet object to access the retrieved records

Command objects represent SQL statements. To use a Command object, you define it, specify the SQL statement you want to use, specify an available connection, and execute the command. To ensure good database performance, you should open your connection just before you execute your command and close it as soon as the command is finished.

You can use one of the earlier SQL statements, as shown here:

```
SqlCommand myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = "SELECT * FROM Authors ORDER BY au_lname ";
```


Or you can use the constructor as a shortcut:

```
SqlCommand myCommand = new SqlCommand(
    "SELECT * FROM Authors ORDER BY au_lname ", myConnection);
```

■ **Note** It's also a good idea to dispose of the Command object when you're finished, although it isn't as critical as closing the Connection object.

Using the DataReader

After you've defined your command, you need to decide how you want to use it. The simplest approach is to use a DataReader, which allows you to quickly retrieve all your results. The DataReader uses a live connection and should be used quickly and then closed. The DataReader is also extremely simple. It supports fast-forward-only read-only access to your results, which is generally all you need when retrieving information. Because of the DataReader's optimized nature, it provides better performance than the DataSet. It should always be your first choice for direct data access.

Before you can use a DataReader, make sure you've opened the connection:

```
myConnection.Open();
```

To create a DataReader, you use the ExecuteReader() method of the command object, as shown here:

```
// You don't need the new keyword, as the Command will create the DataReader.
SqlDataReader myReader;
myReader = myCommand.ExecuteReader();
```

These two lines of code define a variable for a DataReader and then create it by executing the command. After you have the reader, you retrieve a single row at a time by using the Read() method:

```
myReader.Read(); // The first row in the result set is now available.
```

You can then access the values in the current row by using the corresponding field names. The following example adds an item to a list box with the first name and last name for the current row:

```
lstNames.Items.Add(myReader["au_lname"] + ", " + myReader["au_fname"]);
```

To move to the next row, use the Read() method again. If this method returns True, a row of information has been successfully retrieved. If it returns False, you've attempted to read past the end of your result set. There is no way to move backward to a previous row.

As soon as you've finished reading all the results you need, close the DataReader and Connection:

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

Putting It All Together

The next example demonstrates how you can use all the ADO.NET ingredients together to create a simple application that retrieves information from the Authors table. You can select an author record by last name by using a drop-down list box, as shown in Figure 14-10.

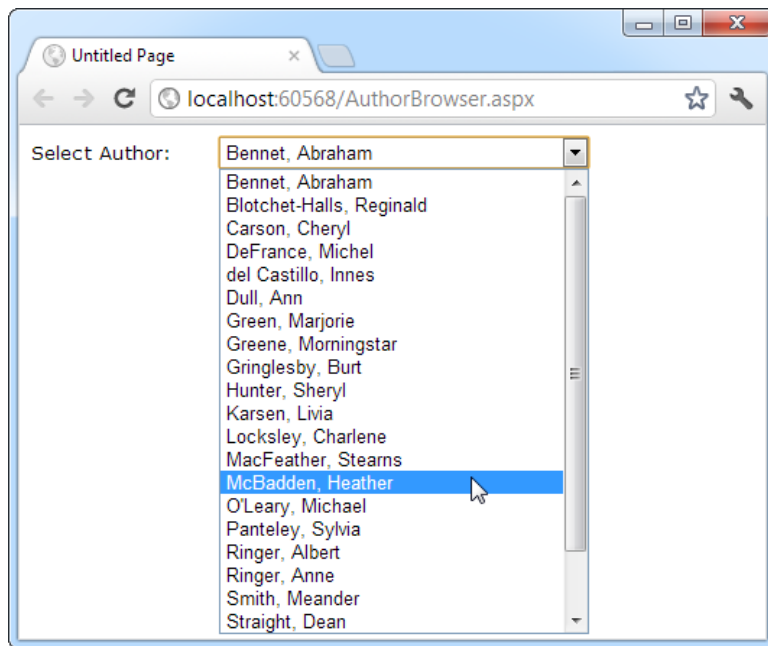


Figure 14-10. Selecting an author

The full record is then retrieved and displayed in a simple label, as shown in Figure 14-11.

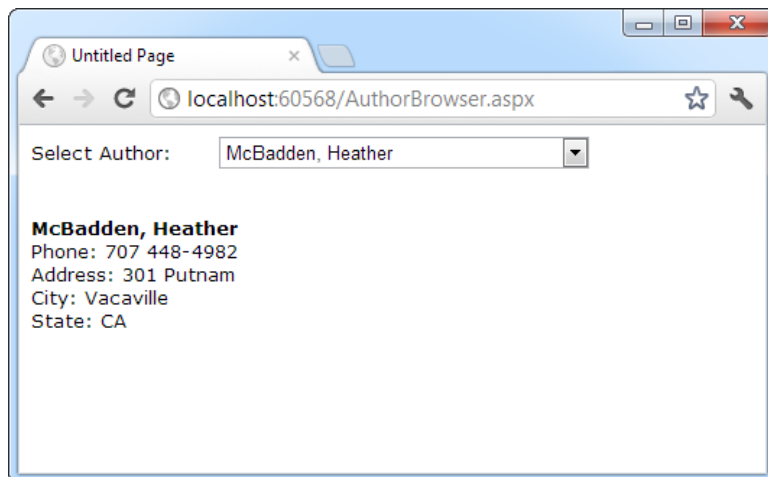


Figure 14-11. Author information

Filling the List Box

To start, the connection string is defined as a private variable for the page class and retrieved from the connection string:

```
private string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
```

The list box is filled when the Page.Load event occurs. Because the list box is set to persist its view state information, this information needs to be retrieved only once—the first time the page is displayed. It will be ignored on all postbacks.

Here's the code that fills the list from the database:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        FillAuthorList();
    }
}

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

    // Define the Select statement.
    // Three pieces of information are needed: the unique id
    // and the first and last name.
    string selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";

    // Define the ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

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

        // For each item, add the author name to the displayed
        // list box text, and store the unique ID in the Value property.
        while (reader.Read())
        {
            ListItem newItem = new ListItem();
            newItem.Text = reader["au_lname"] + ", " + reader["au_fname"];
            newItem.Value = reader["au_id"].ToString();
            lstAuthor.Items.Add(newItem);
        }
        reader.Close();
    }
    catch (Exception err)
```

```

    {
        lblResults.Text = "Error reading list of names. ";
        lblResults.Text += err.Message;
    }
    finally
    {
        con.Close();
    }
}

```

This example looks more sophisticated than the previous bite-sized snippets in this chapter, but it really doesn't introduce anything new. It uses the standard Connection, Command, and DataReader objects. The Connection is opened inside an error-handling block so your page can handle any unexpected errors and provide information. A finally block makes sure the connection is properly closed, even if an error occurs.

The actual code for reading the data uses a loop. With each pass, the Read() method is called to get another row of information. When the reader has read all the available information, this method will return false, the loop condition will evaluate to false, and the loop will end gracefully.

The unique ID (the value in the au_id field) is stored in the Value property of the list box for reference later. This is a crucial ingredient that is needed to allow the corresponding record to be queried again. If you tried to build a query using the author's name, you would need to worry about authors with the same name. You would also have the additional headache of invalid characters (such as the apostrophe in O'Leary) that would invalidate your SQL statement.

Retrieving the Record

The record is retrieved as soon as the user changes the selection in the list box. To make this possible, the AutoPostBack property of the list box is set to true so that its change events are detected automatically.

```

protected void lstAuthor_SelectedIndexChanged(Object sender, EventArgs e)
{
    // Create a Select statement that searches for a record
    // matching the specific author ID from the Value property.
    string selectSQL;
    selectSQL = "SELECT * FROM Authors ";
    selectSQL += "WHERE au_id='" + lstAuthor.SelectedItem.Value + "'";

    // Define the ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

    // Try to open database and read information.
    try
    {
        con.Open();
        reader = cmd.ExecuteReader();
        reader.Read();

        // Build a string with the record information,
        // and display that in a label.
        StringBuilder sb = new StringBuilder();
        sb.Append("<b>");
    }
}

```

```

        sb.Append(reader["au_lname"]);
        sb.Append(", ");
        sb.Append(reader["au_fname"]);
        sb.Append("</b><br />");
        sb.Append("Phone: ");
        sb.Append(reader["phone"]);
        sb.Append("<br />");
        sb.Append("Address: ");
        sb.Append(reader["address"]);
        sb.Append("<br />");
        sb.Append("City: ");
        sb.Append(reader["city"]);
        sb.Append("<br />");
        sb.Append("State: ");
        sb.Append(reader["state"]);
        sb.Append("<br />");
        lblResults.Text = sb.ToString();

        reader.Close();
    }
    catch (Exception err)
    {
        lblResults.Text = "Error getting author. ";
        lblResults.Text += err.Message;
    }
    finally
    {
        con.Close();
    }
}

```

The process is similar to the procedure used to retrieve the last names. There are only a couple of differences:

- The code dynamically creates an SQL statement based on the selected item in the drop-down list box. It uses the Value property of the selected item, which stores the unique identifier. This is a common (and useful) technique.
- Only one record is read. The code assumes that only one author has the matching au_id, which is reasonable because this field is unique.

■ **Note** This example shows how ADO.NET works to retrieve a simple result set. Of course, ADO.NET also provides handy controls that go beyond this generic level and let you provide full-featured grids with sorting and editing. These controls are described in Chapter 15 and Chapter 16. For now, you should concentrate on understanding the fundamentals of ADO.NET and how it works with data.

Updating Data

Now that you understand how to retrieve data, it isn't much more complicated to perform simple insert, update, and delete operations. Once again, you use the Command object, but this time you don't need a DataReader

because no results will be retrieved. You also don't use an SQL Select command. Instead, you use one of three new SQL commands: Update, Insert, or Delete.

To execute an Update, Insert, or Delete statement, you need to create a Command object. You can then execute the command with the `ExecuteNonQuery()` method. This method returns the number of rows that were affected, which allows you to check your assumptions. For example, if you attempt to update or delete a record and are informed that no records were affected, you probably have an error in your Where clause that is preventing any records from being selected. (If, on the other hand, your SQL command has a syntax error or attempts to retrieve information from a nonexistent table, an exception will occur.)

Displaying Values in Text Boxes

Before you can update and insert records, you need to make a change to the previous example. Instead of displaying the field values in a single, fixed label, you need to show each detail in a separate text box.

Figure 14-12 shows the revamped page. It includes two new buttons that allow you to update the record (Update) or delete it (Delete), and two more that allow you to begin creating a new record (Create New) and then insert it (Insert New).

Untitled Page x

localhost:60568/AuthorManager.aspx

Select Author: Blotchet-Halls, Reginald [v] [Update] [Delete]

Or: [Create New] [Insert New]

Unique ID: 648-92-1872 (required: ###-##-#### form)

First Name: Reginald

Last Name: Blotchet-Halls

Phone: 503 745-6402

Address: 55 Hillsdale Bl.

City: Corvallis

State: OR

Zip Code: 97330 (required: any five digits)

Contract: ☒

Figure 14-12. A more advanced author manager

The record selection code is identical from an ADO.NET perspective, but it now uses the individual text boxes:

```
protected void lstAuthor_SelectedIndexChanged(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string selectSQL;
    selectSQL = "SELECT * FROM Authors ";
    selectSQL += "WHERE au_id='" + lstAuthor.SelectedItem.Value + "'";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

    // Try to open database and read information.
    try
    {
        con.Open();
        reader = cmd.ExecuteReader();
        reader.Read();

        // Fill the controls.
        txtID.Text = reader["au_id"].ToString();
        txtFirstName.Text = reader["au_fname"].ToString();
        txtLastName.Text = reader["au_lname"].ToString();
        txtPhone.Text = reader["phone"].ToString();
        txtAddress.Text = reader["address"].ToString();
        txtCity.Text = reader["city"].ToString();
        txtState.Text = reader["state"].ToString();
        txtZip.Text = reader["zip"].ToString();
        chkContract.Checked = (bool)reader["contract"];
        reader.Close();
        lblStatus.Text = "";
    }
    catch (Exception err)
    {
        lblStatus.Text = "Error getting author. ";
        lblStatus.Text += err.Message;
    }
    finally
    {
        con.Close();
    }
}
```

To see the full code, refer to the online samples for this chapter. If you play with the example at length, you'll notice that it lacks a few niceties that would be needed in a professional website. For example, when creating a new record, the name of the last selected user is still visible, and the Update and Delete buttons are still active, which can lead to confusion or errors. A more sophisticated user interface could prevent these problems by disabling inapplicable controls (perhaps by grouping them in a Panel control) or by using separate pages. In this case, however, the page is useful as a quick way to test some basic data access code.

Adding a Record

To start adding a new record, click Create New to clear all the text boxes. Technically, this step isn't required, but it simplifies the user's life:

```
protected void cmdNew_Click(Object sender, EventArgs e)
{
    txtID.Text = "";
    txtFirstName.Text = "";
    txtLastName.Text = "";
    txtPhone.Text = "";
    txtAddress.Text = "";
    txtCity.Text = "";
    txtState.Text = "";
    txtZip.Text = "";
    chkContract.Checked = false;

    lblStatus.Text = "Click Insert New to add the completed record.";
}
```

The Insert New button triggers the ADO.NET code that uses a dynamically generated Insert statement to insert the finished record:

```
protected void cmdInsert_Click(Object sender, EventArgs e)
{
    // Perform user-defined checks.
    // Alternatively, you could use RequiredFieldValidator controls.
    if (txtID.Text == "" || txtFirstName.Text == "" || txtLastName.Text == "")
    {
        lblStatus.Text = "Records require an ID, first name, and last name.";
        return;
    }

    // Define ADO.NET objects.
    string insertSQL;
    insertSQL = "INSERT INTO Authors (";
    insertSQL += "au_id, au_fname, au_lname, ";
    insertSQL += "phone, address, city, state, zip, contract) ";
    insertSQL += "VALUES ('";
    insertSQL += txtID.Text + "', '";
    insertSQL += txtFirstName.Text + "', '";
    insertSQL += txtLastName.Text + "', '";
    insertSQL += txtPhone.Text + "', '";
    insertSQL += txtAddress.Text + "', '";
    insertSQL += txtCity.Text + "', '";
    insertSQL += txtState.Text + "', '";
    insertSQL += txtZip.Text + "', '";
    insertSQL += Convert.ToInt16(chkContract.Checked) + "')";
```



```

SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(insertSQL, con);

// Try to open the database and execute the update.
int added = 0;
try
{
    con.Open();
    added = cmd.ExecuteNonQuery();
    lblStatus.Text = added.ToString() + " records inserted.";
}
catch (Exception err)
{
    lblStatus.Text = "Error inserting record. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the insert succeeded, refresh the author list.
if (added > 0)
{
    FillAuthorList();
}
}

```

If the insert fails, the problem will be reported to the user in a rather unfriendly way (see Figure 14-13). This is typically the result of not specifying valid values. If the insert operation is successful, the page is updated with the new author list.

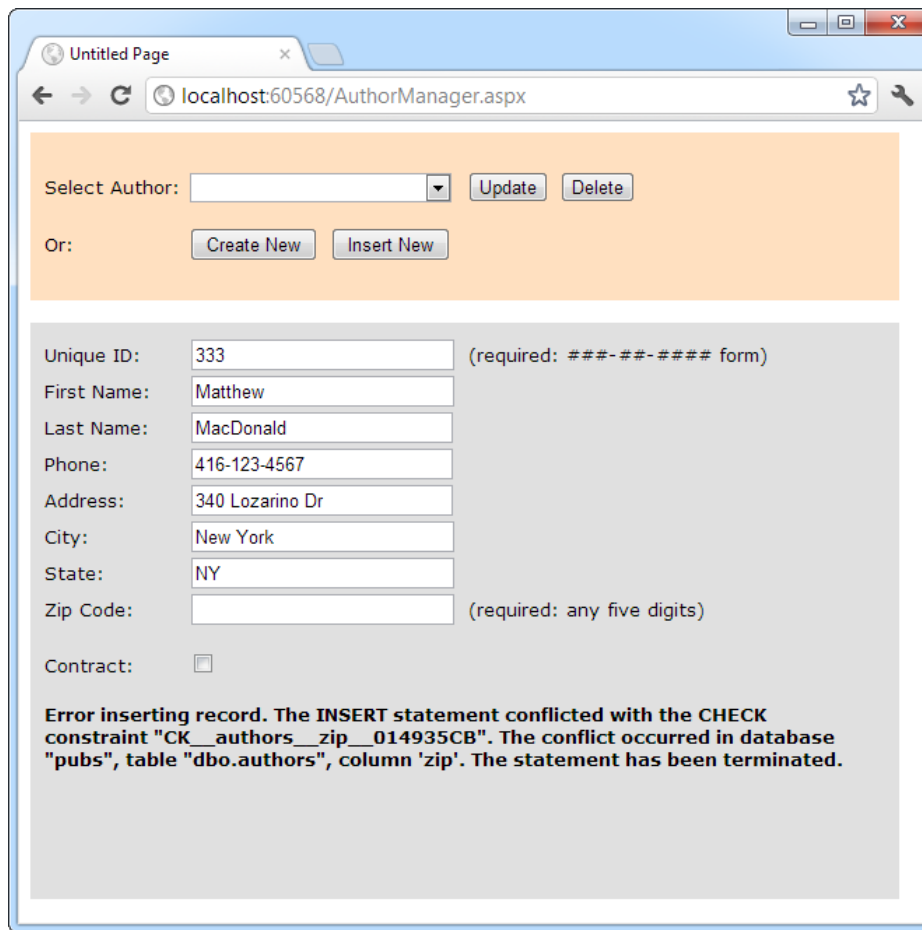


Figure 14-13. A failed insertion

Note In a more polished application, you would use validators (as shown in Chapter 9) and provide more-useful error messages. You should never display the detailed database error information shown in Figure 14-13, because it could give valuable information to malicious users.

Creating More Robust Commands

The previous example performed its database work by using a dynamically pasted-together SQL string. This off-the-cuff approach is great for quickly coding database logic and is easy to understand. However, it has two potentially serious drawbacks:

- Users may accidentally enter characters that will affect your SQL statement. For example, if a value contains an apostrophe ('), the pasted-together SQL string will no longer be valid.

- Users might *deliberately* enter characters that will affect your SQL statement. Examples include using the single apostrophe to close a value prematurely and then following the value with additional SQL code.

The second of these is known as an *SQL injection attack*, which facilitates an amazingly wide range of exploits. Crafty users can use SQL injection attacks to do anything, from returning additional results (such as the orders placed by other customers), to even executing additional SQL statements (such as deleting every record in another table in the same database). In fact, SQL Server includes a special system stored procedure that allows users to execute arbitrary programs on the computer, so this vulnerability can be extremely serious.

You could address these problems by carefully validating the supplied input and checking for dangerous characters such as apostrophes. One approach is to sanitize your input by doubling all apostrophes in the user input (in other words, replace ' with '). Here's an example:

```
string authorID = txtID.Text.Replace("'", "'');
```

A much more robust and convenient approach is to use a *parameterized command*. A parameterized command is one that replaces hard-coded values with placeholders. The placeholders are then added separately and automatically encoded.

For example, this SQL statement:

```
SELECT * FROM Customers WHERE CustomerID = 'ALFKI'
```

would become this:

```
SELECT * FROM Customers WHERE CustomerID = @CustomerID
```

You then need to add a Parameter object for each parameter in the Command.Parameters collection.

The following example rewrites the insert code of the author manager example with a parameterized command:

```
protected void cmdInsert_Click(Object sender, EventArgs e)
{
    // Perform user-defined checks.
    if (txtID.Text == "" || txtFirstName.Text == "" || txtLastName.Text == "")
    {
        lblStatus.Text = "Records require an ID, first name, and last name.";
        return;
    }

    // Define ADO.NET objects.
    string insertSQL;
    insertSQL = "INSERT INTO Authors (";
    insertSQL += "au_id, au_fname, au_lname, ";
    insertSQL += "phone, address, city, state, zip, contract) ";
    insertSQL += "VALUES (";
    insertSQL += "@au_id, @au_fname, @au_lname, ";
    insertSQL += "@phone, @address, @city, @state, @zip, @contract)";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(insertSQL, con);

    // Add the parameters.
    cmd.Parameters.AddWithValue("@au_id", txtID.Text);
    cmd.Parameters.AddWithValue("@au_fname", txtFirstName.Text);
    cmd.Parameters.AddWithValue("@au_lname", txtLastName.Text);
```

```

cmd.Parameters.AddWithValue("@phone", txtPhone.Text);
cmd.Parameters.AddWithValue("@address", txtAddress.Text);
cmd.Parameters.AddWithValue("@city", txtCity.Text);
cmd.Parameters.AddWithValue("@state", txtState.Text);
cmd.Parameters.AddWithValue("@zip", txtZip.Text);
cmd.Parameters.AddWithValue("@contract", chkContract.Checked);

// Try to open the database and execute the update.
int added = 0;
try
{
    con.Open();
    added = cmd.ExecuteNonQuery();
    lblStatus.Text = added.ToString() + " record inserted.";
}
catch (Exception err)
{
    lblStatus.Text = "Error inserting record. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the insert succeeded, refresh the author list.
if (added > 0)
{
    FillAuthorList();
}
}

```

Now that the values have been moved out of the SQL command and to the Parameters collection, there's no way that a misplaced apostrophe or scrap of SQL can cause a problem.

■ **Caution** For basic security, *always* use parameterized commands. Many of the most infamous attacks on e-commerce websites weren't fueled by hard-core hacker knowledge but used simple SQL injection to modify values in web pages or query strings.

Updating a Record

When the user clicks the Update button, the information in the text boxes is applied to the database as follows:

```

protected void cmdUpdate_Click(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string updateSQL;
    updateSQL = "UPDATE Authors SET ";
}

```

```

updateSQL += "au_fname=@au_fname, au_lname=@au_lname, ";
updateSQL += "phone=@phone, address=@address, city=@city, state=@state, ";
updateSQL += "zip=@zip, contract=@contract ";
updateSQL += "WHERE au_id=@au_id_original";

SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(updateSQL, con);

// Add the parameters.
cmd.Parameters.AddWithValue("@au_fname", txtFirstName.Text);
cmd.Parameters.AddWithValue("@au_lname", txtLastName.Text);
cmd.Parameters.AddWithValue("@phone", txtPhone.Text);
cmd.Parameters.AddWithValue("@address", txtAddress.Text);
cmd.Parameters.AddWithValue("@city", txtCity.Text);
cmd.Parameters.AddWithValue("@state", txtState.Text);
cmd.Parameters.AddWithValue("@zip", txtZip.Text);
cmd.Parameters.AddWithValue("@contract", chkContract.Checked);
cmd.Parameters.AddWithValue("@au_id_original",
    lstAuthor.SelectedItem.Value);

// Try to open database and execute the update.
int updated = 0;
try
{
    con.Open();
    updated = cmd.ExecuteNonQuery();
    lblStatus.Text = updated.ToString() + " record updated.";
}
catch (Exception err)
{
    lblStatus.Text = "Error updating author. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the update succeeded, refresh the author list.
if (updated > 0)
{
    FillAuthorList();
}
}

```

Deleting a Record

When the user clicks the Delete button, the author information is removed from the database. The number of affected records is examined, and if the delete operation was successful, the `FillAuthorList()` function is called to refresh the page.

```
protected void cmdDelete_Click(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string deleteSQL;
    deleteSQL = "DELETE FROM Authors ";
    deleteSQL += "WHERE au_id=@au_id";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(deleteSQL, con);
    cmd.Parameters.AddWithValue("@au_id ", lstAuthor.SelectedItem.Value);

    // Try to open the database and delete the record.
    int deleted = 0;
    try
    {
        con.Open();
        deleted = cmd.ExecuteNonQuery();
    }
    catch (Exception err)
    {
        lblStatus.Text = "Error deleting author. ";
        lblStatus.Text += err.Message;
    }
    finally
    {
        con.Close();
    }

    // If the delete succeeded, refresh the author list.
    if (deleted > 0)
    {
        FillAuthorList();
    }
}
```

Interestingly, delete operations rarely succeed with the records in the pubs database, because they have corresponding child records linked in another table of the pubs database. Specifically, each author can have one or more related book titles. Unless the author's records are removed from the `TitleAuthor` table first, the author cannot be deleted. Because of the careful error handling used in the previous example, this problem is faithfully reported in your application (see Figure 14-14) and doesn't cause any real problems.

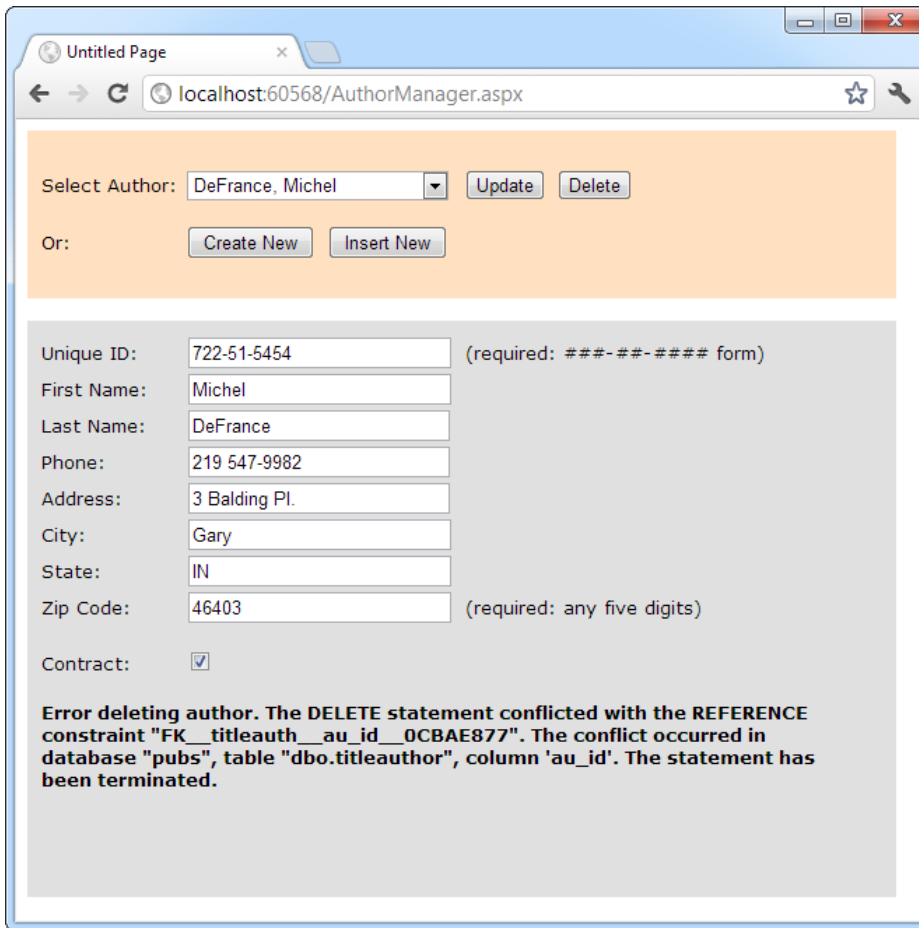


Figure 14-14. A failed delete attempt

To get around this limitation, you can use the Create New and Insert New buttons to add a new record and then delete this record. Because this new record won't be linked to any other records, its deletion will be allowed.

■ **Note** If you have a real-world application that needs to delete records, and these records might have linked records in a child table, there are several possible solutions. One slightly dangerous option is to configure the database to use *cascading deletes* to automatically wipe out linked records. Another option is to do the deleting yourself, with additional ADO.NET code. But the best choice is usually not to delete the record at all (after all, you may need it for tracking and reporting later). Instead, use a bit column to keep track of records that shouldn't be displayed, such as a Discontinued column in a Products table or a Removed column in the Authors table. You can then add a condition to your Select query so that it doesn't retrieve these records (as in `SELECT * FROM Products WHERE Discontinued=0`).

Using Disconnected Data Access

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

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

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

UPDATING DISCONNECTED DATA

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

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

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

Selecting Disconnected Data

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

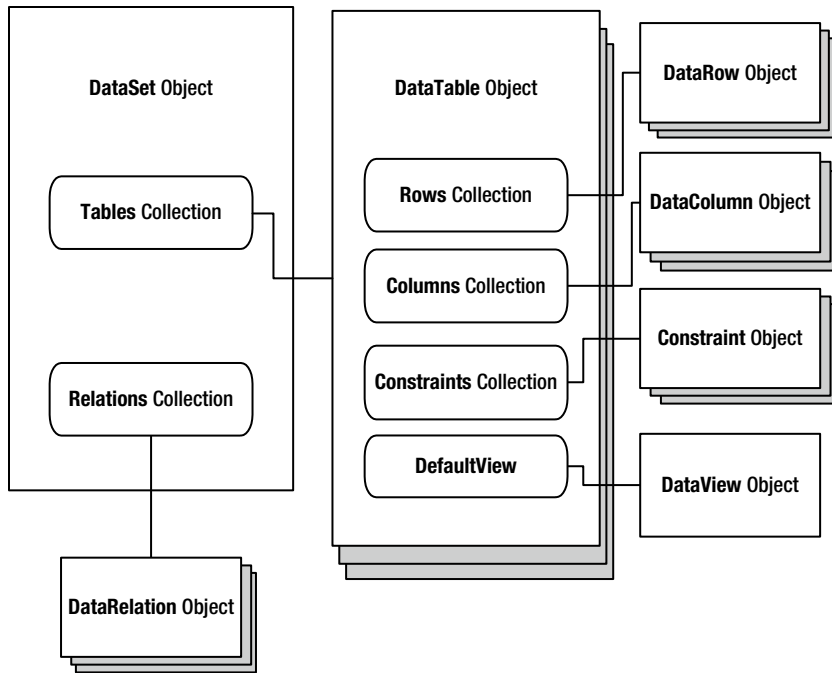


Figure 14-15. The DataSet family of objects

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

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

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

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

    // Try to open database and read information.
    try
```

```

{
    con.Open();

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

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

```

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

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

Selecting Multiple Tables

A `DataSet` can contain as many tables as you need, and you can even add relationships between the tables to better emulate the underlying relational data source. Unfortunately, you have no way to connect tables together automatically based on relationships in the underlying data source. However, you can add relations with a few extra lines of code, as shown in the next example.

