8

Reading from Databases

So far, you've learnt a lot about programming, and seen those techniques in use in a variety of Web pages. Now it's time to turn our attention to one of the most important topics of building Web sites – data. Whatever the type of site you aim to build, data plays an important part. From a personal site (perhaps a vacation diary or a photo album), to a corporate e-commerce site, *data is key!*

There are numerous ways to store data, but most sites use a *database*. In this chapter, we're going to look at data stored in databases, and show how easily it can be used on Web pages. For this we are going to use ADO.NET, which is the data access technology that comes as part of the .NET Framework.

If the thought of databases sounds complex and scary, don't worry. We're going to show you just how easy this can be. In particular, we'll be looking at:

- Basics of databases and how they work
- How to create simple data pages using Web Matrix
- Different ADO.NET classes used for fetching data
- Basics of ADO.NET and how it fetches data
- How to use Web Matrix to simplify developing data access pages

Let's develop some basic understanding of databases first.

Understanding Databases

Understanding some basics about databases is crucial to using data in your pages. You don't need to be a database expert, but there are certain things you will need to know in order to work with data in .NET. For a start, you need to understand how data is stored. All types of data on a computer are stored in files of some sort. Text files, for example, are simple files and just contain

plain text. Spreadsheets, on the other hand, are complex files containing not only the entered text and numbers, but also details about the data, such as what the columns contain, how they are formatted, and so on.

Databases also fall into the category of complex files. When using Microsoft Access, you have an MDB file – this is a database file, but you can't tell anything about the data from the file itself. You need a way to get to the data, either using Microsoft Access itself, or as we are going to do, using the .NET data classes. Before you can access the data, you need to know how it is stored internally.

Tables

Within a database, data is stored in *tables* – these are the key components of all databases. A table is like a spreadsheet, with *rows* and *columns*. You generally have multiple tables for multiple things – each distinct type of data is stored separately, and tables are often linked together.

Let's look at an example that should make this easier to visualize. Consider an ordering system, for example, where you store details of customers and the goods they've ordered. The following table shows rows of customer orders, with columns (or *fields*) each piece of order information:

Customer	Address	Order Date	Order Item	Quantity	Item Cost
John	15 High Street Brumingham England UK	01/07/2003	Widget	10	3.50
John	15 High Street Brumingham England UK	01/07/2003	Doodad	5	2.95
John	15 High Street Brumingham England UK	01/08/2003	Thingy	1	15.98
Chris	25 Easterly Way Cradiff Wales UK	01/08/2003	Widget	1	3.50
Dave	2 Middle Lane Oxborough England UK	01/09/2003	Doodad	2	2.95

Customer	Address	Order Date	Order Item	Quantity	Item Cost
Dave	3 Middle Lane Oxborough England UK	01/09/2003	Thingamajig	1	8.50

This is the sort of thing you'd see in a spreadsheet, but there are a couple of big problems with this. For a start, we have repeated information. John, for example, has his address shown three times. What happens if he moves house? You'd have to change the address everywhere it occurs. Dave has two addresses, but notice they are slightly different. Which one is correct? Are neither correct?

To get around these problems, we use a process called *Normalization*.

Normalization

This is the process of separating repeated information into separate tables. There are whole books dedicated to database design, but we only need to look at the simplest case. A good beginner book on database design is *Database Design for Mere Mortals: A Hands On Guide to Relational Database Design, by Michael J. Hernandez*

What we need to do is split the previous table into three tables, one for each unique piece of information – `Customers`, `Orders`, and `OrderDetails`. To link the three new tables together, we create ID columns that uniquely identify each row. For example, we could create a column called `CustomerID` in the `Customers` table. To link the `Customers` table to the `Orders` table, we also add this `CustomerID` to the `Orders` table. Let's look at our tables now.