In the pubs database, authors are linked to titles by using three tables. This arrangement (called a *many-to-many* relationship, shown in Figure 14-16) allows several authors to be related to one title and several titles to be related to one author. Without the intermediate `TitleAuthor` table, the database would be restricted to a one-to-many relationship, which would allow only a single author for each title.

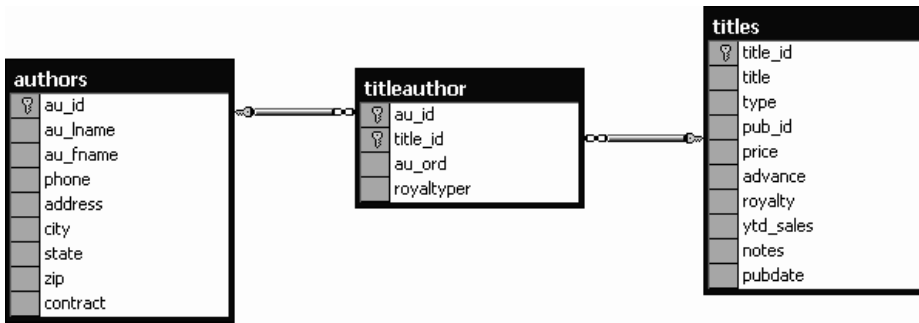


Figure 14-16. A many-to-many relationship

In an application, you would rarely need to access these tables individually. Instead, you would need to combine information from them in some way (for example, to find out what author wrote a given book). On its own, the Titles table indicates only the author ID. It doesn't provide additional information such as the author's name and address. To link this information together, you can use a special SQL Select statement called a *Join query*. Alternatively, you can use the features built into ADO.NET, as demonstrated in this section.

The next example provides a simple page that lists authors and the titles they have written. The interesting thing about this page is that it's generated using ADO.NET table linking.

To start, the standard ADO.NET data access objects are created, including a DataSet. All these steps are performed in a custom CreateList() method, which is called from the Page.Load event handler so that the output is created when the page is first generated:

```
// Define the ADO.NET objects.
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection con = new SqlConnection(connectionString);

string selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";
SqlCommand cmd = new SqlCommand(selectSQL, con);
SqlDataAdapter adapter = new SqlDataAdapter(cmd);
DataSet dsPubs = new DataSet();
```

Next, the information for all three tables is pulled from the database and placed in the DataSet. This task could be accomplished with three separate Command objects, but to make the code a little leaner, this example uses only one and modifies the CommandText property as needed.

```
try
{
    con.Open();
    adapter.Fill(dsPubs, "Authors");

    // This command is still linked to the data adapter.
    cmd.CommandText = "SELECT au_id, title_id FROM TitleAuthor";
    adapter.Fill(dsPubs, "TitleAuthor");

    // This command is still linked to the data adapter.
    cmd.CommandText = "SELECT title_id, title FROM Titles";
    adapter.Fill(dsPubs, "Titles");
}
```

```

catch (Exception err)
{
    lblList.Text = "Error reading list of names. ";
    lblList.Text += err.Message;
}
finally
{
    con.Close();
}

```

Defining Relationships

Now that all the information is in the `DataSet`, you can create two `DataRelation` objects to make it easier to navigate through the linked information. In this case, these `DataRelation` objects match the foreign-key restrictions that are defined in the database.

■ **Note** A *foreign key* is a constraint that you can set up in your database to link one table to another. For example, the `TitleAuthor` table is linked to the `Titles` and the `Authors` tables by two foreign keys. The `title_id` field in the `TitleAuthor` table has a foreign key that binds it to the `title_id` field in the `Titles` table. Similarly, the `au_id` field in the `TitleAuthor` table has a foreign key that binds it to the `au_id` field in the `Authors` table. After these links are established, certain rules come into play. For example, you can't create a `TitleAuthor` record that specifies author or title records that don't exist.

To create a `DataRelation`, you need to specify the linked fields from two tables, and you need to give your `DataRelation` a unique name. The order of the linked fields is important. The first field is the parent, and the second field is the child. (The idea here is that one parent can have many children, but each child can have only one parent. In other words, the *parent-to-child* relationship is another way of saying a *one-to-many* relationship.) In this example, each book title can have more than one entry in the `TitleAuthor` table. Each author can also have more than one entry in the `TitleAuthor` table:

```

DataRelation Titles_TitleAuthor = new DataRelation("Titles_TitleAuthor",
    dsPubs.Tables["Titles"].Columns["title_id"],
    dsPubs.Tables["TitleAuthor"].Columns["title_id"]);

DataRelation Authors_TitleAuthor = new DataRelation("Authors_TitleAuthor",
    dsPubs.Tables["Authors"].Columns["au_id"],
    dsPubs.Tables["TitleAuthor"].Columns["au_id"]);

```

After you've created these `DataRelation` objects, you must add them to the `DataSet`:

```

dsPubs.Relations.Add(Titles_TitleAuthor);
dsPubs.Relations.Add(Authors_TitleAuthor);

```

The remaining code loops through the DataSet. However, unlike the previous example, which moved through one table, this example uses the DataRelation objects to branch to the other linked tables. It works like this:

1. Select the first record from the Authors table.
2. Using the Authors_TitleAuthor relationship, find the child records that correspond to this author. To do so, you call the DataRow.GetChildRows() method, and pass in the DataRelationship object that models the relationship between the Authors and TitleAuthor table.
3. For each matching record in TitleAuthor, look up the corresponding Title record to get the full text title. To do so, you call the DataRow.GetParentRows() method and pass in the DataRelationship object that connects the TitleAuthor and Titles table.
4. Move to the next Author record and repeat the process.

The code is lean and economical:

```
foreach (DataRow rowAuthor in dsPubs.Tables["Authors"].Rows)
{
    lblList.Text += "<br /><b>" + rowAuthor["au_fname"];
    lblList.Text += " " + rowAuthor["au_lname"] + "</b><br />";

    foreach (DataRow rowTitleAuthor in
        rowAuthor.GetChildRows(Authors_TitleAuthor))
    {
        DataRow rowTitle =
            rowTitleAuthor.GetParentRows(Titles_TitleAuthor)[0];
        lblList.Text += "&nbsp;&nbsp;&nbsp;";
        lblList.Text += rowTitle["title"] + "<br />";
    }
}
```

Figure 14-17 shows the final result.

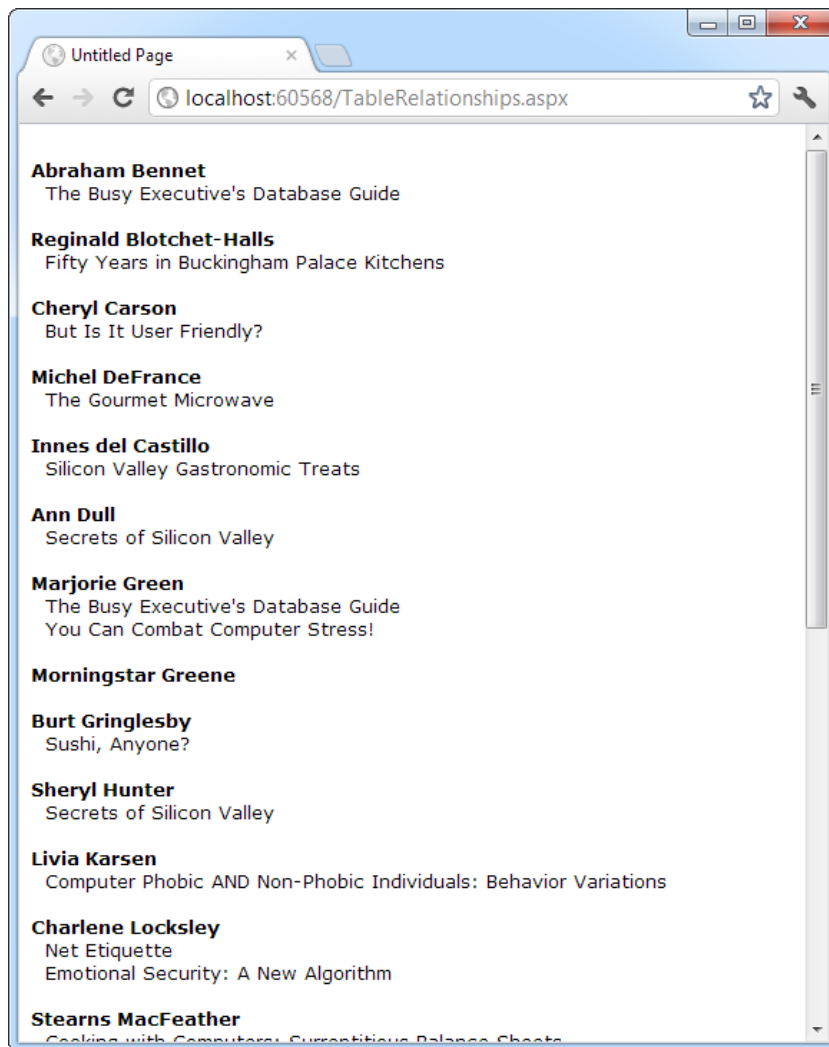


Figure 14-17. Hierarchical information from two tables

If authors and titles have a simple one-to-many relationship, you could use simpler code, as follows:

```
foreach (DataRow rowAuthor in dsPubs.Tables["Authors"].Rows)
{
    // Display author.
    foreach (DataRow rowTitle in rowAuthor.GetChildRows(Authors_Titles))
    {
        // Display title.
    }
}
```

But having seen the more complicated example, you're ready to create and manage multiple `DataRelation` objects on your own.

The Last Word

This chapter gave you a solid introduction to ADO.NET. You now know how to connect to a database in your web pages, retrieve the information you need, and execute commands to update, insert, and delete data.

Although you've seen all the core concepts behind ADO.NET, there's still much more to learn. In the following chapters, you'll learn about ASP.NET's data-binding system and rich data controls, and you'll see how you can use them to write more-practical data-driven web pages. And much later, you'll learn about how to take your skills to the next level by building ADO.NET-powered components (Chapter 22) and using the higher-level LINQ to Entities framework (Chapter 24).



Data Binding

In the previous chapter, you learned how to use ADO.NET to retrieve information from a database, how to store it in the DataSet, and how to apply changes by using direct commands. These techniques are flexible and powerful, but they aren't always convenient.

For example, you can use the DataSet or the DataReader to retrieve rows of information, format them individually, and add them to an HTML table on a web page. Conceptually, this isn't too difficult. However, moving through the data, formatting columns, and displaying it in the correct order still requires a lot of repetitive code. Repetitive code may be easy, but it's also error-prone, difficult to enhance, and unpleasant to read. Fortunately, ASP.NET adds a feature that allows you to skip this process and pop data directly into HTML elements and fully formatted controls. It's called *data binding*. In this chapter, you'll learn how to use data binding to display data more efficiently. You'll also learn how to use the ASP.NET *data source controls* to retrieve your data from a database without writing a line of ADO.NET code.

Introducing Data Binding

The basic principle of data binding is this: you tell a control where to find your data and how you want it displayed, and the control handles the rest of the details. Data binding in ASP.NET seems superficially similar to data binding in the world of desktop or client/server applications, but in truth, it's fundamentally different. In those environments, data binding involves creating a direct connection between a data source and a control in an application window. If the user changes a value in the onscreen control, the data in the linked database is modified automatically. Similarly, if the database changes while the user is working with it (for example, another user commits a change), the display can be refreshed automatically.

This type of data binding isn't practical in the ASP.NET world, because you can't effectively maintain a database connection over the Internet. This "direct" data binding also severely limits scalability and reduces flexibility. In fact, data binding has acquired a bad reputation for exactly these reasons.

ASP.NET data binding, on the other hand, has little in common with direct data binding. ASP.NET data binding works in one direction only. Information moves *from* a data object *into* a control. Then the data objects are thrown away, and the page is sent to the client. If the user modifies the data in a data-bound control, your program can update the corresponding record in the database, but nothing happens automatically.

ASP.NET data binding is much more flexible than old-style data binding. Many of the most powerful data-binding controls, such as the GridView and DetailsView, give you unprecedented control over the presentation of your data, allowing you to format it, change its layout, embed it in other ASP.NET controls, and so on. You'll learn about these features and ASP.NET's rich data controls in Chapter 16.

Types of ASP.NET Data Binding

Two types of ASP.NET data binding exist: single-value binding and repeated-value binding. Single-value data binding is by far the simpler of the two, whereas repeated-value binding provides the foundation for the most advanced ASP.NET data controls.

Single-Value, or “Simple,” Data Binding

You can use *single-value data binding* to add information anywhere on an ASP.NET page. You can even place information into a control property or as plain text inside an HTML tag. Single-value data binding doesn’t necessarily have anything to do with ADO.NET. Instead, single-value data binding allows you to take a variable, a property, or an expression and insert it dynamically into a page. Single-value binding also helps you create templates for the rich data controls you’ll study in Chapter 16.

Repeated-Value, or “List,” Binding

Repeated-value data binding allows you to display an entire table (or just a single field from a table). Unlike single-value data binding, this type of data binding requires a special control that supports it. Typically, this is a list control such as `CheckBoxList` or `ListBox`, but it can also be a much more sophisticated control such as the `GridView` (which is described in Chapter 16). You’ll know that a control supports repeated-value data binding if it provides a `DataSource` property. As with single-value binding, repeated-value binding doesn’t necessarily need to use data from a database, and it doesn’t have to use the ADO.NET objects. For example, you can use repeated-value binding to bind data from a collection or an array.

How Data Binding Works

Data binding works a little differently depending on whether you’re using single-value or repeated-value binding. To use single-value binding, you must insert a data-binding expression into the markup in the .aspx file (not the code-behind file). To use repeated-value binding, you must set one or more properties of a data control. Typically, you’ll perform this initialization when the `Page.Load` event fires. You’ll see examples of both techniques later in this chapter.

After you specify data binding, you need to activate it. You accomplish this task by calling the `DataBind()` method. The `DataBind()` method is a basic piece of functionality supplied in the `Control` class. It automatically binds a control and any child controls that it contains. With repeated-value binding, you can use the `DataBind()` method of the specific list control you’re using. Alternatively, you can bind the whole page at once by calling the `DataBind()` method of the current `Page` object. After you call this method, all the data-binding expressions in the page are evaluated and replaced with the specified value.

Typically, you call the `DataBind()` method in the `Page.Load` event handler. If you forget to use it, ASP.NET will ignore your data-binding expressions, and the client will receive a page that contains empty values.

This is a general description of the whole process. To really understand what’s happening, you need to work with some specific examples.

Using Single-Value Data Binding

Single-value data binding is really just a different approach to dynamic text. To use it, you add special data-binding expressions into your .aspx files. These expressions have the following format:

```
<%# expression_goes_here %>
```

This may look like a script block, but it isn't. If you try to write any code inside this tag, you will receive an error. The only thing you can add is a valid data-binding expression. For example, if you have a public or protected variable named `Country` in your page, you could write the following:

```
<%# Country %>
```

When you call the `DataBind()` method for the page, this text will be replaced with the value for `Country` (for example, Spain). Similarly, you could use a property or a built-in ASP.NET object as follows:

```
<%# Request.Browser.Browser %>
```

This would substitute a string with the current browser name (for example, IE). In fact, you can even call a function defined on your page, or execute a simple expression, provided it returns a result that can be converted to text and displayed on the page. Thus, the following data-binding expressions are all valid:

```
<%# GetUserName(ID) %>
```

```
<%# 1 + (2 * 20) %>
```

```
<%# "John " + "Smith" %>
```

Remember, you place these data-binding expressions in the markup portion of your `.aspx` file, not your code-behind file.

A Simple Data-Binding Example

This section shows a simple example of single-value data binding. The example has been stripped to the bare minimum amount of detail needed to illustrate the concept.

You start with a variable defined in your Page class, which is called `TransactionCount`:

```
public partial class SimpleDataBinding : System.Web.UI.Page
{
    protected int TransactionCount;

    // (Additional code omitted.)
}
```

Note that this variable must be designated as public, protected, or internal, but not private. If you make the variable private, ASP.NET will not be able to access it when it's evaluating the data-binding expression.

Now, assume that this value is set in the `Page.Load` event handler by using some database lookup code. For testing purposes, the example skips this step and hard-codes a value:

```
protected void Page_Load(object sender, EventArgs e)
{
    // (You could use database code here
    // to look up a value for TransactionCount.)
    TransactionCount = 10;

    // Now convert all the data-binding expressions on the page.
    this.DataBind();
}
```

Two actions take place in this event handler: the `TransactionCount` variable is set to 10, and all the data-binding expressions on the page are bound. Currently, no data-binding expressions exist, so this method has no effect. Notice that this example uses the `this` keyword to refer to the current page. You could just write `DataBind()` without the `this` keyword, because the default object is the current Page object. However, using the `this` keyword helps clarify which object is being used.

To make this data binding accomplish something, you need to add a data-binding expression. Usually, it's easiest to add this value directly to the markup in the .aspx file. To do so, click the Source button at the bottom of the web page designer window. Figure 15-1 shows an example with a Label control.

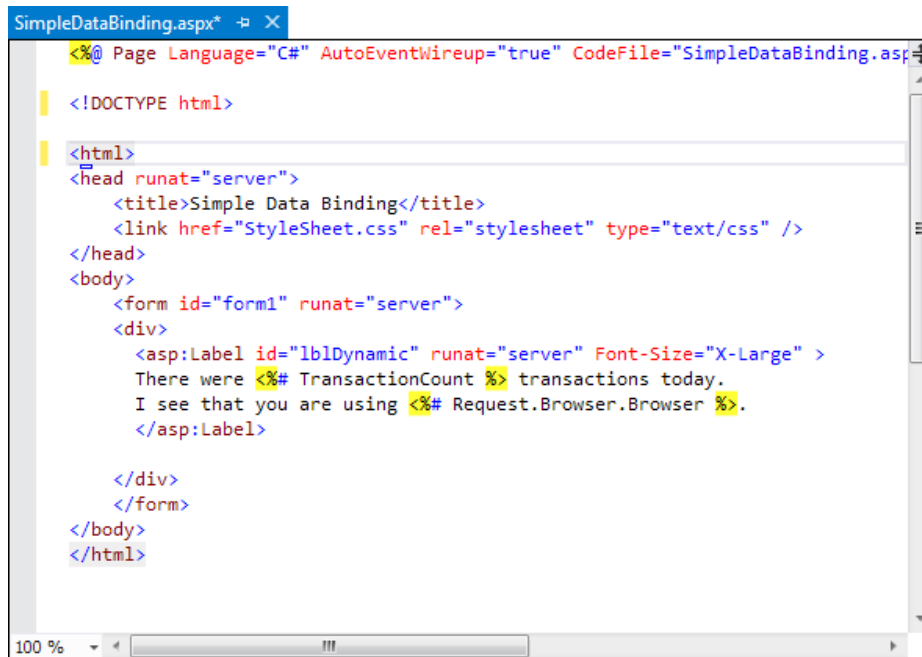


Figure 15-1. Source view in the web page designer

To add your expression, find the tag for the Label control. Modify the text inside the label as shown here:

```

<asp:Label id="lblDynamic" runat="server" Font-Size="X-Large">
There were <%# TransactionCount %> transactions today.
I see that you are using <%# Request.Browser.Browser %>.
</asp:Label>
  
```

This example uses two separate data-binding expressions, which are inserted along with the normal static text. The first data-binding expression references the TransactionCount variable, and the second uses the built-in Request object to determine some information about the user's browser. When you run this page, the output looks like Figure 15-2.

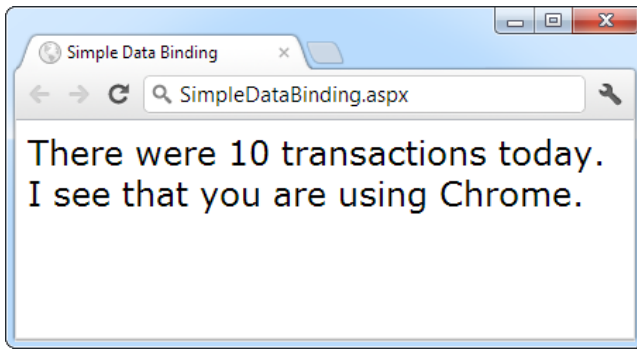


Figure 15-2. *The result of data binding*

The data-binding expressions have been automatically replaced with the appropriate values. If the page is posted back, you could use additional code to modify `TransactionCount`, and as long as you call the `DataBind()` method, that information will be popped into the page in the data-binding expression you've defined.

If, however, you forget to call the `DataBind()` method, the data-binding expressions will be ignored, and the user will see a somewhat confusing window that looks like Figure 15-3.

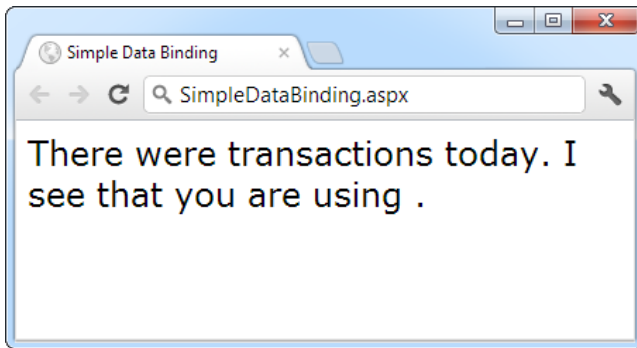


Figure 15-3. *The non-data-bound page*

■ **Note** When using single-value data binding, you need to consider when you should call the `DataBind()` method. For example, if you made the mistake of calling it before you set the `TransactionCount` variable, the corresponding expression would just be converted to 0. Remember, data binding is a one-way street. This means that changing the `TransactionCount` variable after you've used the `DataBind()` method won't produce any visible effect. Unless you call the `DataBind()` method again, the displayed value won't be updated.

Simple Data Binding with Properties

The previous example uses a data-binding expression to set static text information inside a label tag. However, you can also use single-value data binding to set other types of information on your page, including control properties. To do this, you simply have to know where to put the data-binding expression in the web page markup.

For example, consider the following page, which defines a variable named `URL` and uses it to point to a picture in the application directory:

```
public partial class DataBindingUrl : System.Web.UI.Page
{
    protected string URL;

    protected void Page_Load(Object sender, EventArgs e)
    {
        URL = "Images/picture.jpg";
        this.DataBind();
    }
}
```

You can now use this `URL` to create a label, as shown here:

```
<asp:Label id="lblDynamic" runat="server"><%# URL %></asp:Label>
```

You can also use it for a check box caption:

```
<asp:CheckBox id="chkDynamic" Text="<%# URL %>" runat="server" />
```

or you can use it for a target for a hyperlink:

```
<asp:Hyperlink id="lnkDynamic" Text="Click here!" NavigateUrl="<%# URL %>"
runat="server" />
```

You can even use it for a picture:

```
<asp:Image id="imgDynamic" ImageUrl="<%# URL %>" runat="server" />
```

The only trick is that you need to edit these control tags by hand. Figure 15-4 shows what a page that uses all these elements would look like.

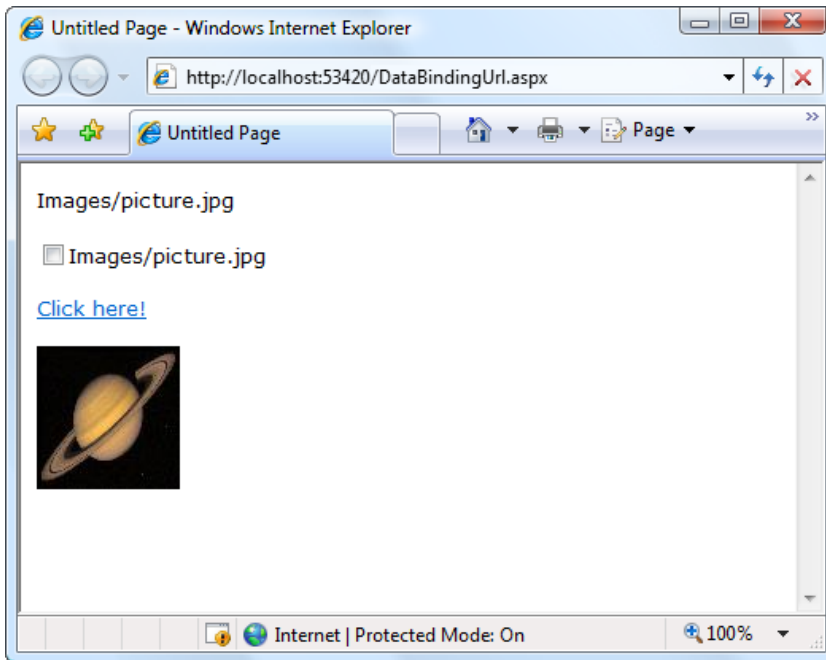


Figure 15-4. Multiple ways to bind the same data

To examine this example in more detail, try the sample code for this chapter.

Problems with Single-Value Data Binding

Before you start using single-value data-binding techniques in every aspect of your ASP.NET programs, you should consider some of the serious drawbacks that this approach can present:

Putting code into a page's user interface: One of ASP.NET's great advantages is that it allows developers to separate the user interface code (the HTML and control tags in the .aspx file) from the actual code used for data access and all other tasks (in the code-behind file). However, overenthusiastic use of single-value data binding can encourage you to disregard that distinction and start coding function calls and even operations into your page. If not carefully managed, this can lead to complete disorder.

Fragmenting code: When using data-binding expressions, it may not be obvious where the functionality resides for different operations. This is particularly a problem if you blend both approaches—for example, if you use data binding to fill a control and also modify that control directly in code. Even worse, the data-binding code may have certain dependencies that aren't immediately obvious. If the page code changes, or a variable or function is removed or renamed, the corresponding data-binding expression could stop providing valid information without any explanation or even an obvious error. All of these details make it more difficult to maintain your code, and make it more difficult for multiple developers to work together on the same project.

Of course, some developers love the flexibility of single-value data binding and use it to great effect, making the rest of their code more economical and streamlined. It's up to you to be aware of (and avoid) the potential drawbacks.

Note In one case, single-value data binding is quite useful—when building *templates*. Templates declare a block of markup that's reused for each record in a table. However, they work only with certain rich data controls, such as the GridView. You'll learn more about this feature in Chapter 16.

If you decide not to use single-value data binding, you can accomplish the same thing by using code. For example, you could use the following event handler to display the same output as the first label example:

```
protected void Page_Load(Object sender, EventArgs e)
{
    TransactionCount = 10;
    lblDynamic.Text = "There were " + TransactionCount.ToString();
    lblDynamic.Text += " transactions today. ";
    lblDynamic.Text += "I see that you are using " + Request.Browser.Browser;
}
```

This code dynamically fills in the label without using data binding. The trade-off is more code.

When you use data-binding expressions, you end up complicating your markup with additional details about your code (such as the names of the variables in your code-behind class). When you use the code-only approach, you end up doing the reverse—complicating your code with additional details about the page markup (such as the text you want to display). In many cases, the best approach depends on your specific scenario. Data-binding expressions are great for injecting small bits of information into an otherwise detailed page. The dynamic code approach gives you more flexibility, and works well when you need to perform more-extensive work to shape the page (for example, interacting with multiple controls, changing content and formatting, retrieving the information you want to display from different sources, and so on).

Using Repeated-Value Data Binding

Although using simple data binding is optional, *repeated-value binding* is so useful that almost every ASP.NET application will want to use it somewhere.

Repeated-value data binding works with the ASP.NET list controls (and the rich data controls described in the next chapter). To use repeated-value binding, you link one of these controls to a data source (such as a field in a data table). When you call `DataBind()`, the control automatically creates a full list by using all the corresponding values. This saves you from writing code that loops through the array or data table and manually adds elements to a control. Repeated-value binding can also simplify your life by supporting advanced formatting and template options that automatically configure how the data should look when it's placed in the control.

To create a data expression for list binding, you need to use a list control that explicitly supports data binding. Luckily, ASP.NET provides a number of list controls, many of which you've probably already used in other applications or examples:

ListBox, DropDownList, CheckBoxLayout, and RadioButtonList: These web controls provide a list for a single field of information.

HtmlSelect: This server-side HTML control represents the HTML `<select>` element and works essentially the same way as the `ListBox` web control. Generally, you'll use this control only for backward compatibility.

GridView, DetailsView, FormView, and ListView: These rich web controls allow you to provide repeating lists or grids that can display more than one field of information at a time. For example, if you bind one of these controls to a full-fledged table in a DataSet, you can display the values from multiple fields. These controls offer the most powerful and flexible options for data binding.

With repeated-value data binding, you can write a data-binding expression in your .aspx file, or you can apply the data binding by setting control properties. In the case of the simpler list controls, you'll usually just set properties. Of course, you can set properties in many ways, such as by using code in a code-behind file or by modifying the control tag in the .aspx file, possibly with the help of Visual Studio's Properties window. The approach you take doesn't matter. The important detail is that with the simple list controls, you don't use any `<%# expression %>` data-binding expressions.

To continue any further with data binding, it will help to divide the subject into a few basic categories. You'll start by looking at data binding with the list controls.

Data Binding with Simple List Controls

In some ways, data binding to a list control is the simplest kind of data binding. You need to follow only three steps:

1. Create and fill some kind of data object. You have numerous options, including an array, the basic ArrayList and Hashtable collections, the strongly typed List and Dictionary collections, and the DataTable and DataSet objects. Essentially, you can use any type of collection that supports the IEnumerable interface, although you'll discover that each class has specific advantages and disadvantages.
2. Link the object to the appropriate control. To do this, you need to set only a couple of properties, including DataSource. If you're binding to a full DataSet, you'll also need to set the DataMember property to identify the appropriate table you want to use.
3. Activate the binding. As with single-value binding, you activate data binding by using the DataBind() method, either for the specific control or for all contained controls at once by using the DataBind() method for the current page.

This process is the same whether you're using the ListBox, the DropDownList, the CheckedList, the RadioButtonList, or even the HtmlSelect control. All these controls provide the same properties and work the same way. The only difference is in the way they appear on the final web page.

A Simple List-Binding Example

To try this type of data binding, add a ListBox control to a new web page. Then use the Page.Load event handler to create a strongly typed List collection to use as a data source:

```
List<string> fruit = new List<string>();
fruit.Add("Kiwi");
fruit.Add("Pear");
fruit.Add("Mango");
fruit.Add("Blueberry");
fruit.Add("Apricot");
fruit.Add("Banana");
fruit.Add("Peach");
fruit.Add("Plum");
```


As you learned in Chapter 3, strongly typed collections such as `List` are ideal when you want your collection to hold just a single type of object (for example, just strings). When you use the generic collections, you choose the item type you want to use, and the collection object is “locked in” to your choice. This means that if you try to add another type of object that doesn’t belong in the collection, you’ll get a compile-time error warning you of the mistake. And when you pull an item out of the collection, you don’t need to write casting code to convert it to the right type, because the compiler already knows what type of objects you’re using.

Now you can link this collection to the `ListBox` control:

```
lstItems.DataSource=fruit;
```

To activate the binding, use the `DataBind()` method:

```
this.DataBind();
```

You could also use `lstItems.DataBind()` to bind just the `ListBox` control. Figure 15-5 shows the resulting web page.

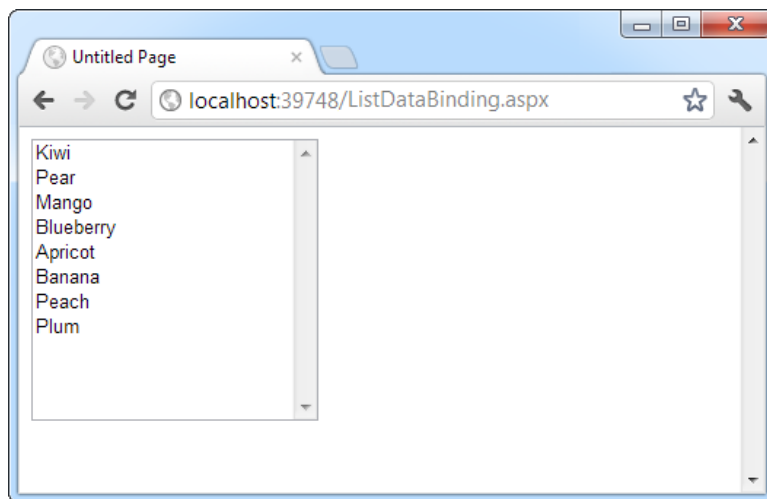


Figure 15-5. A data-bound list

This technique can save quite a few lines of code. This example doesn’t offer a lot of savings because the collection is created just before it’s displayed. In a more realistic application, however, you might be using a function that returns a ready-made collection to you:

```
List<string> fruit;  
fruit = GetFruitsInSeason("Summer");
```

In this case, it’s extremely simple to add the extra two lines needed to bind and display the collection in the window:

```
lstItems.DataSource = fruit;  
this.DataBind();
```

or you could even change it to the following, even more compact, code:

```
lstItems.DataSource = GetFruitsInSeason("Summer");
this.DataBind();
```

On the other hand, consider the extra trouble you would have to go through if you didn't use data binding. This type of savings compounds rapidly, especially when you start combining data binding with multiple controls, advanced objects such as DataSets, or advanced controls that apply formatting through templates.

Multiple Binding

You can bind the same data list object to multiple controls. Consider the following example, which compares all the types of list controls at your disposal by loading them with the same information:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Create and fill the collection.
    List<string> fruit = new List<string>();
    fruit.Add("Kiwi");
    fruit.Add("Pear");
    fruit.Add("Mango");
    fruit.Add("Blueberry");
    fruit.Add("Apricot");
    fruit.Add("Banana");
    fruit.Add("Peach");
    fruit.Add("Plum");

    // Define the binding for the list controls.
    MyListBox.DataSource = fruit;
    MyDropDownListBox.DataSource = fruit;
    MyHtmlSelect.DataSource = fruit;
    MyCheckBoxList.DataSource = fruit;
    MyRadioButtonList.DataSource = fruit;

    // Activate the binding.
    this.DataBind();
}
```

Figure 15-6 shows the rendered page.

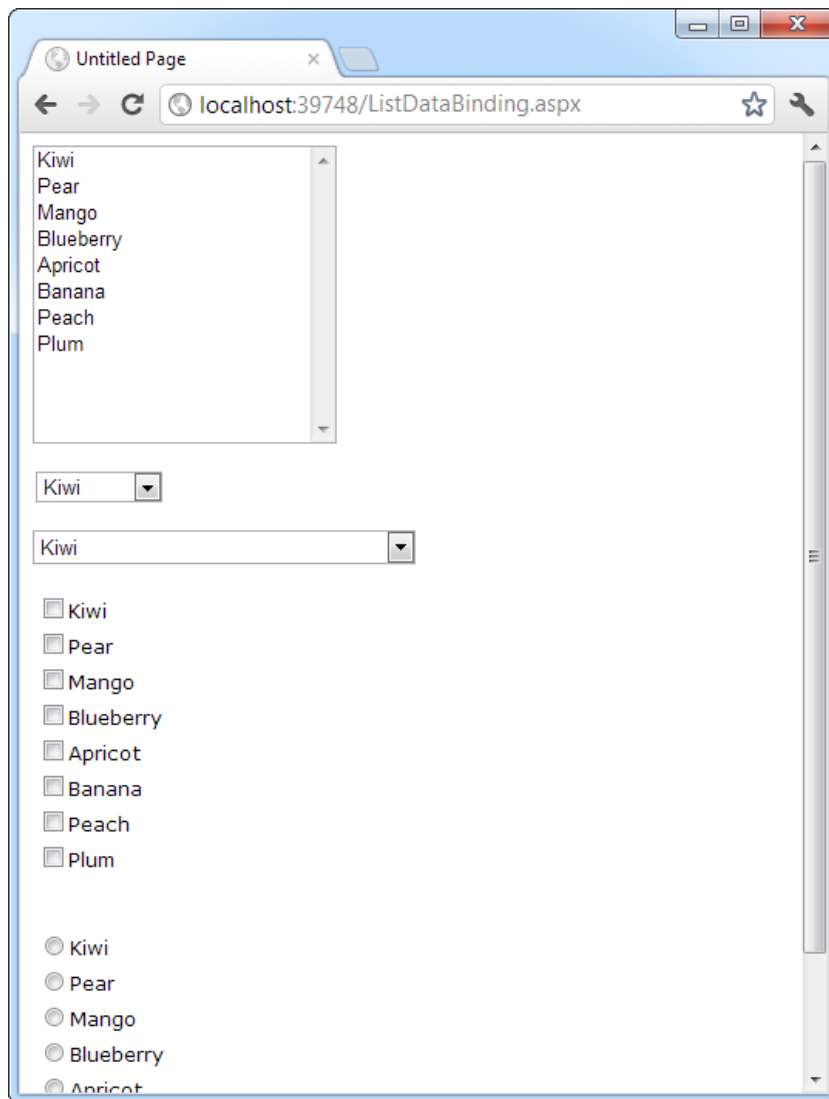


Figure 15-6. Multiple bound lists

This is another area where ASP.NET data binding may differ from what you have experienced in a desktop application. In traditional data binding, all the different controls are sometimes treated like “views” on the same data source, and you can work with only one record from the data source at a time. In this type of data binding, when you select Pear in one list control, the other list controls automatically refresh so that they too have Pear selected (or the corresponding information from the same row). This isn’t how ASP.NET uses data binding. If you want this sort of effect, you need to write custom code to pull it off.

Data Binding with a Dictionary Collection

A *dictionary collection* is a special kind of collection in which every item (or *definition*, to use the dictionary analogy) is indexed with a specific key (or dictionary *word*). This is similar to the way that built-in ASP.NET collections such as Session, Application, and Cache work.

Dictionary collections always need keys. This makes it easier to retrieve the item you want. In ordinary collections, such as the List, you need to find the item you want by its index number position, or—more often—by traveling through the whole collection until you come across the right item. With a dictionary collection, you retrieve the item you want by using its key. Generally, ordinary collections make sense when you need to work with all the items at once, while dictionary collections make sense when you frequently retrieve a single specific item.

You can use two basic dictionary-style collections in .NET. The Hashtable collection (in the System.Collections namespace) allows you to store any type of object and use any type of object for the key values. The Dictionary collection (in the System.Collections.Generic namespace) uses generics to provide the same “locking in” behavior as the List collection. You choose the item type and the key type upfront to prevent errors and reduce the amount of casting code you need to write.

The following example uses the Dictionary collection class, which it creates once—the first time the page is requested. You create a Dictionary object in much the same way you create a List collection. The only difference is that you need to supply a unique key for every item. This example uses the lazy practice of assigning a sequential number for each key:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        // Use integers to index each item. Each item is a string.
        Dictionary<int, string> fruit = new Dictionary<int, string>();

        fruit.Add(1, "Kiwi");
        fruit.Add(2, "Pear");
        fruit.Add(3, "Mango");
        fruit.Add(4, "Blueberry");
        fruit.Add(5, "Apricot");
        fruit.Add(6, "Banana");
        fruit.Add(7, "Peach");
        fruit.Add(8, "Plum");

        // Define the binding for the list controls.
        MyListBox.DataSource = fruit;

        // Choose what you want to display in the list.
        MyListBox.DataTextField = "Value";

        // Activate the binding.
        this.DataBind();
    }
}
```

There’s one new detail here. It’s this line:

```
MyListBox.DataTextField="Value";
```

Each item in a dictionary-style collection has both a key and a value associated with it. If you don’t specify which property you want to display, ASP.NET simply calls the ToString() method on each collection item. This

may or may not produce the result you want. However, by inserting this line of code, you control exactly what appears in the list. The page will now appear as expected, with all the fruit names.

■ **Note** Notice that you need to enclose the property name in quotation marks. ASP.NET uses reflection to inspect your object and find the property that has the name `Value` at runtime.

You might want to experiment with what other types of collections you can bind to a list control. One interesting option is to use a built-in ASP.NET control such as the Session object. An item in the list will be created for every currently defined Session variable, making this trick a nice little debugging tool to quickly check current session information.

Using the `DataValueField` Property

Along with the `DataTextField` property, all list controls that support data binding also provide a `DataValueField` property, which adds the corresponding information to the value attribute in the control element. This allows you to store extra (undisplayed) information that you can access later. For example, you could use these two lines to define your data binding with the previous example:

```
MyListBox.DataTextField = "Value";
MyListBox.DataValueField = "Key";
```

The control will appear the same, with a list of all the fruit names in the collection. However, if you look at the rendered HTML that's sent to the client browser, you'll see that value attributes have been set with the corresponding numeric key for each item:

```
<select name="MyListBox" id="MyListBox" >
  <option value="1">Kiwik</option>
  <option value="2">Pear</option>
  <option value="3">Mango</option>
  <option value="4">Blueberry</option>
  <option value="5">Apricot</option>
  <option value="6">Banana</option>
  <option value="7">Peach</option>
  <option value="8">Plum</option>
</select>
```

You can retrieve this value later by using the `SelectedItem` property to get additional information. For example, you could set the `AutoPostBack` property of the list control to true, and add the following code:

```
protected void MyListBox_SelectedIndexChanged(Object sender,
    EventArgs e)
{
    lblMessage.Text = "You picked: " + MyListBox.SelectedItem.Text;
    lblMessage.Text += " which has the key: " + MyListBox.SelectedItem.Value;
}
```

Figure 15-7 demonstrates the result. This technique is particularly useful with a database. You could embed a unique ID into the value property and be able to quickly look up a corresponding record depending on the user's selection by examining the value of the `SelectedItem` object.

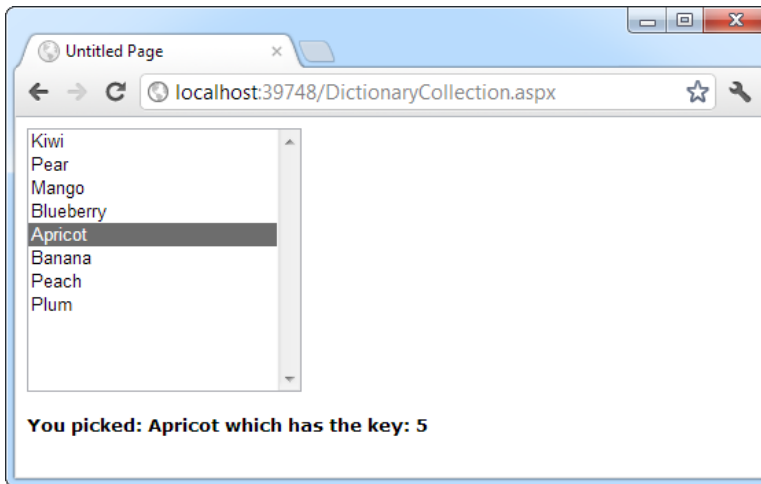


Figure 15-7. Binding to the key and value properties

Note that for this to work, you can't regenerate the list after every postback. If you do, the selected item information will be lost and an error will occur. The preceding example handles this by checking the `Page.IsPostBack` property. If it's false (which indicates that the page is being requested for the first time), the page builds the list. When the page is rendered, the current list of items is stored in view state. When the page is posted back, the list of items already exists and doesn't need to be re-created.

Data Binding with ADO.NET

So far, the examples in this chapter have dealt with data binding that doesn't involve databases or any part of ADO.NET. Although this is an easy way to familiarize yourself with the concepts, and a useful approach in its own right, you get the greatest advantage of data binding when you use it in conjunction with a database.

When you're using data binding with the information drawn from a database, the data-binding process takes place in the same three steps. First you create your data source, which will be a `DataReader` or `DataSet` object. A `DataReader` generally offers the best performance, but it limits your data binding to a single control because it is a forward-only reader. As it fills a control, it traverses the results from beginning to end. After it's finished, it can't go back to the beginning; so it can't be used in another data-binding operation. For this reason, a `DataSet` is a more common choice.

The next example creates a `DataSet` and binds it to a list. In this example, the `DataSet` is filled by hand, but it could just as easily be filled by using a `DataAdapter` object, as you saw in the previous chapter.

To fill a `DataSet` by hand, you need to follow several steps:

1. Create the `DataSet`.
2. Create a new `DataTable` and add it to the `DataSet.Tables` collection.
3. Define the structure of the table by adding `DataColumn` objects (one for each field) to the `DataTable.Columns` collection.
4. Supply the data. You can get a new, blank row that has the same structure as your `DataTable` by calling the `DataTable.NewRow()` method. You must then set the data in all its fields, and add the `DataRow` to the `DataTable.Rows` collection.

Here's how the code unfolds:

```
// Define a DataSet with a single DataTable.
DataSet dsInternal = new DataSet();
dsInternal.Tables.Add("Users");

// Define two columns for this table.
dsInternal.Tables["Users"].Columns.Add("Name");
dsInternal.Tables["Users"].Columns.Add("Country");

// Add some actual information into the table.
DataRow rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "John";
rowNew["Country"] = "Uganda";
dsInternal.Tables["Users"].Rows.Add(rowNew);

rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "Samantha";
rowNew["Country"] = "Belgium";
dsInternal.Tables["Users"].Rows.Add(rowNew);

rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "Rico";
rowNew["Country"] = "Japan";
dsInternal.Tables["Users"].Rows.Add(rowNew);
```

Next you bind the DataTable from the DataSet to the appropriate control. Because list controls can show only a single column at a time, you also need to choose the field you want to display for each item by setting the DataTextField property:

```
// Define the binding.
lstUser.DataSource = dsInternal.Tables["Users"];
lstUser.DataTextField = "Name";
```

Alternatively, you could use the entire DataSet for the data source, instead of just the appropriate table. In that case, you would have to select a table by setting the control's DataMember property. This is an equivalent approach, but the code is slightly different:

```
// Define the binding.
lstUser.DataSource = dsInternal;
lstUser.DataMember = "Users";
lstUser.DataTextField = "Name";
```

As always, the last step is to activate the binding:

```
this.DataBind();
```

The final result is a list with the information from the specified database field, as shown in Figure 15-8. The list box will have an entry for every single record in the table, even if it appears more than once, from the first row to the last.

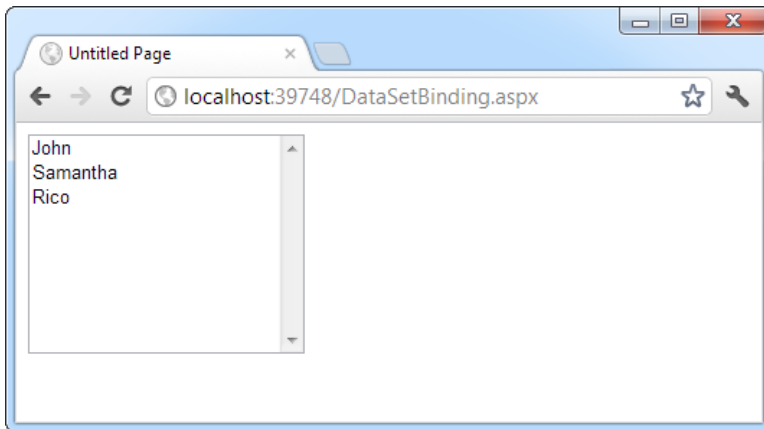


Figure 15-8. *DataSet binding*

■ **Tip** The simple list controls require you to bind their Text or Value property to a single data field in the data source object. However, much more flexibility is provided by the more advanced data-binding controls examined in the next chapter. They allow fields to be combined in just about any way you can imagine.

Creating a Record Editor

The next example is more practical. It's a good illustration of how you might use data binding in a full ASP.NET application. This example allows the user to select a record and update one piece of information by using data-bound list controls.

The first step is to add the connection string to your web.config file. This example uses the Products table from the Northwind database included with many versions of SQL Server. Here's how you can define the connection string for SQL Server Express LocalDB:

```
<configuration>
  <connectionStrings>
    <add name="Northwind" connectionString=
      "Data Source=(localdb)\v11.0;Initial Catalog=Northwind;Integrated
      Security=SSPI" />
  </connectionStrings>
  ...
</configuration>
```

To use the full version of SQL Server, remove the (localdb)\v11.0 portion. To use a database server on another computer, supply the computer name for the Data Source connection string property. (For more details about connection strings, refer to Chapter 14.)

The next step is to retrieve the connection string and store it in a private variable in the Page class so that every part of your page code can access it easily. After you've imported the System.Web.Configuration namespace, you can create a member variable in your code-behind class that's defined like this:

```
private string connectionString =
  WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
```


The next step is to create a drop-down list that allows the user to choose a product for editing. The Page.Load event handler takes care of this task—retrieving the data, binding it to the drop-down list control, and then activating the binding. Before you go any further, make sure you’ve imported the System.Data.SqlClient namespace, which allows you to use the SQL Server provider to retrieve data.

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        // Define the ADO.NET objects for selecting products from the database.
        string selectSQL = "SELECT ProductName, ProductID FROM Products";
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand(selectSQL, con);

        // Open the connection.
        con.Open();

        // Define the binding.
        lstProduct.DataSource = cmd.ExecuteReader();
        lstProduct.DataTextField = "ProductName";
        lstProduct.DataValueField = "ProductID";

        // Activate the binding.
        this.DataBind();

        con.Close();

        // Make sure nothing is currently selected in the list box.
        lstProduct.SelectedIndex = -1;
    }
}
```

Once again, the list is filled only the first time the page is requested (and stored in view state automatically). If the page is posted back, the list keeps its current entries. This reduces the amount of database work, and keeps the page working quickly and efficiently. You should also note that this page doesn’t attempt to deal with errors. If you were using it in a real application, you’d need to use the exception-handling approach demonstrated in Chapter 14.

The actual database code is similar to what was used in the previous chapter. The example uses a Select statement but carefully limits the returned information to just the ProductName and ProductID fields, which are the only pieces of information it will use. The resulting window lists all the products defined in the database, as shown in Figure 15-9.

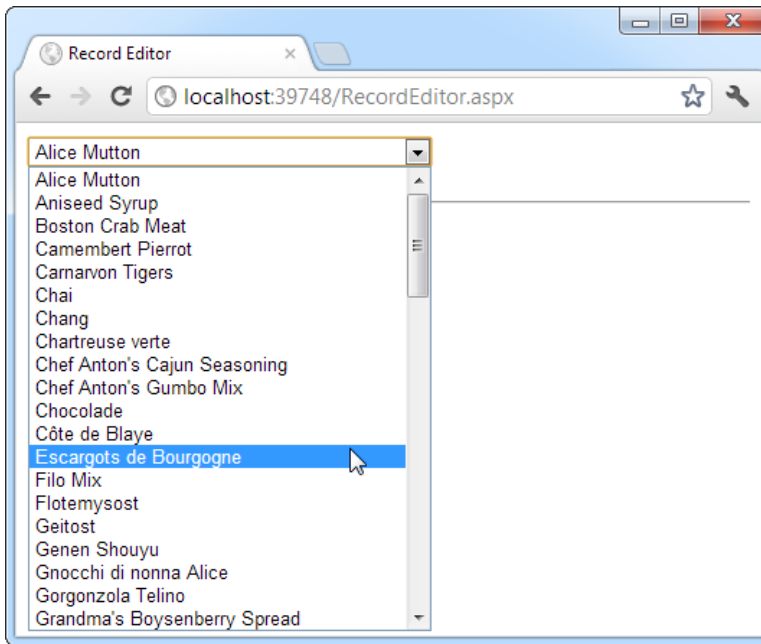


Figure 15-9. Product choices

The drop-down list enables `AutoPostBack`, so as soon as the user makes a selection, a `lstProduct.SelectedItemChanged` event fires. At this point, your code performs the following tasks:

- It reads the corresponding record from the `Products` table and displays additional information about it in a label. In this case, a `Join` query links information from the `Products` and `Categories` tables. The code also determines what the category is for the current product. This is the piece of information it will allow the user to change.
- It reads the full list of `CategoryNames` from the `Categories` table and binds this information to a different list control. Initially, this list is hidden in a panel with its `Visible` property set to `false`. The code reveals the content of this panel by setting `Visible` to `true`.
- It highlights the row in the category list that corresponds to the current product. For example, if the current product is a `Seafood` category, the `Seafood` entry in the list box will be selected.

This logic appears fairly involved, but it's really just an application of what you've learned over the past two chapters. The full listing is as follows:

```
protected void lstProduct_SelectedIndexChanged(object sender, EventArgs e)
{
    // Create a command for selecting the matching product record.
    string selectProduct = "SELECT ProductName, QuantityPerUnit, " +
        "CategoryName FROM Products INNER JOIN Categories ON " +
        "Categories.CategoryID=Products.CategoryID " +
        "WHERE ProductID=@ProductID";
```

```

// Create the Connection and Command objects.
SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmdProducts = new SqlCommand(selectProduct, con);
cmdProducts.Parameters.AddWithValue("@ProductID",
    lstProduct.SelectedItem.Value);

// Retrieve the information for the selected product.
using (con)
{
    con.Open();
    SqlDataReader reader = cmdProducts.ExecuteReader();
    reader.Read();

    // Update the display.
    lblRecordInfo.Text = "<b>Product:</b> " +
        reader["ProductName"] + "<br />";
    lblRecordInfo.Text += "<b>Quantity:</b> " +
        reader["QuantityPerUnit"] + "<br />";
    lblRecordInfo.Text += "<b>Category:</b> " + reader["CategoryName"];

    // Store the corresponding CategoryName for future reference.
    string matchCategory = reader["CategoryName"].ToString();

    // Close the reader.
    reader.Close();