The `Customers` table is as follows:

CustomerID	Customer	Address
1	John	15 High Street Brumingham England UK
2	Chris	25 Easterly Way Cradiff Wales UK
3	Dave	2 Middle Lane Oxborough England UK

The `Orders` table is as follows:

OrderID	CustomerID	OrderDate
1	1	01/07/2003
2	1	01/08/2003
3	2	01/08/2003
4	3	01/09/2003

The `OrderDetails` table is as follows:

OrderDetailsID	OrderID	Order Item	Quantity	Item Cost
1	1	Widget	10	3.50
2	1	Doodad	5	2.95
3	2	Thingy	1	15.98
4	3	Widget	1	3.50
5	4	Doodad	2	2.95
6	4	Thingamajig	1	8.50

We now have three tables that can be linked together by their ID fields as shown in Figure 8-1:

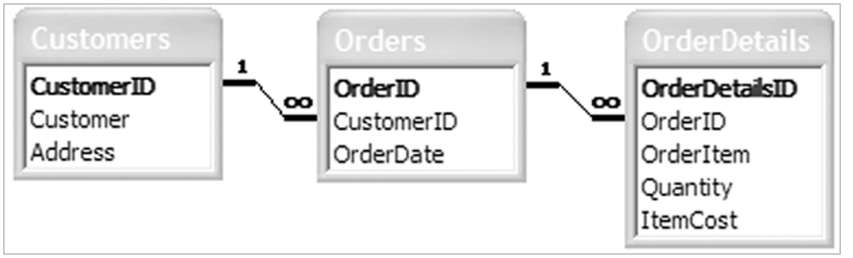


Figure 8-1

We now have links between the tables. The `CustomerID` field in the `Orders` table is used to identify which customer the order is for. Similarly, the `OrderID` field in the `OrderDetails` table identifies which order a particular order line belongs to.

The unique key in a table is defined as its *Primary Key* – it's what uniquely defines a row. When used in another table it is called the *Foreign Key*, so called because it's a key, but one to a foreign table. The

foreign key is simply a column that is the primary key in another table. Because the values of the primary key and the foreign key will be the same, we can use them to link the tables together. This linking of the tables is done in *Structured Query Language (SQL)*, usually as a query or a stored procedure.

SQL and Stored Procedures

Queries are the way in which we deal with data in a database, either to extract data or to manipulate it. We can use an SQL statement or a stored procedure, which is an SQL statement wrapped to provide a simple name. It's worth noting that a stored procedure is actually more than just wrapping an SQL statement in a name, but that's a good enough description for what we need.

In *Chapter 5* when we looked at functions, we had a function name encapsulating some code statements. Think of a stored procedure in a similar way – it wraps a set of SQL statements, allowing us to use the name of the stored procedure to run those SQL statements. We're not going to focus much on this topic as it's outside the scope of this book.

To learn more about SQL, read *SQL for Dummies (ISBN 0-7645-4075-0) by John Wiley & Sons Inc*.

Here are a few reasons why you should always use stored procedures instead of direct SQL:

- **Security**: Using the .NET data classes with stored procedures protects you against certain forms of hacking.
- **Speed**: Stored procedures are optimised the first time they are called, and then the optimised code is used in subsequent calls.
- **Separation**: It keeps the SQL separate from your code.

In the remainder of this book, we'll actually be using a mixture of SQL and stored procedures for the simple reason that sometimes it's easier to use SQL in the context of an example. Remember, our focus is on ASP.NET. We'll be using Microsoft Access for the samples, and although Access doesn't support stored procedures, its use of stored queries is equivalent.

Let's get on with some examples.

The Web Matrix Data Explorer

You've already seen how powerful Web Matrix is for creating Web pages, and this power extends to working with data. Where you've used the Workspace Explorer in the top right hand corner of Web Matrix to work with files, you can use the Data Explorer to work with data. This provides ways of creating databases, connecting to existing ones, and working with tables and queries. Let's give this a go.

Try It Out Connecting to a Database

1. Select the Data Explorer tab, and click the Add Database Connection button – the one that's second in from the right, and will be the only one highlighted, as shown in Figure 8-2, if you haven't already got a database connection open:

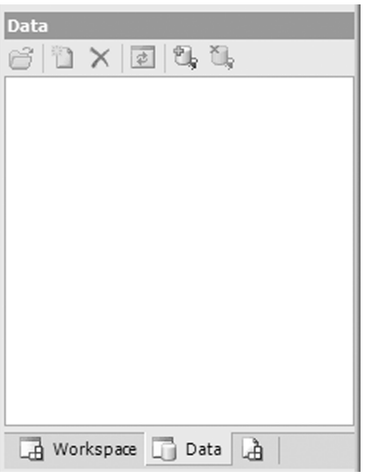


Figure 8-2

2. Select Access Database from the window that appears and press OK.
3. Enter the following into the Data File text area (use a central location for the database, so that we can reuse it later in the book):

```
C:\BegASPNET11\data\Northwind.mdb
```

4. Press OK to connect to the database. This is the Northwind database, one of the sample databases that ships with Microsoft Access.
5. Figure 8-3 shows the tables contained in this database:

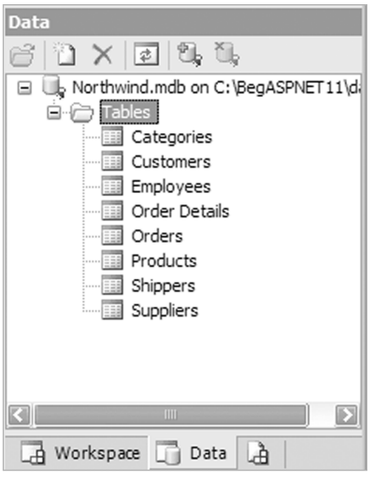


Figure 8-3

You can double-click on these to open the table, and see and change the data. One thing you might notice is that you don't see any queries – that's because Web Matrix doesn't support queries in Access. When connecting to SQL Server, you'll see the stored procedures – you can even create and edit them – but for Access, you are limited to tables only.

How It Works

There's nothing really to explain about how it works. What we are doing is simply creating a connection to a database that Web Matrix can use. This isn't required for ASP.NET to fetch data from databases, but Web Matrix has some great ways to generate code for you, so you don't have to do as much coding.

Creating Data Pages

Pages that display data can be created in a number of ways, and let's first look at the three ways that Web Matrix uses to save you coding. This is the quickest way to get data into your pages and saves a great deal of time. However, what it might not do is give you the knowledge to access databases without using Web Matrix. After we've seen the easy ways, we'll look at the .NET classes that deal with data. This way you'll have techniques to work with and without Web Matrix.

Displaying Data Using the Data Explorer

You've already seen how easy connecting to a database is using the Data Explorer. Creating pages directly from this explorer is even easier – all you have to do is drag the table name and drop it onto a page. This will automatically create a connection on the page and a fully functional data grid. Let's give this a go.

Try It Out Creating a Grid

1. Create a new ASP.NET page called `Grid1.aspx`.
2. From the Data Explorer, drag the `Suppliers` table onto your empty page as shown in Figure 8-4:

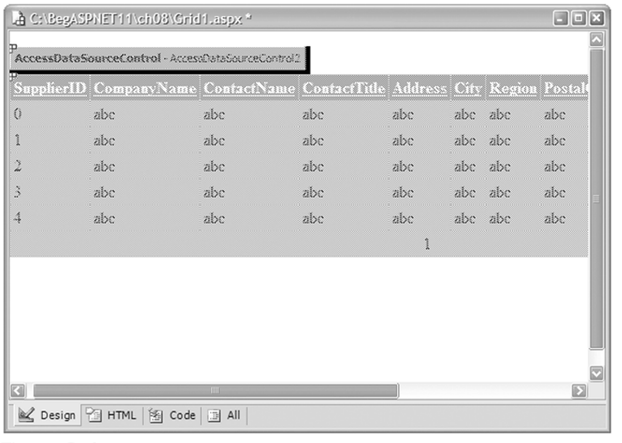
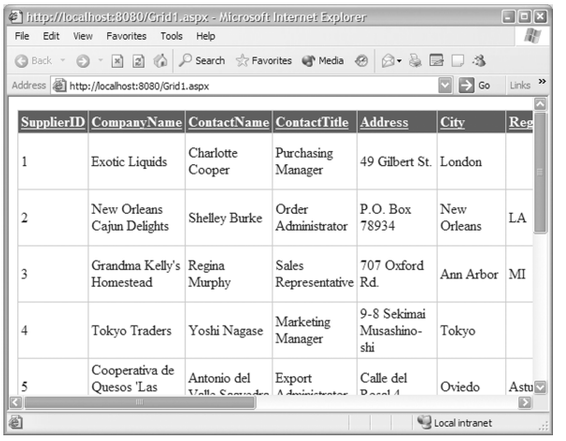