    // Create a new Command for selecting categories.
    string selectCategory = "SELECT CategoryName, " +
        "CategoryID FROM Categories";
    SqlCommand cmdCategories = new SqlCommand(selectCategory, con);

    // Retrieve the category information and bind it.
    lstCategory.DataSource = cmdCategories.ExecuteReader();
    lstCategory.DataTextField = "CategoryName";
    lstCategory.DataValueField = "CategoryID";
    lstCategory.DataBind();

    // Highlight the matching category in the list.
    lstCategory.Items.FindByText(matchCategory).Selected = true;
}

pnlCategory.Visible = true;
}

```

You could improve this code in several ways. It probably makes the most sense to remove these data access routines from this event handler and put them into more-generic functions. For example, you could use a function that accepts a ProductID and returns a single DataRow with the associated product information. Another improvement would be to use a stored procedure to retrieve this information.

The end result is a window that updates itself dynamically whenever a new product is selected, as shown in Figure 15-10.

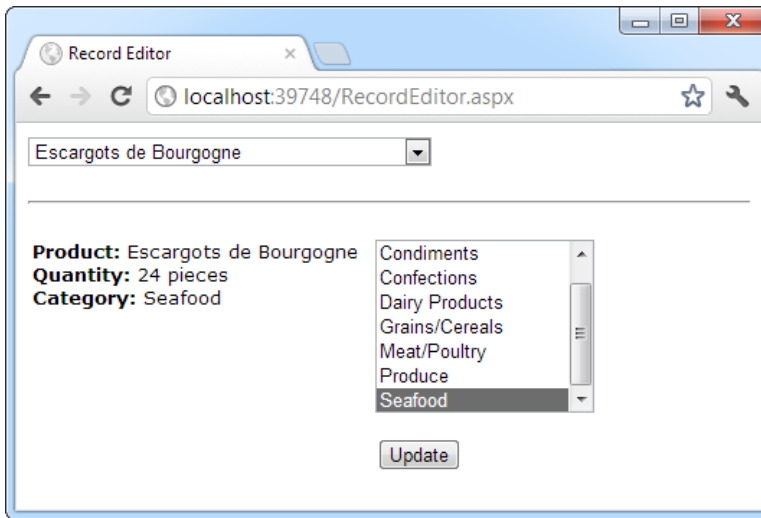


Figure 15-10. Product information

This example still has one more trick in store. If the user selects a different category and clicks Update, the change is made in the database. Of course, this means creating new Connection and Command objects, as follows:

```
protected void cmdUpdate_Click(object sender, EventArgs e)
{
    // Define the Command.
    string updateCommand = "UPDATE Products " +
        "SET CategoryID=@CategoryID WHERE ProductID=@ProductID";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(updateCommand, con);

    cmd.Parameters.AddWithValue("@CategoryID", lstCategory.SelectedItem.Value);
    cmd.Parameters.AddWithValue("@ProductID", lstProduct.SelectedItem.Value);

    // Perform the update.
    using (con)
    {
        con.Open();
        cmd.ExecuteNonQuery();
    }
}
```

You could easily extend this example so that it allows you to edit all the properties in a product record. But before you try that, you might want to experiment with the rich data controls that are shown in the next chapter. Using these controls, you can create sophisticated lists and grids that provide automatic features for selecting, editing, and deleting records.

Working with Data Source Controls

In Chapter 14, you saw how to directly connect to a database, execute a query, loop through the records in the result set, and display them on a page. In this chapter, you've already seen a simpler option—with data binding, you can write your data access logic and then show the results in the page with no looping or control manipulation required. Now it's time to introduce *another* convenience: data source controls. Amazingly enough, data source controls allow you to create data-bound pages without writing any data access code at all.

■ **Note** As you'll soon see, often a gap exists between what you *can* do and what you *should* do. In most professional applications, you'll need to write and fine-tune your data access code for optimum performance or access to specific features. That's why you've spent so much time learning how ADO.NET works, rather than jumping straight to the data source controls.

The data source controls include any control that implements the *IDataSource* interface. The .NET Framework includes the following data source controls:

SqlDataSource: This data source allows you to connect to any data source that has an ADO.NET data provider. This includes SQL Server, Oracle, and OLE DB or ODBC data sources. When using this data source, you don't need to write the data access code.

AccessDataSource: This data source allows you to read and write the data in an Access database file (.mdb). However, its use is discouraged, because Access doesn't scale well to large numbers of users (unlike SQL Server Express).

■ **Note** Access databases do not have a dedicated server engine (such as SQL Server) that coordinates the actions of multiple people and ensures that data won't be lost or corrupted. For that reason, Access databases are best suited for very small websites, where few people need to manipulate data at the same time. A much better small-scale data solution is SQL Server Express, which is described in Chapter 14.

ObjectDataSource: This data source allows you to connect to a custom data access class. This is the preferred approach for large-scale professional web applications, but it forces you to write much more code. You'll study the *ObjectDataSource* in Chapter 22.

XmlDataSource: This data source allows you to connect to an XML file. You'll learn more about XML in Chapter 18.

SiteMapDataSource: This data source allows you to connect to a .sitemap file that describes the navigational structure of your website. You saw this data source in Chapter 13.

EntityDataSource: This data source allows you to query a database by using the LINQ to Entities feature, which you'll tackle in Chapter 24.

LinqDataSource: This data source allows you to query a database by using the LINQ to SQL feature, which is a similar (but somewhat less powerful) predecessor to LINQ to Entities.

You can find all the data source controls in the Data tab of the Toolbox in Visual Studio, with the exception of the `AccessDataSource`.

When you drop a data source control onto your web page, it shows up as a gray box in Visual Studio. However, this box won't appear when you run your web application and request the page (see Figure 15-11).

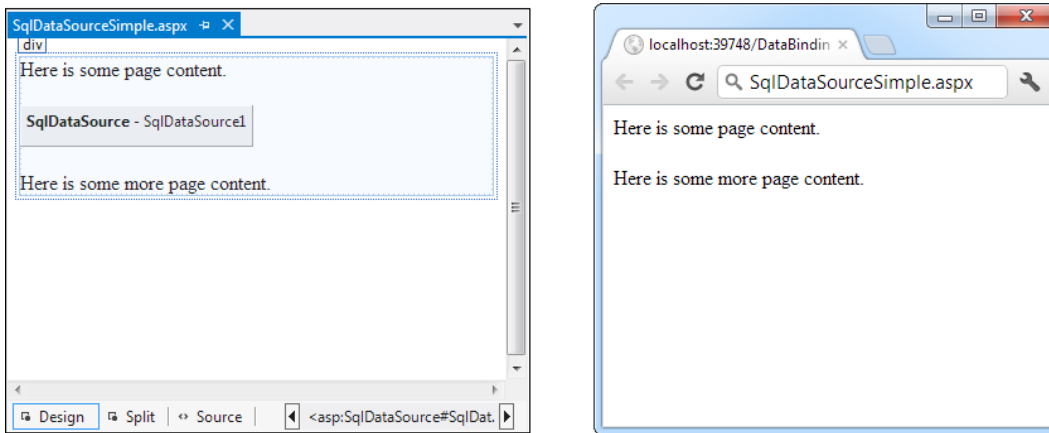


Figure 15-11. A data source control at design time and runtime

If you perform more than one data access task in the same page (for example, you need to be able to query two different tables), you'll need more than one data source control.

The Page Life Cycle with Data Binding

Data source controls can perform two key tasks:

- They can retrieve data from a data source and supply it to bound controls. When you use this feature, your bound controls are automatically filled with data. You don't even need to call `DataBind()`.
- They can update the data source when edits take place. In order to use this feature, you must use one of ASP.NET's rich data controls, such as the `GridView` or `DetailsView`. For example, if you make an edit in the `GridView` and click `Update`, the `GridView` will trigger the update in the data source control, and the data source control will then update the database.

Before you can use the data source controls, you need to understand the page life cycle. The following steps explain the sequence of stages your page goes through in its lifetime. The two steps in bold (4 and 6) indicate the points where the data source controls will spring into action:

1. The page object is created (based on the .aspx file).
2. The page life cycle begins, and the `Page.Init` and `Page.Load` events fire.
3. All other control events fire.

4. **If the user is applying a change, the data source controls perform their update operations now. If a row is being updated, the Updating and Updated events fire. If a row is being inserted, the Inserting and Inserted events fire. If a row is being deleted, the Deleting and Deleted events fire.**
5. The Page.PreRender event fires.
6. **The data source controls perform their queries and insert the data they retrieve into the bound controls. This step happens the first time your page is requested and every time the page is posted back, ensuring that you always have the most up-to-date data. The Selecting and Selected events fire at this point.**
7. The page is rendered and disposed.

In the rest of this chapter, you'll take a closer look at the `SqlDataSource` control, and you'll use it to build the record editor example demonstrated earlier—with a lot less code.

The SqlDataSource

Data source controls turn up in the .aspx markup portion of your web page like ordinary controls. Here's an example:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server" ... />
```

The `SqlDataSource` represents a database connection that uses an ADO.NET provider. However, this has a catch. The `SqlDataSource` needs a generic way to create the `Connection`, `Command`, and `DataReader` objects it requires. This is possible only if your data provider includes something called a *data provider factory*. The factory has the responsibility of creating the provider-specific objects that the `SqlDataSource` needs to access the data source. Fortunately, .NET includes a data provider factory for each of its four data providers:

- `System.Data.SqlClient`
- `System.Data.OracleClient`
- `System.Data.OleDb`
- `System.Data.Odbc`

You can use all of these providers with the `SqlDataSource`. You choose your data source by setting the provider name. Here's a `SqlDataSource` that connects to a SQL Server database by using the SQL Server provider:

```
<asp:SqlDataSource ProviderName="System.Data.SqlClient" ... />
```

Technically, you can omit this piece of information, because the `System.Data.SqlClient` provider factory is the default.

■ Note If you have an up-to-date third-party provider (such as ODP.NET for accessing Oracle databases), it will also include a provider factory that allows you to use it with the `SqlDataSource`.

The next step is to supply the required connection string—without it, you cannot make any connections. Although you can hard-code the connection string directly in the `SqlDataSource` tag, keeping it in the `<connectionStrings>` section of the web.config file is always better, to guarantee greater flexibility and ensure that you won't inadvertently change the connection string.

To refer to a connection string in your .aspx markup, you use a special syntax in this format:

```
<%$ ConnectionStrings:[NameOfConnectionString] %>
```

This looks like a data-binding expression, but it's slightly different. (For one thing, it begins with the character sequence <%\$ instead of <%#.)

For example, if you have a connection string named Northwind in your web.config file that looks like this:

```
<configuration>
  <connectionStrings>
    <add name="Northwind" connectionString=
"Data Source=(localdb)\v11.0;Initial Catalog=Northwind;Integrated
Security=SSPI" />
  </connectionStrings>
  ...
</configuration>
```

you would specify it in the SqlDataSource by using this syntax:

```
<asp:SqlDataSource ConnectionString="<%$ ConnectionStrings:Northwind %>" ... />
```

After you've specified the provider name and connection string, the next step is to add the query logic that the SqlDataSource will use when it connects to the database.

Tip If you want some help creating your connection string, select the SqlDataSource, open the Properties window, and select the ConnectionString property. A drop-down arrow will appear at the right side of the value. If you click that drop-down arrow, you'll see a list of all the connection strings in your web.config file. You can pick one of these connections, or you can choose New Connection (at the bottom of the list) to open the Add Connection dialog box, where you can pick the database you want. Best of all, if you create a new connection, Visual Studio copies the connection string into your web.config file, so you can reuse it with other SqlDataSource objects.

Selecting Records

You can use each SqlDataSource control you create to retrieve a single query. Optionally, you can also add corresponding commands for deleting, inserting, and updating rows. For example, one SqlDataSource is enough to query and update the Customers table in the Northwind database. However, if you need to independently retrieve or update Customers and Orders information, you'll need two SqlDataSource controls.

The SqlDataSource command logic is supplied through four properties—SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand—each of which takes a string. The string you supply can be inline SQL (in which case the corresponding SelectCommandType, InsertCommandType, UpdateCommandType, or DeleteCommandType property should be Text, the default) or the name of a stored procedure (in which case the command type is StoredProcedure). You need to define commands only for the types of actions you want to perform. In other words, if you're using a data source for read-only access to a set of records, you need to define only the SelectCommand property.

Note If you configure a command in the Properties window, you'll see a property named `SelectQuery` instead of `SelectCommand`. `SelectQuery` is a virtual property that's displayed as a design-time convenience. When you edit `SelectQuery` (by clicking the ellipsis next to the property name), you can use a special designer to write the command text (the `SelectCommand`) and add the command parameters (the `SelectParameters`) at the same time. However, this tool works best after you've reviewed the examples in this section, and you understand the way the `SelectCommand` and `SelectParameters` properties really work.

Here's a complete `SqlDataSource` that defines a `Select` command for retrieving product information from the `Products` table:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
    ConnectionString="<%= $ConnectionStrings:Northwind %>"
    SelectCommand="SELECT ProductName, ProductID FROM Products"
/>
```

Tip You can write the data source logic by hand, or you can use a design-time wizard that lets you create a connection and create the command logic in a graphical query builder. You can even test the query as you build it to make sure it returns the records you want. To launch this tool, select the data source control on the design surface of your page, and click the `Configure Data Source` link in the smart tag.

This is enough to build the first stage of the record editor example shown earlier—namely, the drop-down list box that shows all the products. All you need to do is set the `DataSourceID` property to point to the `SqlDataSource` you've created. The easiest way to do this is by using the Properties window, which provides a drop-down list of all the data sources on your current web page. At the same time, make sure you set the `DataTextField` and `DataValueField` properties. After you make these changes, you'll wind up with a control tag like this:

```
<asp:DropDownList ID="lstProduct" runat="server" AutoPostBack="True"
    DataSourceID="sourceProducts" DataTextField="ProductName"
    DataValueField="ProductID" />
```

The best part about this example is that you don't need to write any code. When you run the page, the `DropDownList` control asks the `SqlDataSource` for the data it needs. At this point, the `SqlDataSource` executes the query you defined, fetches the information, and binds it to the `DropDownList`. The whole process unfolds automatically.

How the Data Source Controls Work

As you learned earlier in this chapter, you can bind to a `DataReader` or a `DataSet`. So it's worth asking—which approach does the `SqlDataSource` control use? It's actually your choice, depending on whether you set the `DataSourceMode` to `SqlDataSourceMode.DataSet` (the default) or to `SqlDataSourceMode.DataReader`. The `DataSet` mode is almost always better, because it supports advanced sorting, filtering, and caching settings that depend on the `DataSet`. All these features are disabled in `DataReader` mode. However, you can use the `DataReader` mode with extremely large grids, because it's more memory-efficient. That's because the `DataReader` holds only one record in memory at a time—just long enough to copy the record's information to the linked control.

Another important fact to understand about the data source controls is that when you bind more than one control to the same data source, you cause the query to be executed multiple times. For example, if two controls are bound to the same data source, the data source control performs its query twice—once for each control. This is somewhat inefficient—after all, if you wrote the data-binding code yourself by hand, you’d probably choose to perform the query once and then bind the returned *DataSet* twice. Fortunately, this design isn’t quite as bad as it might seem.

First, you can avoid this multiple-query overhead by using caching, which allows you to store the retrieved data in a temporary memory location where it will be reused automatically. The *SqlDataSource* supports automatic caching if you set *EnableCaching* to true. Chapter 23 provides a full discussion of how caching works and how you can use it with the *SqlDataSource*.

Second, contrary to what you might expect, most of the time you *won’t* be binding more than one control to a data source. That’s because the rich data controls you’ll learn about in Chapter 16—the *GridView*, *DetailsView*, and *FormsView*—have the ability to present multiple pieces of data in a flexible layout. If you use these controls, you’ll need to bind only one control, which allows you to steer clear of this limitation.

It’s also important to remember that data binding is performed at the end of your web page processing, just before the page is rendered. This means the *Page.Load* event will fire, followed by any control events, followed by the *Page.PreRender* event. Only then will the data binding take place.

Parameterized Commands

In the previous example (which used the *SqlDataSource* to retrieve a list of products), the complete query was hard-coded. Often you won’t have this flexibility. Instead, you’ll want to retrieve a subset of data, such as all the products in a given category or all the employees in a specific city.

The record editor that you considered earlier offers an ideal example. After you select a product, you want to execute another command to get the full details for that product. (You might just as easily execute another command to get records that are related to this product.) To make this work, you need two data sources. You’ve already created the first *SqlDataSource*, which fetches limited information about every product. Here’s the second *SqlDataSource*, which gets more-extensive information about a single product (the following query is split over several lines to fit the printed page):

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%= ConnectionStrings:Northwind %>"
  SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID"
/>
```

But this example has a problem. It defines a parameter (*@ProductID*) that identifies the ID of the product you want to retrieve. How do you fill in this piece of information? It turns out you need to add a *<SelectParameters>* section to the *SqlDataSource* tag. Inside this section, you must define each parameter that’s referenced by your *SelectCommand* and tell the *SqlDataSource* where to find the value it should use. You do that by *mapping* the parameter to a value in a control.

Here’s the corrected command:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%= ConnectionStrings:Northwind %>"
  SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID"
  <SelectParameters>
    <asp:ControlParameter ControlID="lstProduct" Name="ProductID"
      PropertyName="SelectedValue" />
  </SelectParameters>
</asp:SqlDataSource>
```

You always indicate parameters with an @ symbol, as in @City. You can define as many parameters as you want, but you must map each one to a value by using a separate element in the SelectParameters collection. In this example, the value for the @ProductID parameter comes from the `lstProduct.SelectedValue` property. In other words, you are binding a value that's currently in a control to place it into a database command. (You could also use the `SelectedText` property to get the currently displayed text, which is the `ProductName` in this example.)

Now all you need to do is bind the `SqlDataSource` to the remaining controls where you want to display information. This is where the example takes a slightly different turn. In the previous version of the record editor, you took the information and used a combination of values to fill in details in a label and a list control. This type of approach doesn't work well with data source controls. First, you can bind only a single data field to most simple controls such as lists. Second, each bound control makes a separate request to the `SqlDataSource`, triggering a separate database query. This means that if you bind a dozen controls, you'll perform the same query a dozen times, with terrible performance. You can alleviate this problem with data source caching (see Chapter 23), but that would indicate you aren't designing your application in a way that lends itself well to the data source control model.

The solution is to use one of the rich data controls, such as the `GridView`, `DetailsView`, or `FormView`. These controls have the smarts to show multiple fields at once, in a highly flexible layout. You'll learn about these three controls in detail in the next chapter, but the following example shows a simple demonstration of how to use the `DetailsView`.

The `DetailsView` is a rich data control that's designed to show multiple fields in a data source. As long as its `AutoGenerateRows` is true (the default), it creates a separate row for each field, with the field caption and value. Figure 15-12 shows the result.

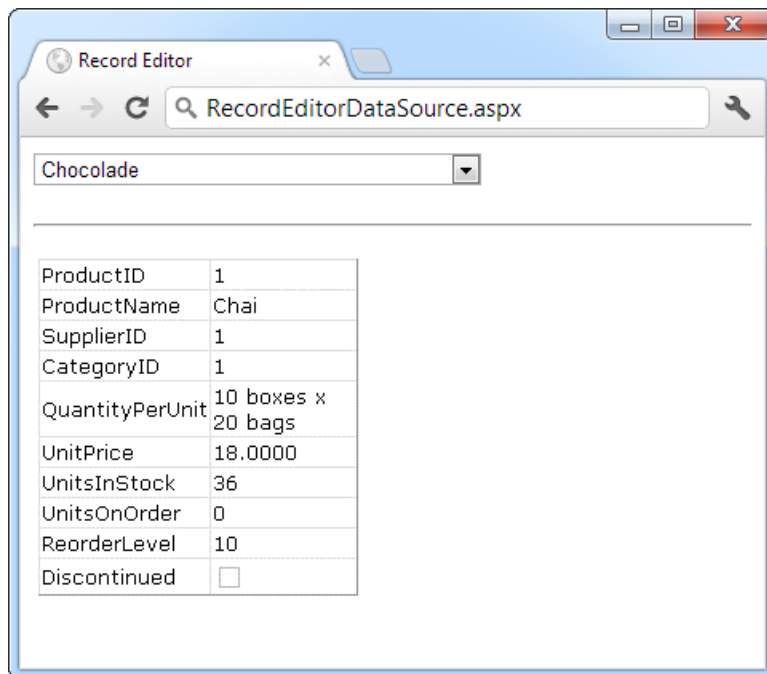


Figure 15-12. Displaying full product information in a `DetailsView`

Here's the basic `DetailsView` tag that makes this possible:

```
<asp:DetailsView ID="detailsProduct" runat="server"
    DataSourceID="sourceProductDetails" />
```

As you can see, the only property you need to set is `DataSourceID`. That binds the `DetailsView` to the `SqlDataSource` you created earlier. This `SqlDataSource` gets the full product information for a single row, based on the selection in the list control. Best of all, this whole example still hasn't required a line of code.

Other Types of Parameters

In the previous example, the `@ProductID` parameter in the second `SqlDataSource` is configured based on the selection in a drop-down list. This type of parameter, which links to a property in another control, is called a *control parameter*. But parameter values aren't necessarily drawn from other controls. You can map a parameter to any of the parameter types defined in Table 15-1.

Table 15-1. *Parameter Types*

Source	Control Tag	Description
Control property	<code><asp:ControlParameter></code>	A property from another control on the page.
Query string value	<code><asp:QueryStringParameter></code>	A value from the current query string.
Session state value	<code><asp:SessionParameter></code>	A value stored in the current user's session.
Cookie value	<code><asp:CookieParameter></code>	A value from any cookie attached to the current request.
Profile value	<code><asp:ProfileParameter></code>	A value from the current user's profile (see Chapter 21 for more about profiles).
Routed URL value	<code><asp:RouteParameter></code>	A value from a routed URL. Routed URLs are an advanced technique that lets you map any URL to a different page (so a request such as http://www.mysite.com/products/112 redirects to the page www.mysite.com/productdetails.aspx?id=112 , for example). To learn more about URL routing, refer to the Visual Studio Help or <i>Pro ASP.NET 4.5 in C#</i> (Apress).
A form variable	<code><asp:FormParameter></code>	A value posted to the page from an input control. Usually, you'll use a control property instead, but you might need to grab a value straight from the <code>Forms</code> collection if you've disabled view state for the corresponding control.

For example, you could split the earlier example into two pages. In the first page, define a list control that shows all the available products:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%= ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductName, ProductID FROM Products"
/>
<asp:DropDownList ID="lstProduct" runat="server" AutoPostBack="True"
  DataSourceID="sourceProducts" DataTextField="ProductName"
  DataValueField="ProductID" />
```

Now you'll need a little extra code to copy the selected product to the query string and redirect the page. Here's a button that does just that:

```
protected void cmdGo_Click(object sender, EventArgs e)
{
    if (lstProduct.SelectedIndex != -1)
    {
        Response.Redirect(
            "QueryParameter2.aspx?prodID=" + lstProduct.SelectedValue);
    }
}
```

Finally, the second page can bind the DetailsView according to the ProductID value that's supplied in the query string:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID">
    <SelectParameters>
        <asp:QueryStringParameter Name="ProductID" QueryStringField="prodID" />
    </SelectParameters>
</asp:SqlDataSource>

<asp:DetailsView ID="detailsProduct" runat="server"
    DataSourceID="sourceProductDetails" />
```

Setting Parameter Values in Code

Sometimes you'll need to set a parameter with a value that isn't represented by any of the parameter classes in Table 15-1. Or you might want to manually modify a parameter value before using it. In both of these scenarios, you need to use code to set the parameter value just before the database operation takes place.

For example, consider the page shown in Figure 15-13. It includes two data-bound controls. The first is a list of all the customers in the database. Here's the markup that defines the list and its data source:

```
<asp:SqlDataSource ID="sourceCustomers" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT CustomerID, ContactName FROM Customers
    OrderBy ContactName" />
    <asp:DropDownList ID="lstCustomers" runat="server"
        DataSourceID="sourceCustomers" DataTextField="ContactName"
        DataValueField="CustomerID" AutoPostBack="True">
</asp:DropDownList>
```

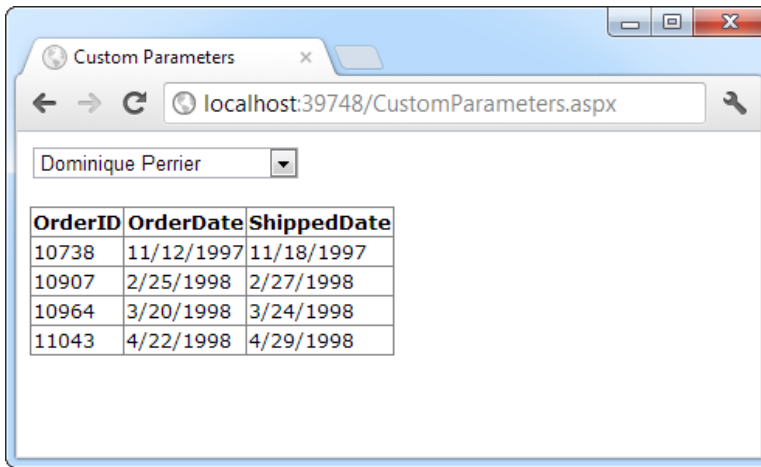


Figure 15-13. Using parameters in a master-details page

When the user picks a customer from the list, the page is posted back (because `AutoPostBack` is set to `true`) and the matching orders are shown in a `GridView` underneath, using a second data source. This data source pulls the `CustomerID` for the currently selected customer from the drop-down list by using a `ControlParameter`:

```
<asp:SqlDataSource ID="sourceCustomers" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%$ ConnectionStrings:Northwind %%"
  SelectCommand="SELECT OrderID,OrderDate,ShippedDate FROM Orders WHERE
CustomerID=@CustomerID">
  <SelectParameters>
    <asp:ControlParameter Name="CustomerID"
      ControlID="lstCustomers" PropertyName="SelectedValue" />
  </SelectParameters>
</asp:SqlDataSource>

<asp:GridView ID="gridOrders" runat="server" DataSourceID="sourceOrders">
</asp:GridView>
```

Now imagine that you want to limit the order list so it shows only orders made in the last week. This is easy enough to accomplish with a `Where` clause that examines the `OrderDate` field. But there's a catch. It doesn't make sense to hard-code the `OrderDate` value in the query itself, because the range is set based on the current date. And there's no parameter that provides exactly the information you need. The easiest way to solve this problem is to add a new parameter—one that you'll be responsible for setting yourself:

```
<asp:SqlDataSource ID="sourceOrders" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%$ ConnectionStrings:Northwind %%"
  SelectCommand="SELECT OrderID,OrderDate,ShippedDate FROM Orders WHERE
CustomerID=@CustomerID AND OrderDate>=@EarliestOrderDate"
  OnSelecting="sourceOrders_Selecting">
  <SelectParameters>
    <asp:ControlParameter Name="CustomerID"
      ControlID="lstCustomers" PropertyName="SelectedValue" />
```

```

<asp:Parameter Name="EarliestOrderDate" Type="DateTime"
  DefaultValue="1900/01/01" />
</SelectParameters>
</asp:SqlDataSource>

```

Although you can modify the value of any parameter, if you aren't planning to pull the value out of any of the places listed in Table 15-1, it makes sense to use an ordinary Parameter object, as represented by the <asp:Parameter> element. You can set the data type (if required) and the default value (as demonstrated in this example).

Now that you've created the parameter, you need to set its value before the command takes place. The SqlDataSource has a number of events that are perfect for setting parameter values. You can fill in parameters for a select operation by reacting to the Selecting event. Similarly, you can use the Updating, Deleting, and Inserting events when updating, deleting, or inserting a record. In these event handlers, you can access the command that's about to be executed, using the Command property of the custom EventArgs object (for example, SqlDataSourceSelectingEventArgs.Command). You can then modify its parameter values by hand. The SqlDataSource also provides similarly named Selected, Updated, Deleted, and Inserted events, but these take place after the operation has been completed, so it's too late to change the parameter value.

Here's the code that's needed to set the parameter value to a date that's seven days in the past, ensuring that you see one week's worth of records:

```

protected void sourceOrders_Selecting(object sender,
  SqlDataSourceSelectingEventArgs e)
{
    e.Command.Parameters["@EarliestOrderDate"].Value =
        DateTime.Today.AddDays(-7);
}

```

Note You'll have to tweak this code slightly if you're using it with the standard Northwind database. The data in the Northwind database is historical, and most orders bear dates around 1997. As a result, the previous code won't actually retrieve any records. But if you use the AddYears() method instead of AddDays(), you can easily move back 13 years or more, to the place you need to be.

Handling Errors

When you deal with an outside resource such as a database, you need to protect your code with a basic amount of error-handling logic. Even if you've avoided every possible coding mistake, you still need to defend against factors outside your control—for example, if the database server isn't running or the network connection is broken.

You can count on the SqlDataSource to properly release any resources (such as connections) if an error occurs. However, the underlying exception won't be handled. Instead, it will bubble up to the page and derail your processing. As with any other unhandled exception, the user will receive a cryptic error message or an error page. This design is unavoidable—if the SqlDataSource suppressed exceptions, it could hide potential problems and make debugging extremely difficult. However, it's a good idea to handle the problem in your web page and show a more suitable error message.

To do this, you need to handle the data source event that occurs immediately *after* the error. If you're performing a query, that's the Selected event. If you're performing an update, delete, or insert operation, you would handle the Updated, Deleted, or Inserted event instead. (If you don't want to offer customized error messages, you could handle all these events with the same event handler.)

In the event handler, you can access the exception object through the SqlDataSourceStatusEventArgs.Exception property. If you want to prevent the error from spreading any further, simply set the SqlDataSourceStatusEventArgs.ExceptionHandled property to true. Then make sure you show an appropriate error message on your web page to inform the user that the command was not completed.

Here's an example:

```
protected void sourceProducts_Selected(object sender,
    SqlDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        lblError.Text = "An exception occurred performing the query.";

        // Consider the error handled.
        e.ExceptionHandled = true;
    }
}
```

Updating Records

Selecting data is only half the equation. The `SqlDataSource` can also apply changes. The only catch is that not all controls support updating. For example, the humble `ListBox` doesn't provide any way for the user to edit values, delete existing items, or insert new ones. Fortunately, ASP.NET's rich data controls—including the `GridView`, `DetailsView`, and `FormView`—all have editing features you can switch on.

Before you can switch on the editing features in a given control, you need to define suitable commands for the operations you want to perform in your data source. That means supplying commands for inserting (`InsertCommand`), deleting (`DeleteCommand`), and updating (`UpdateCommand`). If you know you will allow the user to perform only certain operations (such as updates) but not others (such as insertions and deletions), you can safely omit the commands you don't need.

You define `InsertCommand`, `DeleteCommand`, and `UpdateCommand` in the same way you define the command for the `SelectCommand` property—by using a parameterized query. For example, here's a revised version of the `SqlDataSource` for product information that defines a basic update command to update every field:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %%"
    SelectCommand="SELECT ProductID, ProductName, UnitPrice, UnitsInStock,
UnitsOnOrder, ReorderLevel, Discontinued FROM Products WHERE ProductID=@ProductID"
    UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
Discontinued=@Discontinued WHERE ProductID=@ProductID">
    <SelectParameters>
        <asp:ControlParameter ControlID="lstProduct" Name="ProductID"
            PropertyName="SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```

In this example, the parameter names aren't chosen arbitrarily. As long as you give each parameter the same name as the field it affects, and preface it with the `@` symbol (so `ProductName` becomes `@ProductName`), you don't need to define the parameter. That's because the ASP.NET data controls automatically submit a collection of parameters with the new values before triggering the update. Each parameter in the collection uses this naming convention, which is a major time-saver.

You also need to give the user a way to enter the new values. Most rich data controls make this fairly easy—with the `DetailsView`, it's simply a matter of setting the `AutoGenerateEditButton` property to `true`, as shown here:

```
<asp:DetailsView ID="DetailsView1" runat="server"
    DataSourceID="sourceProductDetails" AutoGenerateEditButton="True" />
```


Now when you run the page, you'll see an edit link. When clicked, this link switches the DetailsView into edit mode. All fields are changed to edit controls (typically text boxes), and the Edit link is replaced with an Update link and a Cancel link (see Figure 15-14).

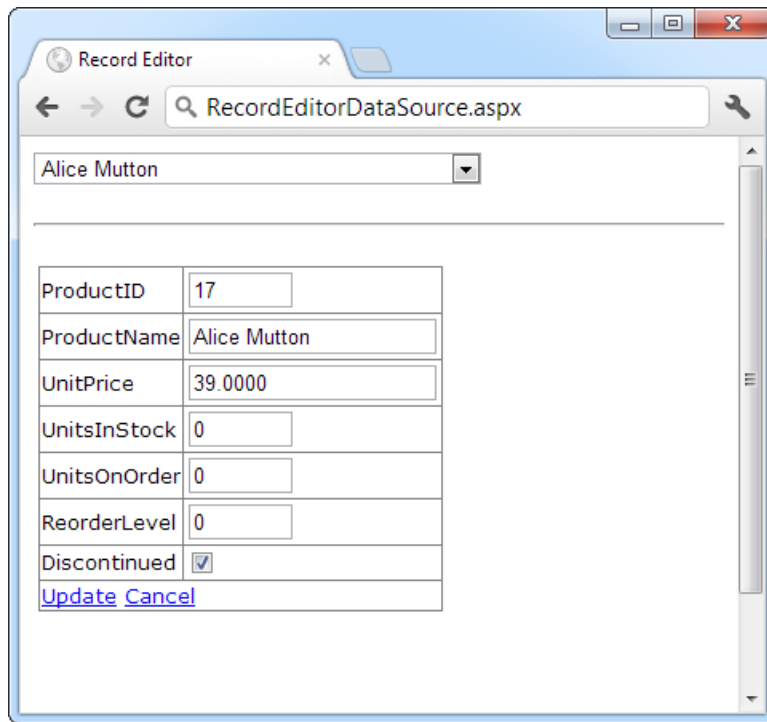


Figure 15-14. *Editing with the DetailsView*

Clicking the Cancel link returns the row to its initial state. Clicking the Update link triggers an update. The DetailsView extracts the field values, uses them to set the parameters in the `SqlDataSource.UpdateParameters` collection, and then triggers the `SqlDataSource.UpdateCommand` to apply the change to the database. Once again, you don't have to write any code.

You can create similar parameterized commands for `DeleteCommand` and `InsertCommand`. To enable deleting and inserting, you need to set the `AutoGenerateDeleteButton` and `AutoGenerateInsertButton` properties of the DetailsView to true. To see a sample page that allows updating, deleting, and inserting, refer to the `UpdateDeleteInsert.aspx` page that's included with the downloadable samples for this chapter.

Strict Concurrency Checking

The update command in the previous example matches the record based on its ID. You can tell this by examining the Where clause:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
    UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
    Discontinued=@Discontinued WHERE ProductID=@ProductID"
```

The problem with this approach is that it opens the door to an update that overwrites the changes of another user, if these changes are made between the time your page is requested and the time your page commits its update.

For example, imagine Chen and Lucy are viewing the same table of product records. Lucy commits a change to the price of a product. A few seconds later, Chen commits a name change to the same product record. Chen's update command not only applies the new name but also overwrites all the other fields with the values from Chen's page—replacing the price Lucy entered with the price from the original page.

One way to solve this problem is to use an approach called *match-all-values* concurrency. In this situation, your update command attempts to match every field. As a result, if the original record has changed, the update command won't find it and the update won't be performed at all. So in the scenario described previously, using the match-all-values strategy, Chen would receive an error when he attempts to apply the new product name, and he would need to edit the record and apply the change again.

To use this approach, you need to add a Where clause that tries to match every field. Here's what the modified command would look like:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
Discontinued=@Discontinued WHERE ProductID=@ProductID AND
ProductName=@original_ProductName AND UnitPrice=@original_UnitPrice AND
UnitsInStock=@original_UnitsInStock AND UnitsOnOrder=@original_UnitsOnOrder AND
ReorderLevel=@original_ReorderLevel AND Discontinued=@original_Discontinued"
```

Although this makes sense conceptually, you're not finished yet. Before this command can work, you need to tell the `SqlDataSource` to maintain the old values from the data source and to give them parameter names that start with `original_`. You do this by setting two properties. First, set the `SqlDataSource.ConflictDetection` property to `ConflictOptions.CompareAllValues` instead of `ConflictOptions.OverwriteChanges` (the default). Next, set the long-winded `OldValuesParameterFormatString` property to the text `"original_{0}"`. This tells the `SqlDataSource` to insert the text *original_* before the field name to create the parameter that stores the old value. Now your command will work as written.

The `SqlDataSource` doesn't raise an exception to notify you if no update is performed. So, if you use the command shown in this example, you need to handle the `SqlDataSource.Updated` event and check the `SqlDataSource.StatusEventArgs.AffectedRows` property. If it's 0, no records have been updated, and you should notify the user about the concurrency problem so the update can be attempted again, as shown here:

```
protected void sourceProductDetails_Updated(object sender,
    SqlDataSourceStatusEventArgs e)
{
    if (e.AffectedRows == 0)
    {
        lblInfo.Text = "No update was performed. " +
            "A concurrency error is likely, or the command is incorrectly written.";
    }
    else
    {
        lblInfo.Text = "Record successfully updated.";
    }
}
```

Figure 15-15 shows the result you'll get if you run two copies of this page in two separate browser windows, begin editing in both of them, and then try to commit both updates.

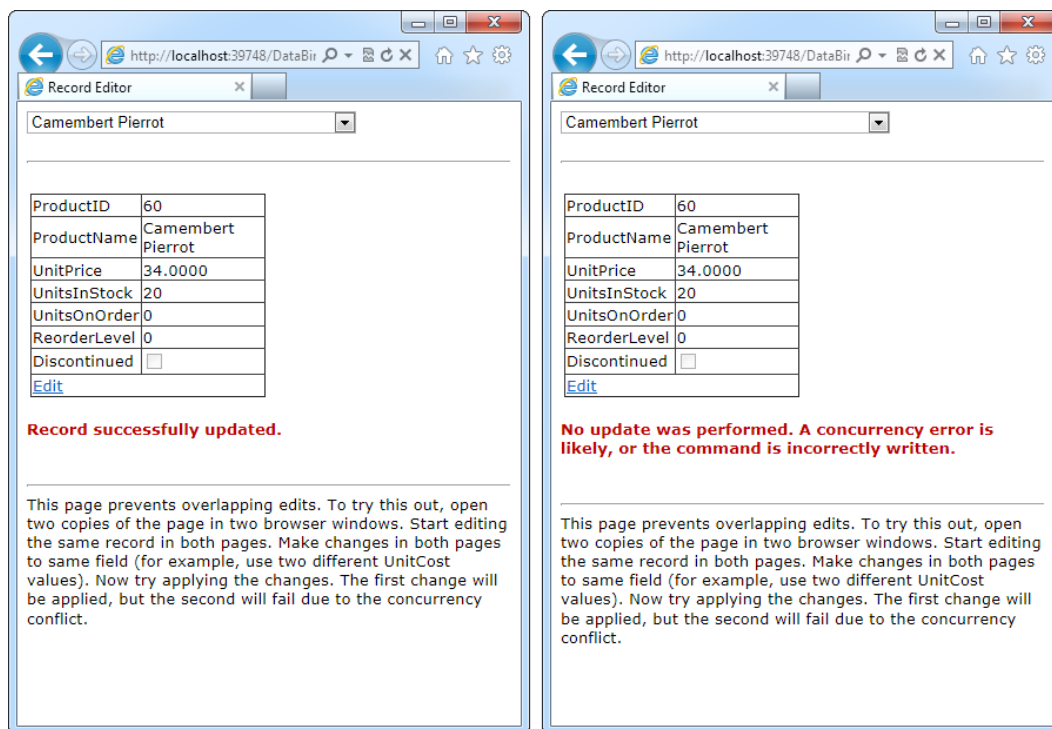


Figure 15-15. A concurrency error in action

Matching every field is an acceptable approach for small records, but it isn't the most efficient strategy if you have tables with huge amounts of data. In this situation, you have two possible solutions: you can match some of the fields (leaving out the ones with really big values) or you can add a timestamp field to your database table, and use that for concurrency checking.

Timestamps are special fields that the database uses to keep track of the state of a record. Whenever any change is made to a record, the database engine updates the timestamp field, giving it a new, automatically generated value. The purpose of a timestamp field is to make strict concurrency checking easier. When you attempt to perform an update to a table that includes a timestamp field, you use a Where clause that matches the appropriate unique ID value (for example, ProductID) and the timestamp field:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder,
ReorderLevel=@ReorderLevel, Discontinued=@Discontinued
WHERE ProductID=@ProductID AND RowTimestamp=@RowTimestamp"
```

The database engine uses the ProductID to look up the matching record. Then it attempts to match the timestamp in order to update the record. If the timestamp matches, you know the record hasn't been changed. The actual *value* of the timestamp isn't important, because that's controlled by the database. You just need to know whether it has changed.

Creating a timestamp is easy. In SQL Server, you create a timestamp field by using the timestamp data type. In other database products, timestamps are sometimes called *row versions*.

The Last Word

This chapter presented a thorough overview of data binding in ASP.NET. First you learned an interesting way to create dynamic text with simple data binding. Although this is a reasonable approach to get information into your page, it doesn't surpass what you can already do with pure code. You also learned how ASP.NET builds on this infrastructure with much more useful features, including repeated-value binding for quick-and-easy data display in a list control, and data source controls, which let you create code-free bound pages.

Using the techniques in this chapter, you can create a wide range of data-bound pages. However, if you want to create a page that incorporates record editing, sorting, and other more advanced tricks, the data-binding features you've learned about so far are just the first step. You'll also need to turn to specialized controls, such as the `DetailsView` and the `GridView`, which build upon these data-binding features. You'll learn how to master these controls in the next chapter. In Chapter 22, you'll learn how to extend your data-binding skills to work with data access components.



The Data Controls

When it comes to data binding, not all ASP.NET controls are created equal. In the previous chapter, you saw how data binding can help you automatically insert single values and lists into all kinds of common controls. In this chapter, you'll concentrate on three more advanced controls—GridView, DetailsView, and FormView—that allow you to bind entire tables of data.

The rich data controls are quite a bit different from the simple list controls. For one thing, they are designed exclusively for data binding. They also have the ability to display more than one field at a time, often in a table-based layout or according to what you've defined. They also support higher-level features such as selecting, editing, and sorting.

The rich data controls include the following:

- *GridView*: The GridView is an all-purpose grid control for showing large tables of information. The GridView is the heavyweight of ASP.NET data controls.
- *DetailsView*: The DetailsView is ideal for showing a single record at a time, in a table that has one row per field. The DetailsView also supports editing.
- *FormView*: Like the DetailsView, the FormView shows a single record at a time and supports editing. The difference is that the FormView is based on templates, which allow you to combine fields in a flexible layout that doesn't need to be table based.
- *ListView*: The ListView plays the same role as the GridView—it allows you to show multiple records. The difference is that the ListView is based on templates. As a result, using the ListView requires a bit more work and gives you slightly more layout flexibility. The ListView isn't described in this book, although you can learn more about it in the Visual Studio Help, or in the book *Pro ASP.NET 4.5 in C#* (Apress).

In this chapter, you'll explore the rich data controls in detail.

The GridView

The GridView is an extremely flexible grid control that displays a multicolumn table. Each record in your data source becomes a separate row in the grid. Each field in the record becomes a separate column in the grid.

The GridView is the most powerful of the rich data controls you'll learn about in this chapter because it comes equipped with the most ready-made functionality. This functionality includes features for automatic paging, sorting, selecting, and editing. The GridView is also the only data control you'll consider in this chapter that can show more than one record at a time.

Automatically Generating Columns

The GridView provides a DataSource property for the data object you want to display, much like the list controls you saw in Chapter 15. Once you've set the DataSource property, you call the DataBind() method to perform the data binding and display each record in the GridView. However, the GridView doesn't provide properties, such as DataTextField and DataValueField, that allow you to choose what column you want to display. That's because the GridView automatically generates a column for *every* field, as long as the AutoGenerateColumns property is true (which is the default).

Here's all you need to create a basic grid with one column for each field:

```
<asp:GridView ID="GridView1" runat="server" />
```

Once you've added this GridView tag to your page, you can fill it with data. Here's an example that performs a query using the ADO.NET objects and binds the retrieved DataSet:

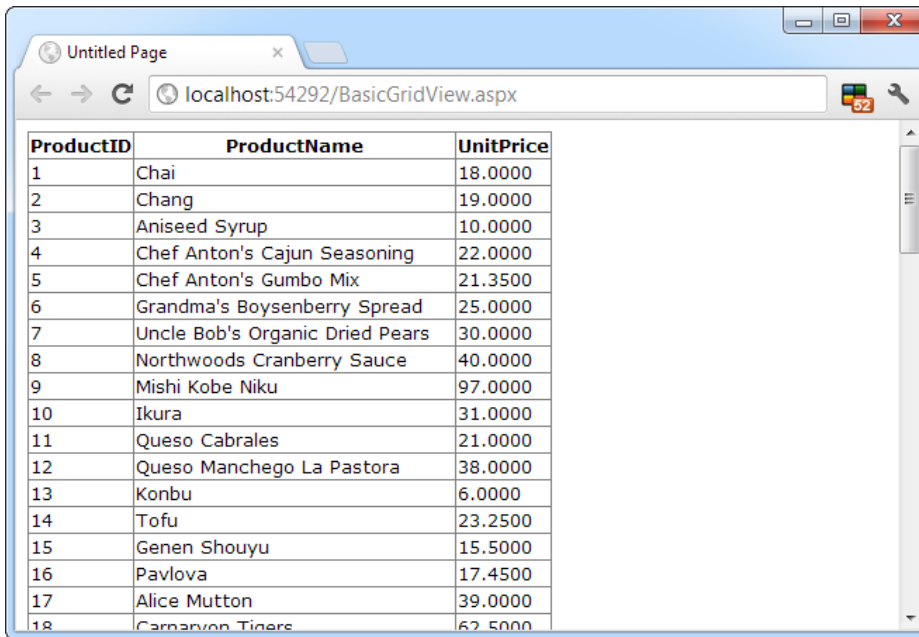
```
protected void Page_Load(object sender, EventArgs e)
{
    // Define the ADO.NET objects.
    string connectionString =
        WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
    string selectSQL = "SELECT ProductID, ProductName, UnitPrice FROM Products";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    // Fill the DataSet.
    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");

    // Perform the binding.
    GridView1.DataSource = ds;
    GridView1.DataBind();
}
```

Remember, in order for this code to work you must have a connection string named Northwind in the web.config file (just as you did for the examples in the previous two chapters).

Figure 16-1 shows the GridView this code creates.



ProductID	ProductName	UnitPrice
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000

Figure 16-1. The bare-bones GridView

Of course, you don't need to write this data access code by hand. As you learned in the previous chapter, you can use the `SqlDataSource` control to define your query. You can then link that query directly to your data control, and ASP.NET will take care of the entire data binding process.

Here's how you would define a `SqlDataSource` to perform the query shown in the previous example:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
  ConnectionString="<%= ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductID, ProductName, UnitPrice FROM Products" />
```

Next, set the `GridView.DataSourceID` property to link the data source to your grid:

```
<asp:GridView ID="GridView1" runat="server"
  DataSourceID="sourceProducts" />
```

These two tags duplicate the example in Figure 16-1 but with significantly less effort. Now you don't have to write any code to execute the query and bind the `DataSet`.

Using the `SqlDataSource` has positive and negative sides. Although it gives you less control, it streamlines your code quite a bit, and it allows you to remove all the database details from your code-behind class. In this chapter, you'll focus on the data source approach because it's much simpler when creating complex data-bound pages that support features such as editing. In Chapter 22, you'll learn how to adapt these examples to use the `ObjectDataSource` instead of the `SqlDataSource`. The `ObjectDataSource` is a great compromise: it allows you to write customized data access code in a database component without giving up the convenient design-time features of the data source controls.

Defining Columns

By default, the `GridView.AutoGenerateColumns` property is true, and the `GridView` creates a column for each field in the bound `DataTable`. This automatic column generation is good for creating quick test pages, but it doesn't give you the flexibility you'll usually want. For example, suppose you want to hide columns, change their order, or configure some aspect of their display, such as the formatting or heading text. In all these cases, you need to set `AutoGenerateColumns` to false and define the columns in the `<Columns>` section of the `GridView` control tag.

■ **Tip** It's possible to have `AutoGenerateColumns` set to true and define columns in the `<Columns>` section. In this case, the columns you explicitly define are added before the autogenerated columns. However, for the most flexibility, you'll usually want to explicitly define every column.

Each column can be any of several column types, as described in Table 16-1. The order of your column tags determines the left-to-right order of columns in the `GridView`.

Table 16-1. *Column Types*

Class	Description
<code>BoundField</code>	This column displays text from a field in the data source.
<code>ButtonField</code>	This column displays a button in this grid column.
<code>CheckBoxField</code>	This column displays a check box in this grid column. It's used automatically for true/false fields (in SQL Server, these are fields that use the bit data type).
<code>CommandField</code>	This column provides selection or editing buttons.
<code>HyperLinkField</code>	This column displays its contents (a field from the data source or static text) as a hyperlink.
<code>ImageField</code>	This column displays image data from a binary field (providing it can be successfully interpreted as a supported image format).
<code>TemplateField</code>	This column allows you to specify multiple fields, custom controls, and arbitrary HTML using a custom template. It gives you the highest degree of control but requires the most work.

The most basic column type is `BoundField`, which binds to one field in the data object. For example, here's the definition for a single data-bound column that displays the `ProductID` field:

```
<asp:BoundField DataField="ProductID" HeaderText="ID" />
```

This tag demonstrates how you can change the header text at the top of a column from `ProductID` to just `ID`.

Here's a complete `GridView` declaration with explicit columns:

```
<asp:GridView ID="GridView1" runat="server" DataSourceID="sourceProducts"
  AutoGenerateColumns="False">
  <Columns>
```



```

<asp:BoundField DataField="ProductID" HeaderText="ID" />
<asp:BoundField DataField="ProductName" HeaderText="Product Name" />
<asp:BoundField DataField="UnitPrice" HeaderText="Price" />
</Columns>
</asp:GridView>

```

Explicitly defining the columns has several advantages:

- You can easily fine-tune your column order, column headings, and other details by tweaking the properties of your column object.
- You can hide columns you don't want to show by removing the column tag. (Don't overuse this technique, because it's better to reduce the amount of data you're retrieving if you don't intend to display it.)
- You'll see your columns in the design environment (in Visual Studio). With automatically generated columns, the GridView simply shows a few generic placeholder columns.
- You can add extra columns to the mix for selecting, editing, and more.

This example shows how you can use this approach to change the header text. However, the `HeaderText` property isn't the only column property you can change in a column. In the next section, you'll learn about a few more.

Configuring Columns

When you explicitly declare a bound field, you have the opportunity to set other properties. Table 16-2 lists these properties.

Table 16-2. *BoundField Properties*

Property	Description
<code>DataField</code>	Identifies the field (by name) that you want to display in this column.
<code>DataFormatString</code>	Formats the field. This is useful for getting the right representation of numbers and dates.
<code>ApplyFormatInEditMode</code>	If true, the <code>DataFormat</code> string is used to format the value even when the value appears in a text box in edit mode. The default is false, which means the underlying value will be used (such as 1143.02 instead of \$1,143.02).
<code>FooterText</code> , <code>HeaderText</code> , and <code>HeaderImageUrl</code>	Sets the text in the header and footer region of the grid if this grid has a header (<code>GridView.ShowHeader</code> is true) and footer (<code>GridView.ShowFooter</code> is true). The header is most commonly used for a descriptive label such as the field name; the footer can contain a dynamically calculated value such as a summary. To show an image in the header <i>instead</i> of text, set the <code>HeaderImageUrl</code> property.
<code>ReadOnly</code>	If true, it prevents the value for this column from being changed in edit mode. No edit control will be provided. Primary key fields are often read-only.
<code>InsertVisible</code>	If true, it prevents the value for this column from being set in insert mode. If you want a column value to be set programmatically or based on a default value defined in the database, you can use this feature.

(continued)

Table 16-2. (continued)

Property	Description
Visible	If false, the column won't be visible in the page (and no HTML will be rendered for it). This gives you a convenient way to programmatically hide or show specific columns, changing the overall view of the data.
SortExpression	Sorts your results based on one or more columns. You'll learn about sorting later in the "Sorting and Paging the GridView" section of this chapter.
HtmlEncode	If true (the default), all text will be HTML encoded to prevent special characters from mangling the page. You could disable HTML encoding if you want to embed a working HTML tag (such as a hyperlink), but this approach isn't safe. It's always better to use HTML encoding on all values and provide other functionality by reacting to GridView selection events.
NullDisplayText	Displays the text that will be shown for a null value. The default is an empty string, although you could change this to a hard-coded value, such as "(not specified)."
ConvertEmptyStringToNull	If true, converts all empty strings to null values (and uses the NullDisplayText to display them).
ControlStyle, HeaderStyle, FooterStyle, and ItemStyle	Configures the appearance for just this column, overriding the styles for the row. You'll learn more about styles throughout this chapter.

Generating Columns with Visual Studio

As you've already learned, you can create a GridView that shows all your fields by setting the `AutoGenerateColumns` property to true. Unfortunately, when you use this approach you lose the ability to control any of the details regarding your columns, including their order, formatting, sorting, and so on. To configure these details, you need to set `AutoGenerateColumns` to false and define your columns explicitly. This requires more work, and it's a bit tedious.

However, there is a nifty trick that solves this problem. You can use explicit columns but get Visual Studio to create the column tags for you automatically. Here's how it works: select the GridView control, and click Refresh Schema in the smart tag. At this point, Visual Studio will retrieve the basic schema information from your data source (for example, the names and data type of each column) and then add one `<BoundField>` element for each field.

Tip If you modify the data source so it returns a different set of columns, you can regenerate the GridView columns. Just select the GridView, and click the Refresh Schema link in the smart tag. This step will wipe out any custom columns you've added (such as editing controls).

Once you've created your columns, you can also use some helpful design-time support to configure the properties of each column (rather than editing the column tag by hand). To do this, select the GridView and click the ellipsis (...) next to the Columns property in the Properties window. You'll see a Fields dialog box that lets you add, remove, and refine your columns (see Figure 16-2).

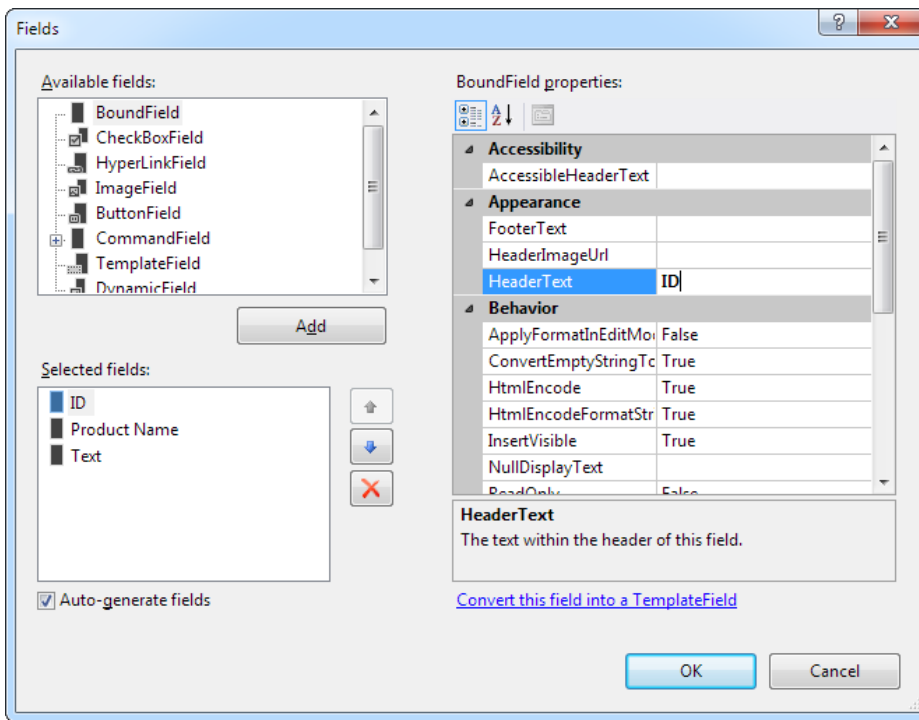


Figure 16-2. Configuring columns in Visual Studio

Now that you understand the underpinnings of the GridView, you've begun to explore some of its higher-level features. In the following sections, you'll tackle these topics:

Formatting: How to format rows and data values

Selecting: How to let users select a row in the GridView and respond accordingly

Editing: How to let users commit record updates, inserts, and deletes

Sorting: How to dynamically reorder the GridView in response to clicks on a column header

Paging: How to divide a large result set into multiple pages of data

Templates: How to take complete control of designing, formatting, and editing by defining templates

Formatting the GridView

Formatting consists of several related tasks. First, you want to ensure that dates, currencies, and other number values are presented in the appropriate way. You handle this job with the `DataFormatString` property. Next, you'll want to apply the perfect mix of colors, fonts, borders, and alignment options to each aspect of the grid, from headers to data items. The `GridView` supports these features through styles. Finally, you can intercept events, examine row data, and apply formatting to specific values programmatically. In the following sections, you'll consider each of these techniques.

The `GridView` also exposes several self-explanatory formatting properties that aren't covered here. These include `GridLines` (for adding or hiding table borders), `CellPadding` and `CellSpacing` (for controlling the overall spacing between cells), and `Caption` and `CaptionAlign` (for adding a title to the top of the grid).

■ **Tip** Want to create a `GridView` that scrolls—inside a web page? It's easy. Just place the `GridView` inside a `Panel` control, set the appropriate size for the panel, and set the `Panel.Scrollbars` property to `Auto`, `Vertical`, or `Both`.

Formatting Fields

Each `BoundField` column provides a `DataFormatString` property you can use to configure the appearance of numbers and dates using a *format string*.

Format strings generally consist of a placeholder and a format indicator, which are wrapped inside curly brackets. A typical format string looks something like this:

```
{0:C}
```

In this case, the 0 represents the value that will be formatted, and the letter indicates a predetermined format style. Here, C means currency format, which formats a number as an amount of money (so 3400.34 becomes \$3,400.34, assuming the web server is set to use U.S. regional settings). Here's a column that uses this format string:

```
<asp:BoundField DataField="UnitPrice" HeaderText="Price"
  DataFormatString="{0:C}" />
```

Table 16-3 shows some of the other formatting options for numeric values.

Table 16-3. *Numeric Format Strings*

Type	Format String	Example
Currency	{0:C}	\$1,234.50. Brackets indicate negative values: (\$1,234.50). The currency sign is locale-specific.
Scientific (Exponential)	{0:E}	1.234500E+003
Percentage	{0:P}	45.6 %
Fixed Decimal	{0:F?}	Depends on the number of decimal places you set. {0:F3} would be 123.400. {0:F0} would be 123.

You can find other examples in the MSDN Help. For date or time values, you'll find an extensive list. For example, if you want to write the `BirthDate` value in the format `month/day/year` (as in `12/30/12`), you use the following column:

```
<asp:BoundField DataField="BirthDate" HeaderText="Birth Date"
  DataFormatString="{0:MM/dd/yy}" />
```

Table 16-4 shows some more examples.

Table 16-4. *Time and Date Format Strings*

Type	Format String	Syntax	Example
Short Date	{0:d}	M/d/yyyy	10/30/2012
Long Date	{0:D}	dddd, MMMM dd, yyyy	Monday, January 30, 2012
Long Date and Short Time	{0:f}	dddd, MMMM dd, yyyy HH:mm aa	Monday, January 30, 2012 10:00 AM
Long Date and Long Time	{0:F}	dddd, MMMM dd, yyyy HH:mm:ss aa	Monday, January 30, 2012 10:00:23 AM
ISO Sortable Standard	{0:s}	yyyy-MM-ddTHH:mm:ss	2012-01-30 T10:00:23
Month and Day	{0:M}	MMMM dd	January 30
General	{0:G}	M/d/yyyy HH:mm:ss aa (depends on locale-specific settings)	10/30/2012 10:00:23 AM

The format characters are not specific to the `GridView`. You can use them with other controls, with data-bound expressions in templates (as you'll see later in the "Using `GridView` Templates" section), and as parameters for many methods. For example, the `Decimal` and `DateTime` types expose their own `ToString()` methods that accept a format string, allowing you to format values manually.

Using Styles

The `GridView` exposes a rich formatting model that's based on *styles*. Altogether, you can set eight `GridView` styles, as described in Table 16-5.

Table 16-5. *GridView Styles*

Style	Description
HeaderStyle	Configures the appearance of the header row that contains column titles, if you've chosen to show it (if ShowHeader is true).
RowStyle	Configures the appearance of every data row.
AlternatingRowStyle	If set, applies additional formatting to every other row. This formatting acts in addition to the RowStyle formatting. For example, if you set a font using RowStyle, it is also applied to alternating rows, unless you explicitly set a different font through AlternatingRowStyle.
SelectedRowStyle	Configures the appearance of the row that's currently selected. This formatting acts in addition to the RowStyle formatting.
EditRowStyle	Configures the appearance of the row that's in edit mode. This formatting acts in addition to the RowStyle formatting.
EmptyDataRowStyle	Configures the style that's used for the single empty row in the special case where the bound data object contains no rows.
FooterStyle	Configures the appearance of the footer row at the bottom of the GridView, if you've chosen to show it (if ShowFooter is true).
PagerStyle	Configures the appearance of the row with the page links, if you've enabled paging (set AllowPaging to true).

Styles are not simple single-value properties. Instead, each style exposes a Style object that includes properties for choosing colors (ForeColor and BackColor), adding borders (BorderColor, BorderStyle, and BorderWidth), sizing the row (Height and Width), aligning the row (HorizontalAlign and VerticalAlign), and configuring the appearance of text (Font and Wrap). These style properties allow you to refine almost every aspect of an item's appearance.

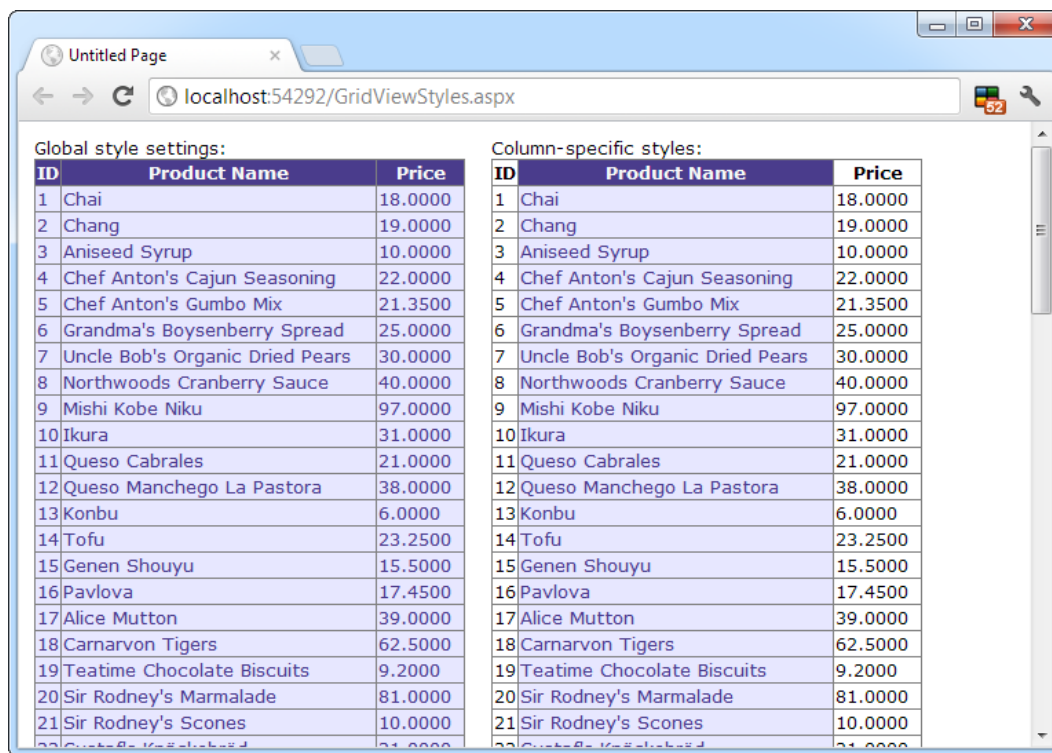
Here's an example that changes the style of rows and headers in a GridView:

```
<asp:GridView ID="GridView1" runat="server" DataSourceID="sourceProducts"
  AutoGenerateColumns="False">
  <RowStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
  <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
  <Columns>
    <asp:BoundField DataField="ProductID" HeaderText="ID" />
    <asp:BoundField DataField="ProductName" HeaderText="Product Name" />
    <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
  </Columns>
</asp:GridView>
```

In this example, every column is affected by the formatting changes. However, you can also define column-specific styles. To create a column-specific style, you simply need to rearrange the control tag so that the formatting tag becomes a nested tag *inside* the appropriate column tag. Here's an example that formats just the ProductName column:

```
<asp:GridView ID="GridView2" runat="server" DataSourceID="sourceProducts"
  AutoGenerateColumns="False" >
  <Columns>
    <asp:BoundField DataField="ProductID" HeaderText="ID" />
    <asp:BoundField DataField="ProductName" HeaderText="Product Name">
      <ItemStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
      <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
    </asp:BoundField>
    <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
  </Columns>
</asp:GridView>
```

Figure 16-3 compares these two examples. You can use a combination of ordinary style settings and column-specific style settings (which override ordinary style settings if they conflict).



ID	Product Name	Price
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000
19	Teatime Chocolate Biscuits	9.2000
20	Sir Rodney's Marmalade	81.0000
21	Sir Rodney's Scones	10.0000
22	Gustaf's Coffee	21.0000

ID	Product Name	Price
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000
19	Teatime Chocolate Biscuits	9.2000
20	Sir Rodney's Marmalade	81.0000
21	Sir Rodney's Scones	10.0000
22	Gustaf's Coffee	21.0000

Figure 16-3. Formatting the GridView

One reason you might use column-specific formatting is to define specific column widths. If you don't define a specific column width, ASP.NET makes each column just wide enough to fit the data it contains (or, if wrapping is enabled, to fit the text without splitting a word over a line break). If values range in size, the width is determined by the largest value or the width of the column header, whichever is larger. However, if the grid is wide enough, you might want to expand a column so it doesn't appear to be crowded against the adjacent columns. In this case, you need to explicitly define a larger width.

Configuring Styles with Visual Studio

There's no reason to code style properties by hand in the GridView control tag because the GridView provides rich design-time support. To set style properties, you can use the Properties window to modify the style properties. For example, to configure the font of the header, expand the HeaderStyle property to show the nested Font property, and set that. The only limitation on this approach is that it doesn't allow you to set the style for individual columns; if you need that trick, you must first call up the Fields dialog box (shown in Figure 16-2) by editing the Columns property. Then, select the appropriate column and set the style properties accordingly.

You can even set a combination of styles using a preset theme by clicking the Auto Format link in the GridView smart tag. Figure 16-4 shows the Auto Format dialog box with some of the preset styles you can choose. Select Remove Formatting to clear all the style settings.

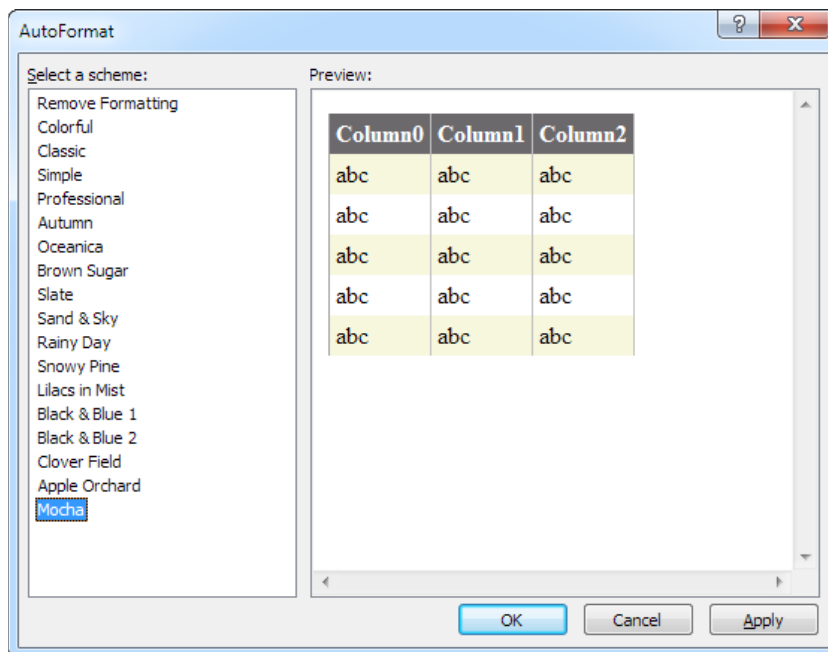


Figure 16-4. Automatically formatting a GridView

Once you've chosen and inserted the styles into your GridView tag, you can tweak them by hand or by using the Properties window.

Formatting-Specific Values

The formatting you've learned so far isn't that fine-grained. At its most specific, this formatting applies to a single column of values. But what if you want to change the formatting for a specific row or even for just a single cell?

The solution is to react to the `GridView.RowDataBound` event. This event is raised for each row, just after it's filled with data. At this point, you can access the current row as a `GridViewRow` object. The `GridViewRow.DataItem` property provides the data object for the given row, and the `GridViewRow.Cells` collection allows you to retrieve the row content. You can use the `GridViewRow` to change colors and alignment, add or remove child controls, and so on.

The following example handles the `RowDataBound` event and changes the background color to highlight high prices (those more expensive than \$50):

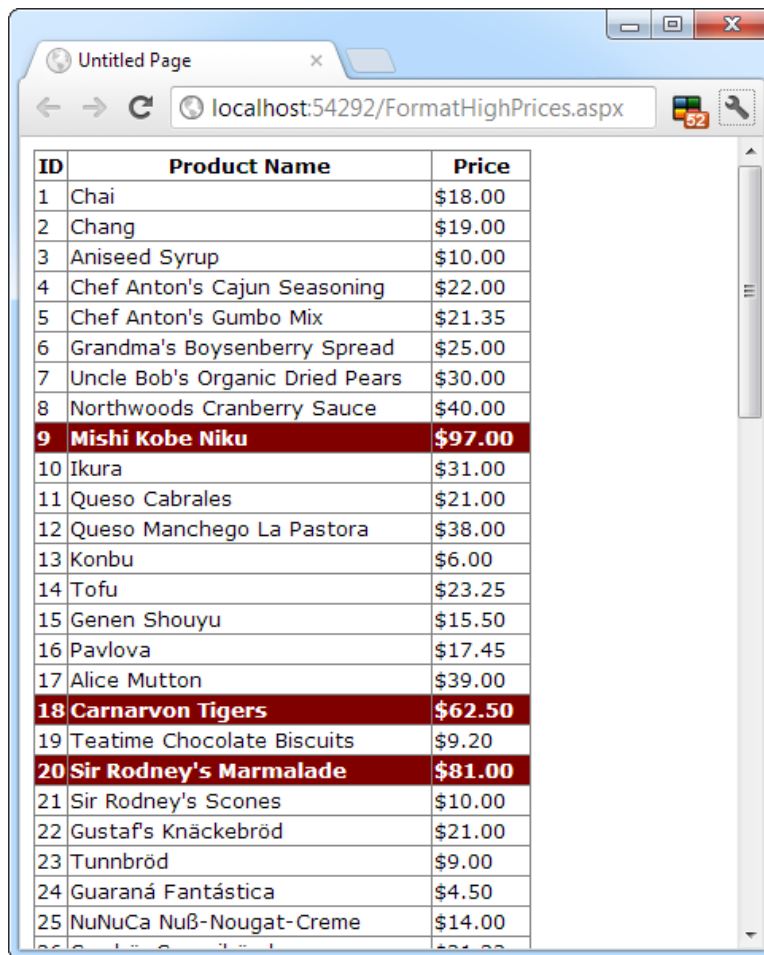
```
protected void GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Get the price for this row.
        decimal price = (decimal)DataBinder.Eval(e.Row.DataItem, "UnitPrice");

        if (price > 50)
        {
            e.Row.BackColor = System.Drawing.Color.Maroon;
            e.Row.ForeColor = System.Drawing.Color.White;
            e.Row.Font.Bold = true;
        }
    }
}
```

First, the code checks whether the item being created is a row or an alternate row. If neither, it means the item is another interface element, such as the pager, footer, or header, and the procedure does nothing. If the item is the right type, the code extracts the `UnitPrice` field from the data-bound item.

To get a value from the bound data object (provided through the `GridViewRowEventArgs.Row.DataItem` property), you need to cast the data object to the correct type. The trick is that the type depends on the way you're performing your data binding. In this example, you're binding to the `SqlDataSource` in `DataSet` mode, which means each data item will be a `DataRowView` object. (If you were to bind in `DataReader` mode, a `DbDataRecord` represents each item instead.) To avoid coding these details, which can make it more difficult to change your data access code, you can rely on the `DataBinder.Eval()` helper method, which understands all these types of data objects. That's the technique used in this example.

Figure 16-5 shows the resulting page.



ID	Product Name	Price
1	Chai	\$18.00
2	Chang	\$19.00
3	Aniseed Syrup	\$10.00
4	Chef Anton's Cajun Seasoning	\$22.00
5	Chef Anton's Gumbo Mix	\$21.35
6	Grandma's Boysenberry Spread	\$25.00
7	Uncle Bob's Organic Dried Pears	\$30.00
8	Northwoods Cranberry Sauce	\$40.00
9	Mishi Kobe Niku	\$97.00
10	Ikura	\$31.00
11	Queso Cabrales	\$21.00
12	Queso Manchego La Pastora	\$38.00
13	Konbu	\$6.00
14	Tofu	\$23.25
15	Genen Shouyu	\$15.50
16	Pavlova	\$17.45
17	Alice Mutton	\$39.00
18	Carnarvon Tigers	\$62.50
19	Teatime Chocolate Biscuits	\$9.20
20	Sir Rodney's Marmalade	\$81.00
21	Sir Rodney's Scones	\$10.00
22	Gustaf's Knäckebröd	\$21.00
23	Tunnbröd	\$9.00
24	Guaraná Fantástica	\$4.50
25	NuNuCa Nuß-Nougat-Creme	\$14.00

Figure 16-5. Formatting individual rows based on values

Selecting a GridView Row

Selecting an item refers to the ability to click a row and have it change color (or become highlighted) to indicate that the user is currently working with this record. At the same time, you might want to display additional information about the record in another control. With the GridView, selection happens almost automatically once you set up a few basics.

Before you can use item selection, you must define a different style for selected items. The `SelectedRowStyle` determines how the selected row or cell will appear. If you don't set this style, it will default to the same value as `RowStyle`, which means the user won't be able to tell which row is currently selected. Usually, selected rows will have a different `BackColor` property.

To find out what item is currently selected (or to change the selection), you can use the GridView's `SelectedIndex` property. It will be -1 if no item is currently selected. Also, you can react to the `SelectedIndexChanged` event to handle any additional related tasks. For example, you might want to update another control with additional information about the selected record.

Adding a Select Button

The GridView provides built-in support for selection. You simply need to add a CommandField column with the ShowSelectButton property set to true. ASP.NET can render the CommandField as a hyperlink, a button, or a fixed image. You choose the type using the ButtonType property. You can then specify the text through the SelectText property or specify the link to the image through the SelectImageUrl property.


Here's an example that displays a select button:

```
<asp:CommandField ShowSelectButton="True" ButtonType="Button"
  SelectText="Select" />
```

And here's an example that shows a small clickable icon:

```
<asp:CommandField ShowSelectButton="True" ButtonType="Image"
  SelectImageUrl="select.gif" />
```

Figure 16-6 shows a page with a text select button (and product 14 selected).



	ProductID	ProductName	UnitPrice
Select	1	Chai	18.0000
Select	2	Chang	19.0000
Select	3	Aniseed Syrup	10.0000
Select	4	Chef Anton's Cajun Seasoning	22.0000
Select	5	Chef Anton's Gumbo Mix	21.3500
Select	6	Grandma's Boysenberry Spread	25.0000
Select	7	Uncle Bob's Organic Dried Pears	30.0000
Select	8	Northwoods Cranberry Sauce	40.0000
Select	9	Mishi Kobe Niku	97.0000
Select	10	Ikura	31.0000
Select	11	Queso Cabrales	21.0000
Select	12	Queso Manchego La Pastora	38.0000
Select	13	Konbu	6.0000
Select	14	Tofu	23.2500
Select	15	Genen Shouyu	15.5000
Select	16	Pavlova	17.4500
Select	17	Alice Mutton	39.0000
Select	18	Carnarvon Tigers	62.5000
Select	19	Teatime Chocolate Biscuits	9.2000

Figure 16-6. GridView selection

When you click a select button, the page is posted back and a series of steps unfolds. First, the GridView.SelectedIndexChanging event fires, which you can intercept to cancel the operation. Next, the GridView.SelectedIndex property is adjusted to point to the selected row. Finally, the GridView.SelectedIndexChanged event fires, which you can handle if you want to manually update other controls to reflect the new selection. When the page is rendered, the selected row is given the selected row style.

■ **Tip** Rather than add the select button yourself, you can choose Enable Selection from the GridView's smart tag, which adds a basic select button for you.

Using a Data Field as a Select Button

You don't need to create a new column to support row selection. Instead, you can turn an existing column into a link. This technique is commonly implemented to allow users to select rows in a table by the unique ID value.

To use this technique, remove the CommandField column and add a ButtonField column instead. Then, set the DataTextField to the name of the field you want to use.

```
<asp:ButtonField ButtonType="Button" DataTextField="ProductID" />
```

This field will be underlined and turned into a button that, when clicked, will post back the page and trigger the GridView.RowCommand event. You could handle this event, determine which row has been clicked, and programmatically set the SelectedIndex property of the GridView. However, you can use an easier method. Instead, just configure the link to raise the SelectedIndexChanged event by specifying a CommandName with the text *Select*, as shown here:

```
<asp:ButtonField CommandName="Select" ButtonType="Button"
  DataTextField="ProductID" />
```

Now, clicking the data field automatically selects the record.

Using Selection to Create Master-Details Pages

As demonstrated in the previous chapter, you can draw a value out of a control and use it to perform a query in your data source. For example, you can take the currently selected item in a list, and feed that value to a SqlDataSource that gets more information for the corresponding record.

This trick is a great way to build *master-details pages*—pages that let you navigate relationships in a database. A typical master-details page has two GridView controls. The first GridView shows the master (or parent) table. When a user selects an item in the first GridView, the second GridView is filled with related records from the details (or parent) table. For example, a typical implementation of this technique might have a Customers table in the first GridView. Select a customer, and the second GridView is filled with the list of orders made by that customer.

To create a master-details page, you need to extract the SelectedIndex property from the first GridView and use that to craft a query for the second GridView. However, this approach has one problem. SelectedIndex returns a zero-based index number that represents where the row occurs in the grid. This isn't the information you need to insert into the query that gets the related records. Instead, you need a unique key field from the corresponding row. For example, if you have a table of products, you need to be able to get the ProductID for the selected row. In order to get this information, you need to tell the GridView to keep track of the key field values.

The way you do this is by setting the DataKeyNames property for the GridView. This property requires a comma-separated list of one or more key fields. Each name you supply must match one of the fields in the bound

data source. Usually, you'll have only one key field. Here's an example that tells the GridView to keep track of the CategoryID values in a list of product categories:

```
<asp:GridView ID="gridCategories" runat="server"
DataKeyNames="CategoryID" ... >
```

Once you've established this link, the GridView is nice enough to keep track of the key fields for the selected record. It allows you to retrieve this information at any time through the SelectedDataKey property.

The following example puts it all together. It defines two GridView controls. The first shows a list of categories. The second shows the products that fall into the currently selected category (or, if no category has been selected, this GridView doesn't appear at all).

Here's the page markup for this example:

```
Categories:<br />
<asp:GridView ID="gridCategories" runat="server" DataSourceID="sourceCategories"
  DataKeyNames="CategoryID">
  <Columns>
    <asp:CommandField ShowSelectButton="True" />
  </Columns>
  <SelectedRowStyle BackColor="#FFCC66" Font-Bold="True"
    ForeColor="#663399" />
</asp:GridView>
<asp:SqlDataSource ID="sourceCategories" runat="server"
  ConnectionString="<%= $ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT * FROM Categories"></asp:SqlDataSource>
<br />

Products in this category:<br />
<asp:GridView ID="gridProducts" runat="server" DataSourceID="sourceProducts">
  <SelectedRowStyle BackColor="#FFCC66" Font-Bold="True" ForeColor="#663399" />
</asp:GridView>
<asp:SqlDataSource ID="sourceProducts" runat="server"
  ConnectionString="<%= $ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductID, ProductName, UnitPrice FROM Products WHERE
  CategoryID=@CategoryID">
  <SelectParameters>
    <asp:ControlParameter Name="CategoryID" ControlID="gridCategories"
      PropertyName="SelectedDataKey.Value" />
  </SelectParameters>
</asp:SqlDataSource>
```

As you can see, you need two data sources, one for each GridView. The second data source uses a ControlParameter that links it to the SelectedDataKey property of the first GridView. Best of all, you still don't need to write any code or handle the SelectedIndexChanged event on your own.

Figure 16-7 shows this example in action.

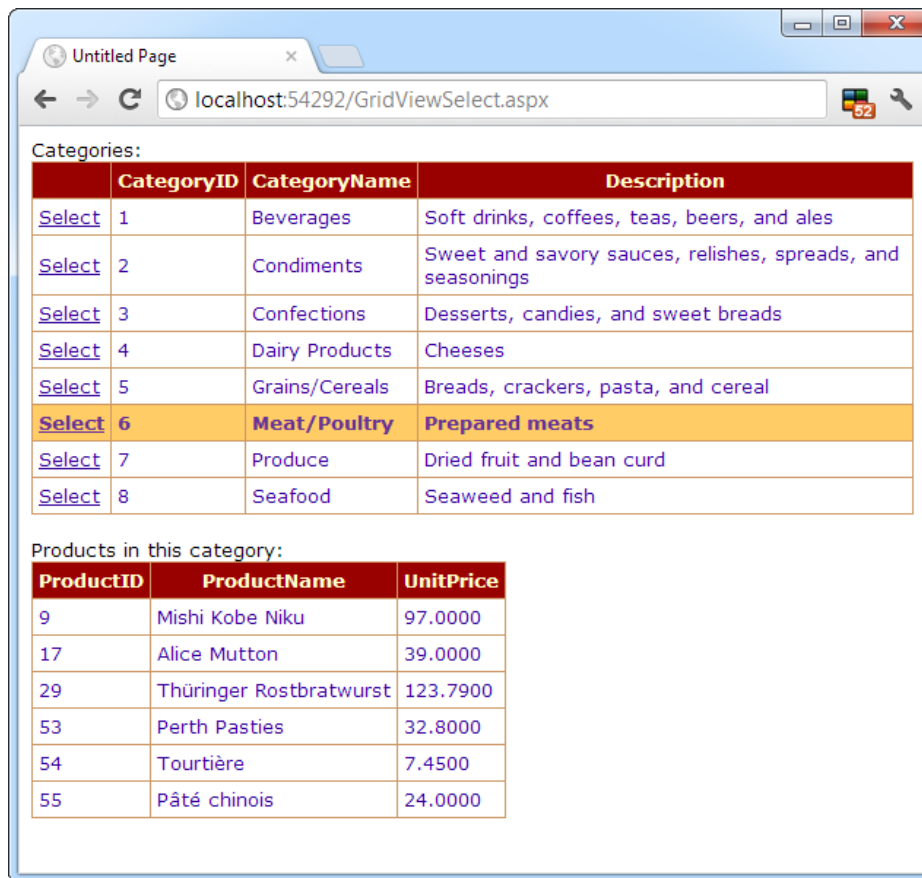


Figure 16-7. A master-details page

Editing with the GridView

The GridView provides support for editing that's almost as convenient as its support for selection. To switch a row into select mode, you simply set the `SelectedIndex` property to the corresponding row number. To switch a row into edit mode, you set the `EditIndex` property in the same way.

Of course, both of these tasks can take place automatically if you use specialized button types. For selection, you use a `CommandField` column with the `ShowSelectButton` property set to true. To add edit controls, you follow almost the same step—once again, you use the `CommandField` column, but now you set `ShowEditButton` to true.

Here's an example of a GridView that supports editing:

```
<asp:GridView ID="gridProducts" runat="server" DataSourceID="sourceProducts"
  AutoGenerateColumns="False" DataKeyNames="ProductID">
  <Columns>
    <asp:BoundField DataField="ProductID" HeaderText="ID" ReadOnly="True" />
    <asp:BoundField DataField="ProductName" HeaderText="Product Name"/>
    <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
    <asp:CommandField ShowEditButton="True" />
  </Columns>
</asp:GridView>
```

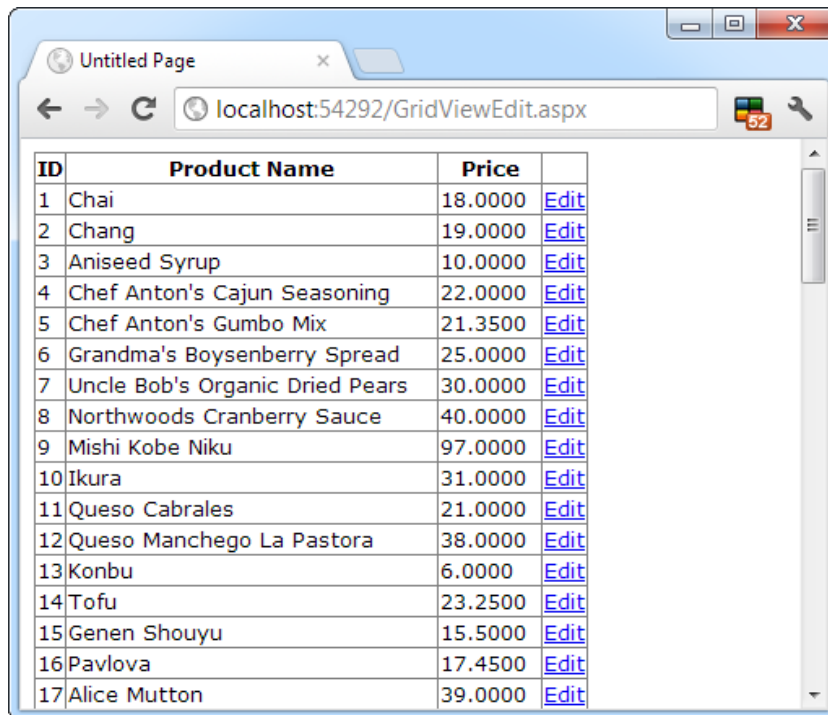
And here's a revised data source control that can commit your changes:

```
<asp:SqlDataSource id="sourceProducts" runat="server"
  ConnectionString="<%%$ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductID, ProductName, UnitPrice FROM Products"
  UpdateCommand="UPDATE Products SET ProductName=@ProductName,
UnitPrice=@UnitPrice WHERE ProductID=@ProductID" />
```

■ **Note** If you receive a `SqlException` that says “Must declare the scalar variable @ProductID,” the most likely problem is that you haven’t set the `GridView.DataKeyNames` property. Because the `ProductID` field can’t be modified, the `GridView` won’t pass the `ProductID` value to the `SqlDataSource` unless it’s designated a key field.

Remember, you don’t need to define the update parameters as long as you make sure they match the field names (with an *at* sign [`@`] at the beginning). Chapter 15 has more information about using update commands with the `SqlDataSource` control.

When you add a `CommandField` with the `ShowEditButton` property set to true, the `GridView` editing controls appear in an additional column. When you run the page and the `GridView` is bound and displayed, the edit column shows an `Edit` link next to every record (see Figure 16-8).



ID	Product Name	Price	
1	Chai	18.0000	Edit
2	Chang	19.0000	Edit
3	Aniseed Syrup	10.0000	Edit
4	Chef Anton's Cajun Seasoning	22.0000	Edit
5	Chef Anton's Gumbo Mix	21.3500	Edit
6	Grandma's Boysenberry Spread	25.0000	Edit
7	Uncle Bob's Organic Dried Pears	30.0000	Edit
8	Northwoods Cranberry Sauce	40.0000	Edit
9	Mishi Kobe Niku	97.0000	Edit
10	Ikura	31.0000	Edit
11	Queso Cabrales	21.0000	Edit
12	Queso Manchego La Pastora	38.0000	Edit
13	Konbu	6.0000	Edit
14	Tofu	23.2500	Edit
15	Genen Shouyu	15.5000	Edit
16	Pavlova	17.4500	Edit
17	Alice Mutton	39.0000	Edit

Figure 16-8. The editing controls

When clicked, this link switches the corresponding row into edit mode. All fields are changed to text boxes, with the exception of read-only fields (which are not editable) and true/false bit fields (which are shown as check boxes). The Edit link is replaced with an Update link and a Cancel link (see Figure 16-9).

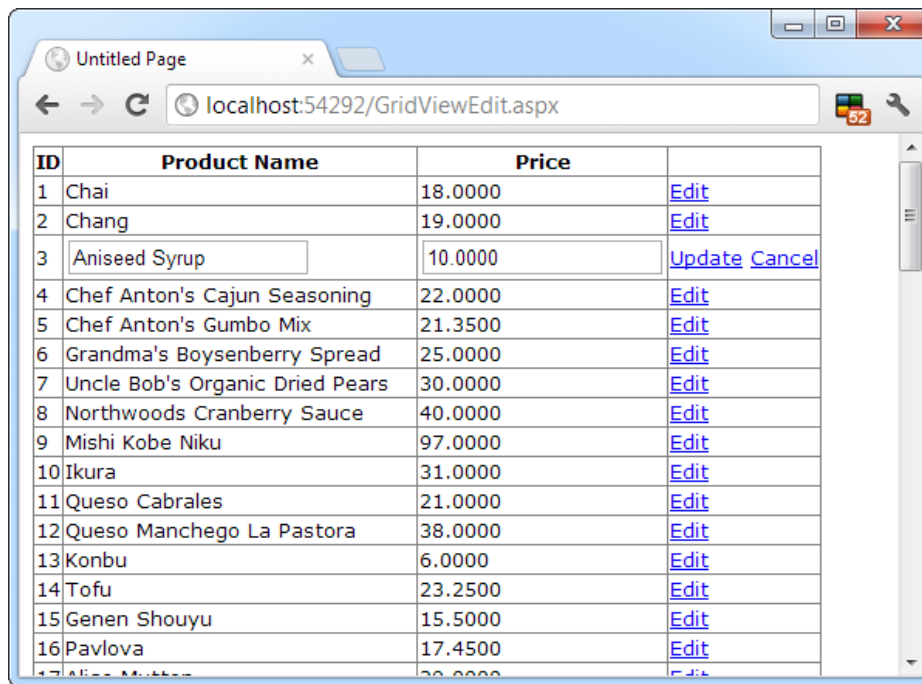


Figure 16-9. Editing a record

The Cancel link returns the row to its initial state. The Update link passes the values to the `SqlDataSource.UpdateParameters` collection (using the field names) and then triggers the `SqlDataSource.Update()` method to apply the change to the database. Once again, you don't have to write any code, provided you've filled in the `UpdateCommand` for the linked data source control.

You can use a similar approach to add support for record deleting. To enable deleting, you need to add a column to the GridView that has the `ShowDeleteButton` property set to true. As long as your linked `SqlDataSource` has the `DeleteCommand` property filled in, these operations will work automatically. If you want to write your own code that plugs into this process (for example, updating a label to inform the user the update has been made), consider reacting to the GridView event that fires after an update operation is committed, such as `RowDeleted` and `RowUpdated`. You can also prevent changes you don't like by reacting to the `RowDeleting` and `RowUpdating` events and setting the cancel flag in the event arguments.

The GridView does not support inserting records. If you want that ability, you can use one of the single-record display controls described later in this chapter, such as the `DetailsView` or `FormView`. For example, a typical ASP.NET page for data entry might show a list of records in a GridView and provide a `DetailsView` that allows the user to add new records.

■ **Note** The basic built-in updating features of the GridView don't give you a lot of flexibility. You can't change the types of controls that are used for editing, format these controls, or add validation. However, you can add all these features by building your own editing templates, a topic presented later in the "Using GridView Templates" section.

Sorting and Paging the GridView

The GridView is a great all-in-one solution for displaying all kinds of data, but it becomes a little unwieldy as the number of fields and rows in your data source grows. Dense grids contribute to large pages that are slow to transmit over the network and difficult for the user to navigate. The GridView has two features that address these issues and make data more manageable: sorting and paging.

Both sorting and paging can be performed by the database server, provided you craft the right SQL using the Order By and Where clauses. In fact, sometimes this is the best approach for performance. However, the sorting and paging provided by the GridView and SqlDataSource are easy to implement and thoroughly flexible. These techniques are particularly useful if you need to show the same data in several ways and you want to let the user decide how the data should be ordered.

Sorting

The GridView sorting features allow the user to reorder the results in the GridView by clicking a column header. It's convenient—and easy to implement.

Although you may not realize it, when you bind to a DataTable, you actually use another object called the DataView. The DataView sits between the ASP.NET web page binding and your DataTable. Usually it does little aside from providing the information from the associated DataTable. However, you can customize the DataView so it applies its own sort order. That way, you can also customize the data that appear in the web page without needing to actually modify your data.

You can create a new DataView object by hand and bind the DataView directly to a data control such as the GridView. However, the GridView and SqlDataSource controls make it even easier. They provide several properties you can set to control sorting. Once you've configured these properties, the sorting is automatic, and you still won't need to write any code in your page class.

To enable sorting, you must set the GridView.AllowSorting property to true. Next, you need to define a SortExpression for each column that can be sorted. In theory, a sort expression takes the form used in the ORDER BY clause of a SQL query and can use any syntax that's understood by the data source control. In practice, you'll almost always use a single field name to sort the grid using the data in that column. For example, here's how you could define the ProductName column so it sorts by alphabetically ordering rows:

```
<asp:BoundField DataField="ProductName" HeaderText="Product Name"
    SortExpression="ProductName" />
```

Note that if you don't want a column to be sort-enabled, you simply don't set its SortExpression property. Figure 16-10 shows an example with a grid that has sort expressions for all three columns and is currently sorted by product name.

ProductID	ProductName	UnitPrice
17	Alice Mutton	39.0000
3	Aniseed Syrup	10.0000
40	Boston Crab Meat	18.4000
60	Camembert Pierrot	34.0000
18	Carnarvon Tigers	62.5000
1	Chai	18.0000
2	Chang	19.0000
39	Chartreuse verte	18.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500

1 2 3 4 5 6 7 8

Figure 16-10. *Sorting the GridView*

Once you've associated a sort expression with the column and set the `AllowSorting` property to true, the GridView will render the headers with clickable links, as shown in Figure 16-10. However, it's up to the data source control to implement the actual sorting logic. How the sorting is implemented depends on the data source you're using.

Not all data sources support sorting, but the `SqlDataSource` does, provided the `DataSourceMode` property is set to `DataSet` (the default), not `DataReader`. In `DataReader` mode, the records are retrieved one at a time, and each record is stuffed into the bound control (such as a GridView) before the `SqlDataSource` moves to the next one. In `DataSet` mode, the entire results are placed in a `DataSet` and then the records are copied from the `DataSet` into the bound control. If the data need to be sorted, the sorting happens between these two steps—after the records are retrieved but before they're bound in the web page.

■ **Note** The sort is according to the data type of the column. Numeric and date columns are ordered from smallest to largest. String columns are sorted alphanumerically without regard to case. Columns that contain binary data cannot be sorted. However, if you click a column header twice, the second click *reverses* the sort order, putting the records in descending order. (Click a third time to switch it back to ascending order.)

Sorting and Selecting

If you use sorting and selection at the same time, you'll discover another issue. To see this problem in action, select a row and then sort the data by any column. You'll see that the selection will remain, but it will shift to a

new item that has the same index as the previous item. In other words, if you select the second row and perform a sort, the second row will still be selected in the new page, even though this isn't the record you selected.

To fix this problem, you must simply set the `GridView.EnablePersistedSelection` property true. Now, ASP.NET will ensure that the selected item is identified by its data key. As a result, the selected item will remain selected, even if it moves to a new position in the `GridView` after a sort.

Paging

Often, a database search will return too many rows to be realistically displayed in a single page. If the client is using a slow connection, an extremely large `GridView` can take a frustrating amount of time to arrive. Once the data are retrieved, the user may find out they don't contain the right content anyway or that the search was too broad and they can't easily wade through all the results to find the important information.

The `GridView` handles this scenario with an automatic paging feature. When you use automatic paging, the full results are retrieved from the data source and placed into a `DataSet`. Once the `DataSet` is bound to the `GridView`, however, the data are subdivided into smaller groupings (for example, with 20 rows each), and only a single batch is sent to the user. The other groups are abandoned when the page finishes processing. When the user moves to the next page, the same process is repeated—in other words, the full query is performed once again. The `GridView` extracts just one group of rows, and the page is rendered.

To allow the user to skip from one page to another, the `GridView` displays a group of pager controls at the bottom of the grid. These pager controls could be previous/next links (often displayed as < and >) or number links (1, 2, 3, 4, 5, . . .) that lead to specific pages. If you've ever used a search engine, you've seen paging at work.

By setting a few properties, you can make the `GridView` control manage the paging for you. Table 16-6 describes the key properties.

Table 16-6. *Paging Members of the GridView*

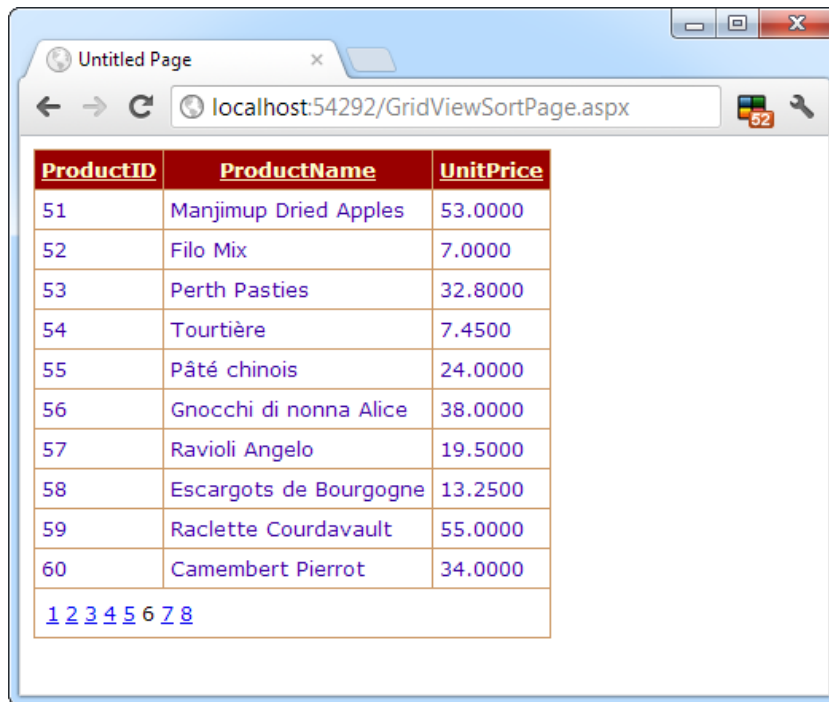
Property	Description
<code>AllowPaging</code>	Enables or disables the paging of the bound records. It is false by default.
<code>PageSize</code>	Gets or sets the number of items to display on a single page of the grid. The default value is 10.
<code>PageIndex</code>	Gets or sets the zero-based index of the currently displayed page, if paging is enabled.
<code>PagerSettings</code>	Provides a <code>PagerSettings</code> object that wraps a variety of formatting options for the pager controls. These options determine where the paging controls are shown and what text or images they contain. You can set these properties to fine-tune the appearance of the pager controls, or you can use the defaults.
<code>PagerStyle</code>	Provides a style object you can use to configure fonts, colors, and text alignment for the paging controls.
<code>PageIndexChanging</code> and <code>PageIndexChanged</code> events	Occur when one of the page selection elements is clicked, just before the <code>PageIndex</code> is changed (<code>PageIndexChanging</code>) and just after (<code>PageIndexChanged</code>).

To use automatic paging, you need to set `AllowPaging` to true (which shows the page controls), and you need to set `PageSize` to determine how many rows are allowed on each page.

Here's an example of a GridView control declaration that sets these properties:

```
<asp:GridView ID="GridView1" runat="server" DataSourceID="sourceProducts"
  PageSize="10" AllowPaging="True" ...>
  ...
</asp:GridView>
```

This is enough to start using paging. Figure 16-11 shows an example with ten records per page (for a total of eight pages).



The screenshot shows a web browser window titled 'Untitled Page' with the address bar displaying 'localhost:54292/GridViewSortPage.aspx'. The browser content displays a table with three columns: ProductID, ProductName, and UnitPrice. The table contains 10 rows of data. Below the table is a pagination control showing links for pages 1 through 8, with page 5 currently selected.

ProductID	ProductName	UnitPrice
51	Manjimup Dried Apples	53.0000
52	Filo Mix	7.0000
53	Perth Pasties	32.8000
54	Tourtière	7.4500
55	Pâté chinois	24.0000
56	Gnocchi di nonna Alice	38.0000
57	Ravioli Angelo	19.5000
58	Escargots de Bourgogne	13.2500
59	Raclette Courdavault	55.0000
60	Camembert Pierrot	34.0000

1 2 3 4 5 6 7 8

Figure 16-11. *Paging the GridView*

Paging and Selection

By default, paging and selection don't play nicely together. If you enable both for the GridView, you'll notice that the same row position remains selected as you move from one page to another. For example, if you select the first row on the first page and then move to the second page, the first row on the second page will become selected. To fix this quirk, set the GridView.EnablePersistedSelection property to true. Now, as you move from one page to another, the selection will automatically be removed from the GridView (and the SelectedIndex property will be set to -1). But if you move back to the page that held the originally selected row, that row will be re-selected. This behavior is intuitive, and it neatly ensures that your code won't be confused by a selected row that isn't currently visible.

PAGING AND PERFORMANCE

When you use paging, every time a new page is requested, the full DataSet is queried from the database. This means paging does not reduce the amount of time required to query the database. In fact, because the information is split into multiple pages and you need to repeat the query every time the user moves to a new page, the database load actually *increases*. However, because any given page contains only a subset of the total data, the page size is smaller and will be transmitted faster, reducing the client's wait. The end result is a more responsive and manageable page.

You can use paging in certain ways without increasing the amount of work the database needs to perform. One option is to cache the entire DataSet in server memory. That way, every time the user moves to a different page, you simply need to retrieve the data from memory and rebind it, avoiding the database altogether. You'll learn how to use this technique in Chapter 23.

Using GridView Templates

So far, the examples have used the GridView control to show data using separate bound columns for each field. If you want to place multiple values in the same cell, or you want the unlimited ability to customize the content in a cell by adding HTML tags and server controls, you need to use a TemplateField.

The TemplateField allows you to define a completely customized *template* for a column. Inside the template you can add control tags, arbitrary HTML elements, and data binding expressions. You have complete freedom to arrange everything the way you want.

For example, imagine you want to create a column that combines the in-stock, on-order, and reorder level information for a product. To accomplish this trick, you can construct an ItemTemplate like this:

```
<asp:TemplateField HeaderText="Status">
  <ItemTemplate>
    <b>In Stock:</b>
    <%= Eval("UnitsInStock") %> <br />
    <b>On Order:</b>
    <%= Eval("UnitsOnOrder") %> <br />
    <b>Reorder:</b>
    <%= Eval("ReorderLevel") %>
  </ItemTemplate>
</asp:TemplateField>
```

■ **Note** Your template only has access to the fields that are in the bound data object. So if you want to show the UnitsInStock, UnitsOnOrder, and ReorderLevel fields, you need to make sure the SqlDataSource query returns this information.

To create the data binding expressions, the template uses the `Eval()` method, which is a static method of the `System.Web.UI.DataBinder` class. `Eval()` is an indispensable convenience: it automatically retrieves the data item that's bound to the current row, uses reflection to find the matching field, and retrieves the value.

■ **Tip** The `Eval()` method also adds the extremely useful ability to format data fields on the fly. To use this feature, you must call the overloaded version of the `Eval()` method that accepts an additional format string parameter. Here's an example:

```
<%# Eval("BirthDate", "{0:MM/dd/yy}") %>
```

You can use any of the format strings defined in Table 16-3 and Table 16-4 with the `Eval()` method.

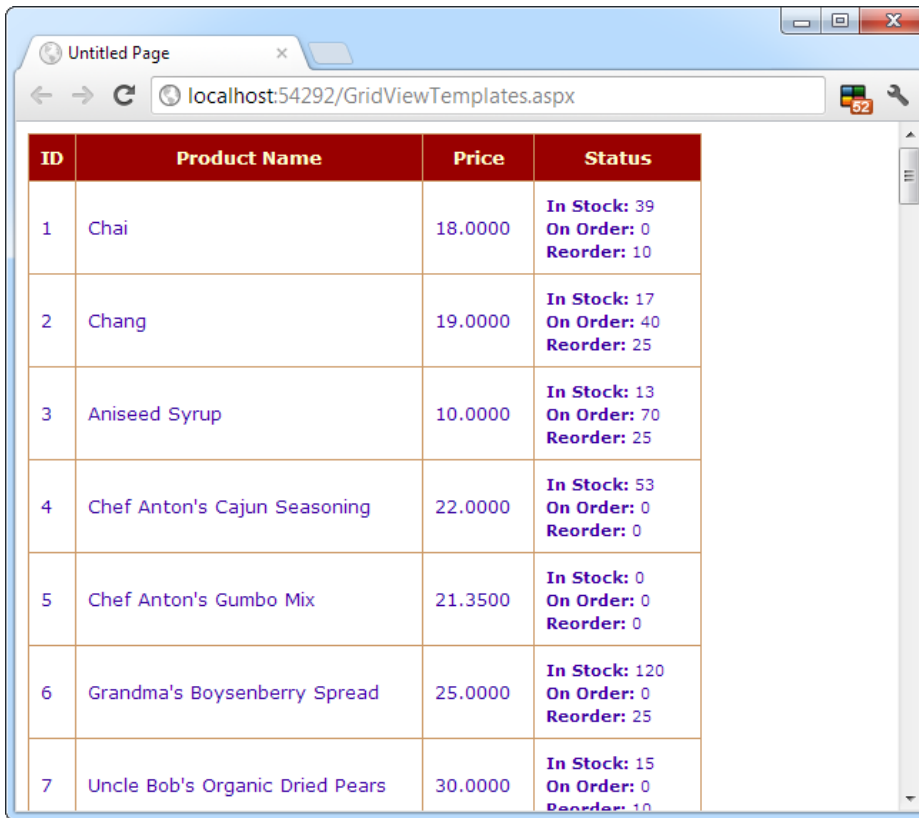
You'll notice that this example template includes three data binding expressions. These expressions get the actual information from the current row. The rest of the content in the template defines static text, tags, and controls.

You also need to make sure the data source provides these three pieces of information. If you attempt to bind a field that isn't present in your result set, you'll receive a runtime error. If you retrieve additional fields that are never bound to any template, no problem will occur.

Here's the revised data source with these fields:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
    ConnectionString="<%= $ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT ProductID, ProductName, UnitPrice, UnitsInStock,
UnitsOnOrder,ReorderLevel FROM Products"
    UpdateCommand="UPDATE Products SET ProductName=@ProductName,
UnitPrice=@UnitPrice WHERE ProductID=@ProductID">
</asp:SqlDataSource>
```

When you bind the `GridView`, it fetches the data from the data source and walks through the collection of items. It processes the `ItemTemplate` for each item, evaluates the data binding expressions, and adds the rendered HTML to the table. You're free to mix template columns with other column types. Figure 16-12 shows an example with several normal columns and the template column at the end.



ID	Product Name	Price	Status
1	Chai	18.0000	In Stock: 39 On Order: 0 Reorder: 10
2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25
3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25
4	Chef Anton's Cajun Seasoning	22.0000	In Stock: 53 On Order: 0 Reorder: 0
5	Chef Anton's Gumbo Mix	21.3500	In Stock: 0 On Order: 0 Reorder: 0
6	Grandma's Boysenberry Spread	25.0000	In Stock: 120 On Order: 0 Reorder: 25
7	Uncle Bob's Organic Dried Pears	30.0000	In Stock: 15 On Order: 0 Reorder: 10

Figure 16-12. A GridView with a template column

Using Multiple Templates

The previous example uses a single template to configure the appearance of data items. However, the `ItemTemplate` isn't the only template that the `TemplateField` provides. In fact, the `TemplateField` allows you to configure various aspects of its appearance with a number of templates. Inside every template column, you can use the templates listed in Table 16-7.

Table 16-7. *TemplateField Templates*

Mode	Description
HeaderTemplate	Determines the appearance and content of the header cell.
FooterTemplate	Determines the appearance and content of the footer cell (if you set ShowFooter to true).
ItemTemplate	Determines the appearance and content of each data cell.
AlternatingItemTemplate	Determines the appearance and content of even-numbered rows. For example, if you set the AlternatingItemTemplate to have a shaded background color, the GridView applies this shading to every second row.
EditItemTemplate	Determines the appearance and controls used in edit mode.
InsertItemTemplate	Determines the appearance and controls used in edit mode. The GridView doesn't support this template, but the DetailsView and FormView controls (which are described later in this chapter) do.

Of the templates listed in Table 16-7, the EditItemTemplate is one of the most useful because it gives you the ability to control the editing experience for the field. If you don't use template fields, you're limited to ordinary text boxes, and you won't have any validation. The GridView also defines two templates you can use outside any column. These are the PagerTemplate, which lets you customize the appearance of pager controls, and the EmptyDataTemplate, which lets you set the content that should appear if the GridView is bound to an empty data object.

Editing Templates in Visual Studio

Visual Studio includes solid support for editing templates in the web page designer. To try this, follow these steps:

1. Create a GridView with at least one template column.
2. Select the GridView, and click Edit Templates in the smart tag. This switches the GridView into template edit mode.
3. In the smart tag, use the Display drop-down list to choose the template you want to edit (see Figure 16-13). You can choose either of the two templates that apply to the whole GridView (EmptyDataTemplate or PagerTemplate), or you can choose a specific template for one of the template columns.