Figure 8-4

3. Save the page and run it as shown in Figure 8-5:



SupplierID	CompanyName	ContactName	ContactTitle	Address	City	Reg
1	Exotic Liquids	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	
2	New Orleans Cajun Delights	Shelley Burke	Order Administrator	P.O. Box 78934	New Orleans	LA
3	Grandma Kelly's Homestead	Regina Murphy	Sales Representative	707 Oxford Rd.	Ann Arbor	MI
4	Tokyo Traders	Yoshi Nagase	Marketing Manager	9-8 Sekimai Musashino-shi	Tokyo	
5	Cooperativa de Quesos 'Las	Antonio del	Export	Calle del	Oviedo	Astu

Figure 8-5

Amazing! A sortable grid full of data and you didn't have to write even a single line of code!

How It Works

The workings rely on two controls – the `AccessDataSourceControl` that provides the connection to the database, and an `MxDataGrid`, which is a Web Matrix control (also covered in *Chapter 10*) that displays the data. Looking at the HTML view for these controls gives you a good idea of what they do.

Let's start with the `AccessDataSourceControl`:

```
<wmx:AccessDataSourceControl id="AccessDataSourceControl2"
  runat="server" SelectCommand="SELECT * FROM [Suppliers]"
  ConnectionString="Provider=Microsoft.Jet.OLEDB.4.0; Ole DB Services=-4;
Data Source=C:\BegASPNET11\data\Northwind.mdb"></wmx:AccessDataSourceControl>
```

The first thing to notice is the way the control is declared. You're used to seeing `asp:` at the beginning of controls, but not `wmx:`. This prefix is the namespace – remember the previous chapter where we said that namespaces provide a separation between classes. In this case, these controls are part of Web Matrix, and have thus been given a namespace that is different from the standard server controls.

Apart from the `id` and `runat`, two other attributes provide the details regarding which database to connect to and what data to fetch:

- The `SelectCommand`: Defines the SQL that will return the required data – in this case, it's all rows and columns from the `Suppliers` table. This is the default since we dragged this table, but we can customize the `SelectCommand` to return only selected rows or columns.
- The `ConnectionString`: Defines the OLEDB connection string. You only need to worry about the bit with the path of the database file – the `Data Source` bit (if you move the file, you'll need to change this). The other parts of the `ConnectionString` just define the type of database and

some database specific features. You don't need to know about these specifically (they are fully documented in the .NET help files); just copy them if you ever need to use them again.

At this stage, you have enough details to connect to a database and fetch data, but don't have any way to *display* it. For that we are going to use the `MxDataGrid` control:

```
<wmx:MxDataGrid id="MxDataGrid2" runat="server"
  DataSourceControlID="AccessDataSourceControl2" BorderColor="#CCCCCC"
  AllowSorting="true" DataMember="Suppliers" AllowPaging="true"
  BackColor="White" CellPadding="3" DataKeyField="SupplierID"
  BorderWidth="1px" BorderStyle="None">
    <PagerStyle horizontalalign="Center" forecolor="#000066"
      backcolor="White" mode="NumericPages"></PagerStyle>
    <FooterStyle forecolor="#000066" backcolor="White"></FooterStyle>
    <SelectedItemStyle font-bold="true" forecolor="White"
      backcolor="#669999"></SelectedItemStyle>
    <ItemStyle forecolor="#000066"></ItemStyle>
    <HeaderStyle font-bold="true" forecolor="White"
      backcolor="#006699"></HeaderStyle>
</wmx:MxDataGrid>
```

This may seem complex but is actually very simple. Let's look at all of the attributes:

Attribute	Description
`DataSourceControlID`	This contains the ID of the data source control from which data will be fetched. In this case, it's the ID of the `AccessDataSourceControl` we described earlier.
`BorderColor`	This is the color of the grid border.
`AllowSorting`	Indicates whether or not the grid will support sorting.
`DataMember`	This contains the database table name.
`AllowPaging`	Indicates whether or not the grid supports paging. The default number of rows in a page is `10`, and this can be changed with the `PageSize` attribute.
`BackColor`	This is the background color for the grid.
`CellPadding`	This defines the amount of padding between grid cells. A higher number means the cells will be spaced further apart.
`DataKeyField`	This is the primary key of the table.
`BorderWidth`	This is how wide the border of the grid is. Here it is `1` pixel (`px` stands for pixel), which is a thin border.
`BorderStyle`	This is the style of the border.

As part of the grid, we also have some style elements:

- `PagerStyle`: Defines the style of the pager section. In our grid, this is the last row showing the page numbers, but it appears before the footer if a footer row is being shown.
- `FooterStyle`: Defines the style of the footer row. In our grid, we aren't showing a footer, but the style is set so that the footer will look correct if it is shown.
- `SelectedItemStyle`: Defines the style of items when they are selected. Our grid isn't selectable by default, but like the `FooterStyle` the default style is set in case item selection is added.
- `ItemStyle`: Defines the style for each row of data in the grid.
- `HeaderStyle`: Defines the style for the header row, where the column names are shown.

That's all there is to this example – two controls that are linked together. When the page is loaded, the `AccessDataSourceControl` connects to the database and runs the command. The `MxDataGrid` then fetches the data stored by the data source control and constructs a grid around it. In fact, the grid is the most complex piece of code here because of all the properties being set - purely to change the look. At its simplest, you could have the following:

```
<wmx:MxDataGrid id="MxDataGrid2" runat="server"
  DataSourceControlID="AccessDataSourceControl2">
</wmx:MxDataGrid>
```

This only contains the attributes required to display data.

Displaying Data Using the Web Matrix Template Pages

You've probably noticed a number of *template pages* when you add a new page in Web Matrix – some of those are for data reports. These provide a simple way to get more functionality into grids than the example earlier used.

The supplied template pages are as follows:

- Simple Data Report: Gives a simple grid without paging or sorting
- Filtered Data Report: Gives a grid with a filter option, so you can select the rows displayed
- Data Report with Paging: Gives a grid with paging enabled
- Data Report with Paging and Sorting: Gives a grid with paging and column sorting enabled
- Master – Detail Grids: Gives two grids, representing a master table and a child table
- Editable Grid: Gives a grid allowing updates to the data
- Simple Stored Procedure: Gives a grid that uses a stored procedure for its data source

All of these supplied templates connect to a SQL Server database, and need modification if they are to be used with a different database. However, they provide a quick way to get pages constructed, allowing you to make a few simple changes to get what you need, rather than coding from scratch.

Let's look at one of these - the report with paging and sorting.

Try It Out Creating a Data Page

1. Create a new page using the Data Pages templates. Pick the Data Report with Paging and Sorting, and call it `SortPage.aspx`.

2. In the design window, select the All tab and change this line:

```
<%@ import Namespace="System.Data.SqlClient" %>
```

   To:

```
<%@ import Namespace ="System.Data.OleDb" %>
```

   If this is not done, errors will be encountered while loading the page.

3. In the design window, select the Code tab, find the `BindGrid()` subroutine, and change the code so it looks like the following:

```
void BindGrid()
{
  // TODO: update the ConnectionString value for your application
  string ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
                            "Data Source=C:\BegASPNet11\data\Northwind.mdb";
  string CommandText;

  // TODO: update the CommandText value for your application
  if (SortField == String.Empty)
    CommandText = "select * from Suppliers order by CompanyName";
  else
    CommandText = "select * from Suppliers order by " + SortField;

  OleDbConnection myConnection = new OleDbConnection(ConnectionString);
  OleDbDataAdapter myCommand = new OleDbDataAdapter(CommandText,
                                                    myConnection);

  DataSet ds = new DataSet();
  myCommand.Fill(ds);

  DataGrid1.DataSource = ds;
  DataGrid1.DataBind();
}
```

   Use a different path if you've installed the samples in a directory other than `C:\BegASPNET11`.