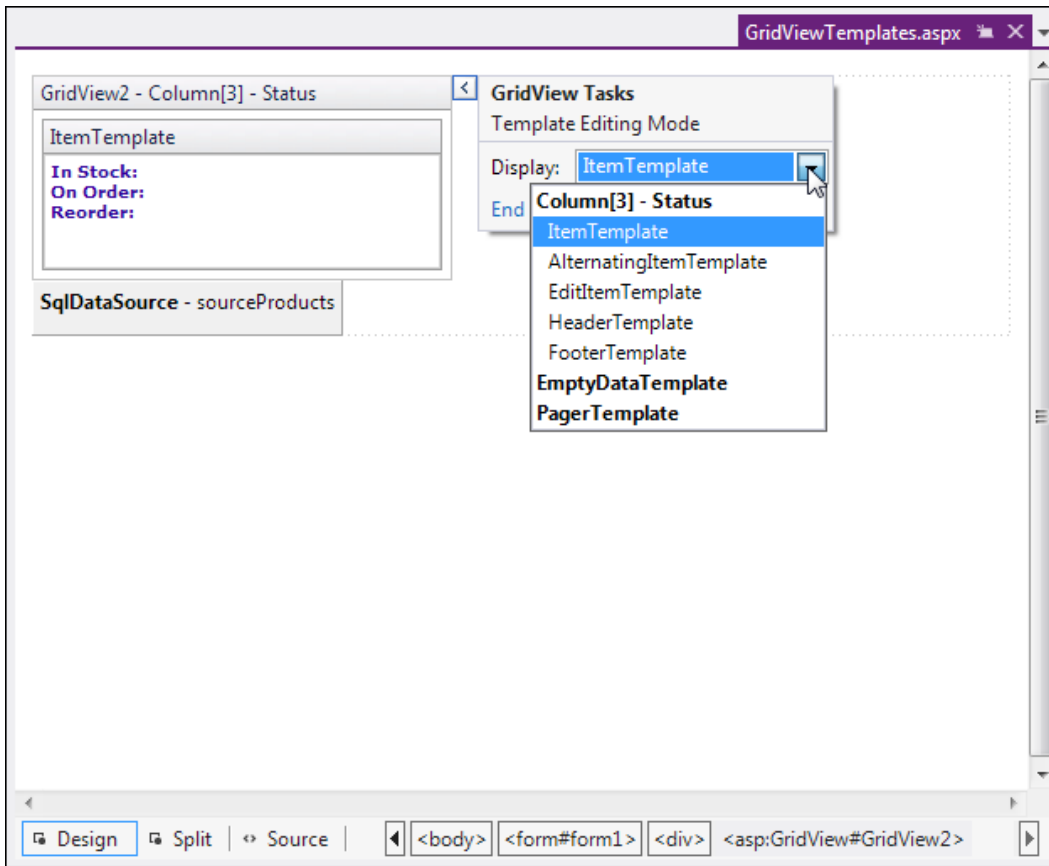


Figure 16-13. Editing a template in Visual Studio

4. Enter your content in the control. You can enter static content, drag-and-drop controls, and so on.
5. When you're finished, choose End Template Editing from the smart tag.

Handling Events in a Template

In some cases, you might need to handle events that are raised by the controls you add to a template column. For example, imagine you want to add a clickable image link by including an ImageButton control. This is easy enough to accomplish:

```
<asp:TemplateField HeaderText="Status">
  <ItemTemplate>
    <asp:ImageButton ID="ImageButton1" runat="server"
      ImageUrl="statuspic.gif" />
  </ItemTemplate>
</asp:TemplateField>
```

The problem is that if you add a control to a template, the GridView creates multiple copies of that control, one for each data item. When the ImageButton is clicked, you need a way to determine which image was clicked and to which row it belongs.

The way to resolve this problem is to use an event from the GridView, *not* the contained button. The GridView.RowCommand event serves this purpose because it fires whenever any button is clicked in any template. This process, where a control event in a template is turned into an event in the containing control, is called *event bubbling*.

Of course, you still need a way to pass information to the RowCommand event to identify the row where the action took place. The secret lies in two string properties that all button controls provide: CommandName and CommandArgument. CommandName sets a descriptive name you can use to distinguish clicks on your ImageButton from clicks on other button controls in the GridView. The CommandArgument supplies a piece of row-specific data you can use to identify the row that was clicked. You can supply this information using a data binding expression.

Here's a template field that contains the revised ImageButton tag:

```
<asp:TemplateField HeaderText="Status">
  <ItemTemplate>
    <asp:ImageButton ID="ImageButton1" runat="server"
      ImageUrl="statuspic.gif"
      CommandName="StatusClick" CommandArgument='<%= Eval("ProductID") %>' />
    </ItemTemplate>
  </asp:TemplateField>
```

And here's the code you need in order to respond when an ImageButton is clicked:

```
protected void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName == "StatusClick")
        lblInfo.Text = "You clicked product #" + e.CommandArgument.ToString();
}
```

This example displays a simple message with the ProductID in a label.

Editing with a Template

One of the best reasons to use a template is to provide a better editing experience. In the previous chapter, you saw how the GridView provides automatic editing capabilities—all you need to do is switch a row into edit mode by setting the GridView.EditIndex property. The easiest way to make this possible is to add a CommandField column with the ShowEditButton set to true. Then, the user simply clicks a link in the appropriate row to begin editing it. At this point, every label in every column is replaced by a text box (unless the field is read-only).

The standard editing support has several limitations:

It's not always appropriate to edit values using a text box: Certain types of data are best handled with other controls (such as drop-down lists). Large fields need multiline text boxes, and so on.

You get no validation: It would be nice to restrict the editing possibilities so that currency figures can't be entered as negative numbers, for example. You can do that by adding validator controls to an EditItemTemplate.

The visual appearance is often ugly: A row of text boxes across a grid takes up too much space and rarely seems professional.

In a template column, you don't have these issues. Instead, you explicitly define the edit controls and their layout using the `EditItemTemplate`. This can be a somewhat laborious process.

Here's the template column used earlier for stock information with an editing template:

```
<asp:TemplateField HeaderText="Status">
  <ItemStyle Width="100px" />
  <ItemTemplate>
    <b>In Stock:</b><%# Eval("UnitsInStock") %><br />
    <b>On Order:</b><%# Eval("UnitsOnOrder") %><br />
    <b>Reorder:</b><%# Eval("ReorderLevel") %>
  </ItemTemplate>
  <EditItemTemplate>
    <b>In Stock:</b><%# Eval("UnitsInStock") %><br />
    <b>On Order:</b><%# Eval("UnitsOnOrder") %><br /><br />
    <b>Reorder:</b>
    <asp:TextBox Text='<%# Bind("ReorderLevel") %>' Width="25px"
      runat="server" id="txtReorder" />
  </EditItemTemplate>
</asp:TemplateField>
```

Figure 16-14 shows the row in edit mode.

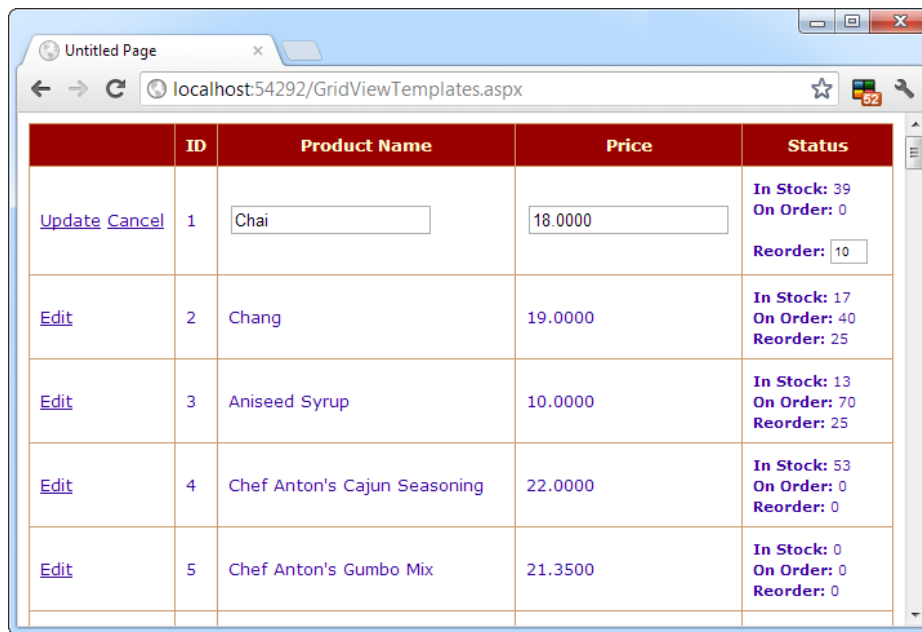


Figure 16-14. Using an edit template

When binding an editable value to a control, you must use the `Bind()` method in your data binding expression instead of the ordinary `Eval()` method. Unlike the `Eval()` method, which can be placed anywhere in a page, the `Bind()` method must be used to set a control property. Only the `Bind()` method creates the two-way link, ensuring that updated values will be returned to the server.

One interesting detail here is that even though the item template shows three fields, the editing template allows only one of these to be changed. When the GridView commits an update, it will submit only the bound, editable parameters. In the previous example, this means the GridView will pass back a `@ReorderLevel` parameter but *not* a `@UnitsInStock` or `@UnitsOnOrder` parameter. This is important because when you write your parameterized update command, it must use only the parameters you have available. Here's the modified `SqlDataSource` control with the correct command:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT ProductID, ProductName, UnitPrice, UnitsInStock,
UnitsOnOrder,ReorderLevel FROM Products"
    UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
ReorderLevel=@ReorderLevel WHERE ProductID=@ProductID">
</asp:SqlDataSource>
```

Editing with Validation

Now that you have your template ready, why not add a frill, such as a validator, to catch editing mistakes? In the following example, a `RangeValidator` prevents changes that put the `ReorderLevel` at less than 0 or more than 100:

```
<asp:TemplateField HeaderText="Status">
    <ItemStyle Width="100px" />
    <ItemTemplate>
        <b>In Stock:</b><%# Eval("UnitsInStock") %><br />
        <b>On Order:</b><%# Eval("UnitsOnOrder") %><br />
        <b>Reorder:</b><%# Eval("ReorderLevel") %>
    </ItemTemplate>
    <EditItemTemplate>
        <b>In Stock:</b><%# Eval("UnitsInStock") %><br />
        <b>On Order:</b><%# Eval("UnitsOnOrder") %><br /><br />
        <b>Reorder:</b>
        <asp:TextBox Text="<%# Bind("ReorderLevel") %>" Width="25px"
            runat="server" id="txtReorder" />
        <asp:RangeValidator id="rngValidator" MinimumValue="0" MaximumValue="100"
            ControlToValidate="txtReorder" runat="server"
            ErrorMessage="Value out of range." Type="Integer"/>
    </EditItemTemplate>
</asp:TemplateField>
```

Figure 16-15 shows the validation at work. If the value isn't valid, the browser doesn't allow the page to be posted back, and no database code runs.

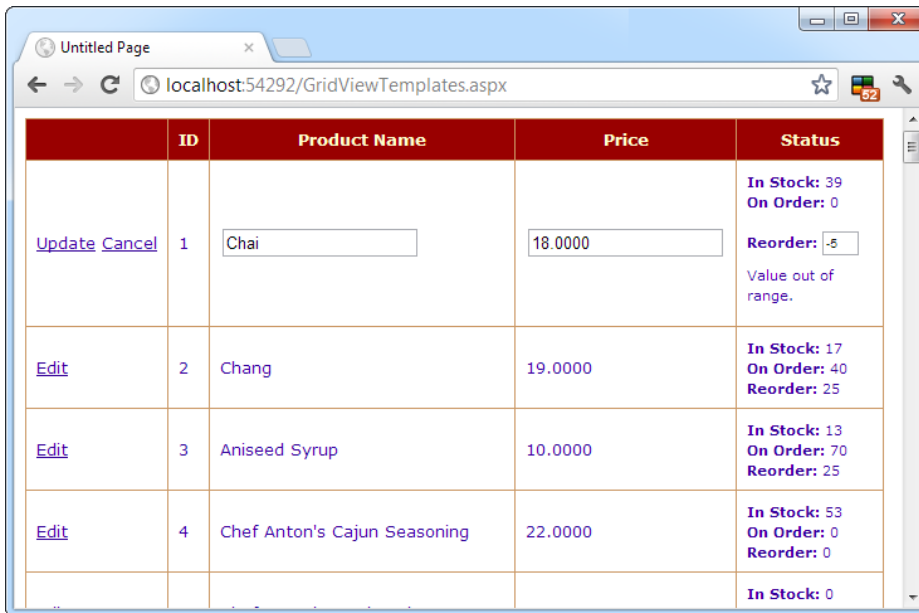


Figure 16-15. Creating an edit template with validation

Note The `SqlDataSource` is intelligent enough to handle validation properly even if you have disabled client-side validation (or the browser doesn't support it). In this situation, the page is posted back, but the `SqlDataSource` notices that it contains invalid data (by inspecting the `Page.IsValid` property), and doesn't attempt to perform its update. For more information about client-side and server-side validation, refer to Chapter 9.

Editing Without a Command Column

So far, all the examples you've seen have used a `CommandField` that automatically generates edit controls. However, now that you've made the transition to a template-based approach, it's worth considering how you can add your own edit controls.

It's actually quite easy. All you need to do is add a button control to the item template and set the `CommandName` to `Edit`. This automatically triggers the editing process, which fires the appropriate events and switches the row into edit mode.

```
<ItemTemplate>
  <b>In Stock:</b><# Eval("UnitsInStock") %><br />
  <b>On Order:</b><# Eval("UnitsOnOrder") %><br />
  <b>Reorder:</b><# Eval("ReorderLevel") %>
<br /><br />
  <asp:LinkButton runat="server" Text="Edit"
    CommandName="Edit" ID="LinkButton1" />
</ItemTemplate>
```

In the edit item template, you need two more buttons with CommandName values of Update and Cancel:

```
<EditItemTemplate>
<b>In Stock:</b><# Eval("UnitsInStock") %><br />
<b>On Order:</b><# Eval("UnitsOnOrder") %><br /><br />
<b>Reorder:</b>
<asp:TextBox Text='<# Bind("ReorderLevel") %>' Width="25px"
  runat="server" id="txtReorder" />
<br /><br />
<asp:LinkButton runat="server" Text="Update"
  CommandName="Update" ID="LinkButton1" />
<asp:LinkButton runat="server" Text="Cancel"
  CommandName="Cancel" ID="LinkButton2" CausesValidation="False" />
</EditItemTemplate>
```

Notice that the Cancel button must have its CausesValidation property set to false to bypass validation. That way, you can cancel the edit even if the current data aren't valid.

As long as you use these names, the GridView editing events will fire and the data source controls will react in the same way as if you were using the automatically generated editing controls. Figure 16-16 shows the custom edit buttons.

ID	Product Name	Price	Status
1	<input type="text" value="Chai"/>	<input type="text" value="18.0000"/>	In Stock: 39 On Order: 0 Reorder: <input type="text" value="10"/> Update Cancel
2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25 Edit
3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25

Figure 16-16. Custom edit controls

The DetailsView and FormView

The GridView excels at showing a dense table with multiple rows of information. However, sometimes you want to provide a detailed look at a single record. You could work out a solution using a template column in a GridView, but ASP.NET includes two controls that are tailored for this purpose: the DetailsView and the

FormView. Both show a single record at a time but can include optional pager buttons that let you step through a series of records (showing one per page). Both give you an easy way to insert a new record, which the GridView doesn't allow. And both support templates, but the FormView *requires* them. This is the key distinction between the two controls.

One other difference is the fact that the DetailsView renders its content inside a table, while the FormView gives you the flexibility to display your content without a table. Thus, if you're planning to use templates, the FormView gives you the most flexibility. But if you want to avoid the complexity of templates, the DetailsView gives you a simpler model that lets you build a multirow data display out of field objects, in much the same way that the GridView is built out of column objects.

Now that you understand the features of the GridView, you can get up to speed with the DetailsView and the FormView quite quickly. That's because both borrow a portion of the GridView model.

The DetailsView

The DetailsView displays a single record at a time. It places each field in a separate row of a table.

You saw in Chapter 15 how to create a basic DetailsView to show the currently selected record. The DetailsView also allows you to move from one record to the next using paging controls, if you've set the AllowPaging property to true. You can configure the paging controls using the PagerStyle and PagerSettings properties in the same way as you tweak the pager for the GridView.

Figure 16-17 shows the DetailsView when it's bound to a set of product records, with full product information.

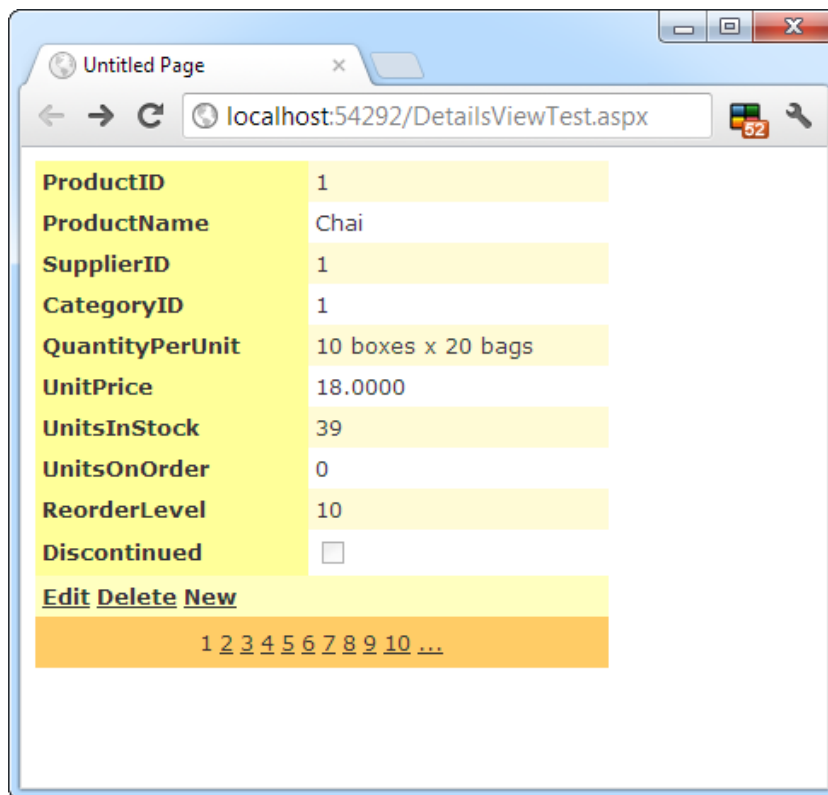


Figure 16-17. The DetailsView with paging

It's tempting to use the DetailsView pager controls to make a handy record browser. Unfortunately, this approach can be quite inefficient. One problem is that a separate postback is required each time the user moves from one record to another (whereas a grid control can show multiple records on the same page). But the real drawback is that each time the page is posted back, the full set of records is retrieved, even though only a single record is shown. This results in needless extra work for the database server. If you choose to implement a record browser page with the DetailsView, at a bare minimum you must enable caching to reduce the database work (see Chapter 23).

■ **Tip** It's almost always a better idea to use another control to let the user choose a specific record (for example, by choosing an ID from a list box), and then show the full record in the DetailsView using a parameterized command that matches just the selected record. Chapter 15 demonstrates this technique.

Defining Fields

The DetailsView uses reflection to generate the fields it shows. This means it examines the data object and creates a separate row for each field it finds, just like the GridView. You can disable this automatic row generation by setting `AutoGenerateRows` to false. It's then up to you to declare information you want to display.

Interestingly, you use the same field tags to build a DetailsView as you use to design a GridView. For example, fields from the data item are represented with the `BoundField` tag, buttons can be created with the `ButtonField`, and so on. For the full list, refer to the earlier Table 16-1.

The following code defines a DetailsView that shows product information. This tag creates the same grid of information shown in Figure 16-17, when `AutoGenerateRows` was set to true.

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
DataSourceID="sourceProducts">
  <Fields>
    <asp:BoundField DataField="ProductID" HeaderText="ProductID"
      ReadOnly="True" />
    <asp:BoundField DataField="ProductName" HeaderText="ProductName" />
    <asp:BoundField DataField="SupplierID" HeaderText="SupplierID" />
    <asp:BoundField DataField="CategoryID" HeaderText="CategoryID" />
    <asp:BoundField DataField="QuantityPerUnit" HeaderText="QuantityPerUnit" />
    <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice" />
    <asp:BoundField DataField="UnitsInStock" HeaderText="UnitsInStock" />
    <asp:BoundField DataField="UnitsOnOrder" HeaderText="UnitsOnOrder" />
    <asp:BoundField DataField="ReorderLevel" HeaderText="ReorderLevel" />
    <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued" />
  </Fields>
  ...
</asp:DetailsView>
```

You can use the `BoundField` tag to set properties such as header text, formatting string, editing behavior, and so on (refer to Table 16-2). In addition, you can use the `ShowHeader` property. When it's false, the header text is left out of the row, and the field data take up both cells.

■ **Tip** Rather than coding each field by hand, you can use the same shortcut as you used with the GridView. Simply select the control at design time, and select `Refresh Schema` from the smart tag.

The field model isn't the only part of the GridView that the DetailsView control adopts. It also uses a similar set of styles, a similar set of events, and a similar editing model. The only difference is that instead of creating a dedicated column for editing controls, you simply set one of the Boolean properties of the DetailsView, such as `AutoGenerateDeleteButton`, `AutoGenerateEditButton`, and `AutoGenerateInsertButton`. The links for these tasks are added to the bottom of the DetailsView. When you add or edit a record, the DetailsView uses standard text box controls (see Figure 16-18), just as the GridView does. For more editing flexibility, you'll want to use templates with the DetailsView (by adding a `TemplateField` instead of a `BoundField`) or the `FormView` control (as described next).

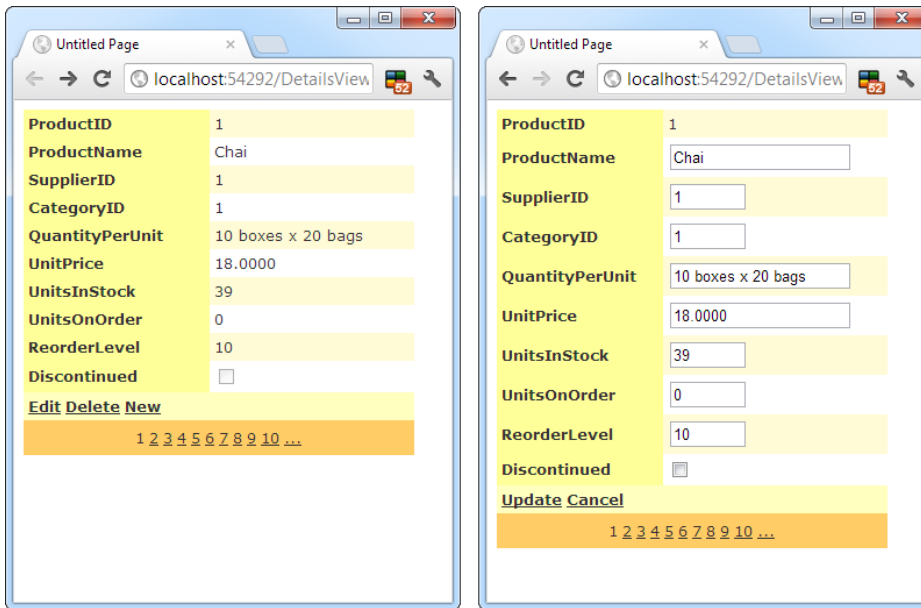


Figure 16-18. Editing in the DetailsView

The FormView

If you need the ultimate flexibility of templates, the `FormView` provides a template-only control for displaying and editing a single record.

The beauty of the `FormView` template model is that it matches quite closely the model of the `TemplateField` in the `GridView`. This means you can work with the following templates:

- `ItemTemplate`
- `EditItemTemplate`
- `InsertItemTemplate`
- `FooterTemplate`
- `HeaderTemplate`
- `EmptyDataTemplate`
- `PagerTemplate`

Note Unlike the `GridView` and `DetailsView`, which allow you to add as many `TemplateField` objects as you want, the `FormView` allows just a single copy of each template. If you want to show multiple values, you must add multiple binding expressions to the same `ItemTemplate`.

You can use the same template content you use with a `TemplateField` in a `GridView` in the `FormView`. Earlier in this chapter, you saw how you can use a template field to combine the stock information of a product into one column (as shown in Figure 16-12). Here's how you can use the same template in the `FormView`:

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="sourceProducts">
  <ItemTemplate>
    <b>In Stock:</b>
    <%# Eval("UnitsInStock") %>
    <br />
    <b>On Order:</b>
    <%# Eval("UnitsOnOrder") %>
    <br />
    <b>Reorder:</b>
    <%# Eval("ReorderLevel") %>
    <br />
  </ItemTemplate>
</asp:FormView>
```

Like the `DetailsView`, the `FormView` can show a single record at a time. (If the data source has more than one record, you'll see only the first one.) You can deal with this issue by setting the `AllowPaging` property to `true` so that paging links are automatically created. These links allow the user to move from one record to the next, as in the previous example with the `DetailsView`.

Another option is to bind to a data source that returns just one record. Figure 16-19 shows an example where a drop-down list control lets you choose a product, and a second data source shows the matching record in the `FormView` control.

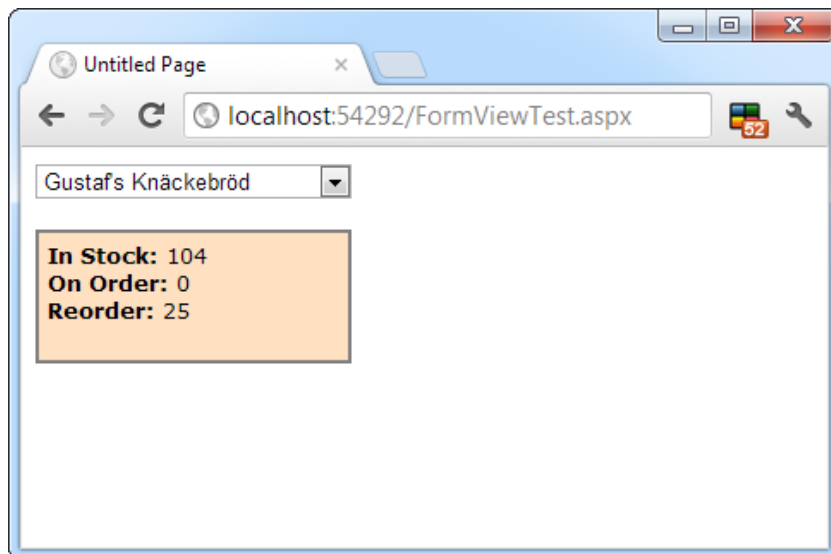


Figure 16-19. A `FormView` that shows a single record

Here's the markup you need to define the drop-down list and its data source:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
  ConnectionString="<%$ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductID, ProductName FROM Products">
</asp:SqlDataSource>

<asp:DropDownList ID="lstProducts" runat="server"
  AutoPostBack="True" DataSourceID="sourceProducts"
  DataTextField="ProductName" DataValueField="ProductID" Width="184px">
</asp:DropDownList>
```

The FormView uses the template from the previous example (it's the shaded region on the page). Here's the markup for the FormView (not including the template) and the data source that gets the full details for the selected product.

```
<asp:SqlDataSource ID="sourceProductFull" runat="server"
  ConnectionString="<%$ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID">
  <SelectParameters>
    <asp:ControlParameter Name="ProductID"
      ControlID="lstProducts" PropertyName="SelectedValue" />
  </SelectParameters>
</asp:SqlDataSource>

<asp:FormView ID="formProductDetails" runat="server"
  DataSourceID="sourceProductFull"
  BackColor="#FFE0C0" CellPadding="5">
  <ItemTemplate>
    ...
  </ItemTemplate>
</asp:FormView>
```

■ **Note** If you want to support editing with the FormView, you need to add button controls that trigger the edit and update processes, as described in the “Editing with a Template” section.

The Last Word

In this chapter, you considered everything you need to build rich data-bound pages. You took a detailed tour of the GridView and considered its support for formatting, selecting, sorting, paging, using templates, and editing. You also considered the DetailsView and the FormView, which allow you to display and edit individual records. Using these three controls, you can build all-in-one pages that display and edit data without needing to write pages of ADO.NET code. Best of all, every data control is thoroughly configurable, which means you can tailor it to fit just about any web application.