4. Save the file and run it; you'll see something like Figure 8-6:

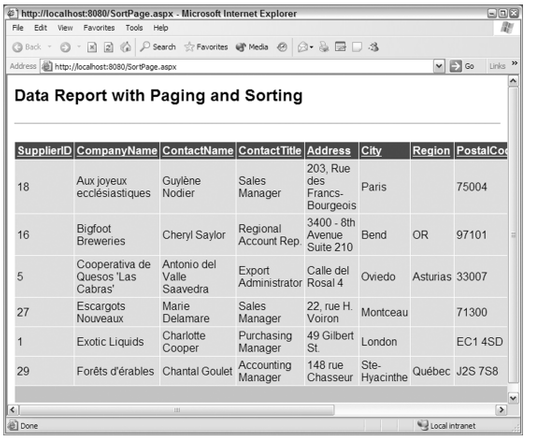


Figure 8-6

This isn't much different from the drag and drop approach we used in the first example, but it uses the .NET data classes and a `DataGrid` control, rather than the Web Matrix controls (`AccessDataSourceControl` and `MxDataGrid`). It means this technique will work even if Web Matrix isn't installed on the server running the page. Let's see how it works.

How It Works

The first thing to look at is the namespace change:

```
<%@ import Namespace="System.Data.OleDb" %>
```

By default, the data pages are configured to use SQL Server and therefore use the `SqlClient` namespace. Since we are using Access, we have to use the `OleDb` namespace.

Now let's look at the declaration of the grid itself. We won't show all the properties, as some are to do with the visual style. Instead, we'll concentrate on those that are related to the code we'll see:

```
<asp:datagrid id="DataGrid1" runat="server"
    AllowPaging="true" PageSize="6" OnPageIndexChanged="DataGrid_Page"
    AllowSorting="true" OnSortCommand="DataGrid_Sort">
```

Here we have the following properties defined:

- `AllowPaging`: When set to `true`, allows the grid to page the results. This works in a way similar to the `MxDataGrid` where the page numbers are shown at the bottom of the grid.
- `PageSize`: Defines the number of rows to show per page.
- `OnPageIndexChanged`: Defines the event procedure to call when the page number is changed. When a page number link is clicked, the procedure defined here is run.

- `AllowSorting`: Allows the grid to sort the rows on the basis of column selections. Setting this to `true` enables links on the column headings.
- `OnSortCommand`: Defines the event procedure to call when a column heading is clicked.

Now let's look at the code that uses this grid, starting with the `Page_Load()` event:

```
void Page_Load(object sender, EventArgs e)
{
  if (!Page.IsPostBack) {

    // Databind the data grid on the first request only
    // (on postback, rebind only in paging and sorting commands)

    BindGrid();
  }
}
```

Here we are calling the `BindGrid()` routine, but only if this is the first time the page has been loaded. This ensures that the grid, in its initial state, displays data in a default sort order. You'll see how this works as we go through the code.

Next, we have two events for the grid. The first is for when a page is selected on the grid, and is the event procedure defined in the `OnPageIndexChanged` attribute:

```
void DataGrid_Page(object sender, DataGridPageChangedEventArgs e)
{
  DataGrid1.CurrentPageIndex = e.NewPageIndex;
  BindGrid();
}
```

Notice that the second argument to this procedure is of type `DataGridPageChangedEventArgs`. This is automatically sent by ASP.NET and contains two properties, of which we are interested in only one – `NewPageIndex`. This identifies the number of the page selected, so we set the `CurrentPageIndex` property of the grid to the selected page number. We then call the `BindGrid()` routine to re-fetch the data and bind it to the grid. Later, we'll look at why you need to do this.

The second event procedure is for sorting the grid, and is defined in the `OnSortCommand` attribute:

```
void DataGrid_Sort(object sender, DataGridSortCommandEventArgs e)
{
  DataGrid1.CurrentPageIndex = 0;
  SortField = e.SortExpression;
  BindGrid();
}
```

The second argument for this procedure is of type `DataGridSortCommandEventArgs`, which contains the expression on which the grid is being sorted. In this case, this is automatically set by the `DataGrid` as the column headings are sortable, and so contains the column name.

The first line sets the `CurrentPageIndex` of the grid to `0`, having the effect of starting the grid at page 1. We do this because we are re-sorting. We then set `SortField` to the sorted field, and rebind the grid.

Notice that `SortField` hasn't been declared as a variable – in fact it's a property. This might seem confusing because properties are always attached to objects, prompting the question what object is this

one attached to. Well, since it hasn't got a named object, ASP.NET takes this as being a *property* of the current `Page`. By default, a `Page` doesn't have a `SortField` property, so we define one:

```
protected String SortField {

  get {
    object o = ViewState["SortField"];
    return (o == null) ? String.Empty : (String)o;
  }
  set {
    ViewState["SortField"] = value;
  }
}
```

The interesting point is that we haven't defined a class. Because we are coding within an ASP.NET page, the `Page` *is* a class, so all we are doing is adding a property to the page (for the purpose of referencing the sorted field later when we bind the grid). When the page is run, ASP.NET adds your code to the class for the page. It's not like the examples in the previous chapter, where we were creating a separate class – here we want our property to be part of the same class as the rest of the code.

The `get` part of the property first fetches the sort value from the `ViewState` into an object variable (all items in `ViewState` are returned as objects), and then checks to see if the object is `null`. This would be the case if the sort hasn't been defined, such as the first time the page is loaded. If it is `null`, then an empty string is returned, otherwise the object is converted to a string with the `(String)` cast and that is returned. This is a perfectly safe conversion because we know that the `ViewState` for this item only contains a string, as that's what the `set` part of the property does. `ViewState` was covered in *Chapter 6*.

> *Using `String.Empty` is a special way of defining an empty string, and avoids having to use open and close quotation marks next to each other, where it's often difficult to see if there is a space between the quotation marks.*

Now let's look at the `BindGrid()` routine:

```
void BindGrid() {
```

The first two lines define string variables to hold the connection string and the text for the command to run. Notice that the connection string has been changed to an Access one:

```
  string ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
                "Data Source=C:\BegASPNet11\data\Northwind.mdb";
  string CommandText;
```

The use of the `@` symbol before part of the string tells the C# compiler to treat the string exactly as it is typed. We need to do this because in C#, the backward slash character (`\`) is treated as an escape sequence, indicating that the following character is something special. To avoid this we can either use to slash characters together (`\\`), meaning we really want a slash character, or use the `@` symbol. Later in the chapter, you'll see examples of the `\\` style.

Next, we check the `SortField` property to see if we are sorting the data in the order selected by the user (that is, if the user has clicked one of the column headings). This is accessing the `SortField` property of the `Page` and therefore calls the `get` part of the property. If the sort order hasn't been defined, the `String.Empty` is the value of `SortField`. So we set the command string to order by the `CompanyName`. If a sort string has been set, then we use that as the sort order. In either case, we are simply selecting all rows and columns from the `Suppliers` table:

```
    if (SortField == String.Empty)
      CommandText = "select * from Suppliers order by CompanyName";
    else
      CommandText = "select * from Suppliers order by " + SortField;
```

These commands use SQL statements, but we could equally have used stored queries or stored procedures. In practice, you should use stored queries, but using SQL directly here means we don't have to create the stored query – since we're concentrating on ASP.NET we don't want to distract ourselves with the stored procedure. We'll be looking at stored procedures later in the chapter.

Now we come to the part where we connect to the database. Don't worry too much about this code – although we are going to explain it, we're not going to go into too much detail in this section, as we'll be going over the theory later. To define the connection we use an `OleDbConnection` object, and as part of the instantiation we pass in the connection string details. This tells ASP.NET which database to connect to, but doesn't actually open the connection. It defines *where* to connect to when we are ready to connect:

```
    OleDbConnection myConnection = new OleDbConnection(ConnectionString);
```

Now we use an `OleDbDataAdapter` to define the command to run – this will be the `SELECT` query to fetch the data. The data adapter performs two functions. It provides the link between the database and the `DataSet`. It is also how data is fetched from and sent to the database (we'll be looking at the `DataAdapter` in detail in the next chapter). The two arguments we pass in are the command text to run the SQL statement, and the connection object. These define which command to run and which database to run it against:

```
    OleDbDataAdapter myCommand = new OleDbDataAdapter(CommandText,
                                                      myConnection);
```

Note that we still haven't connected to the database and fetched any data, as we've nowhere to store that data. For that we use a `DataSet` object, which you can think of as a mini database (it's not actually a mini database, but that descriptions works well for the moment). It provides a place for the data to be held while we manipulate it:

```
    DataSet ds = new DataSet();
```

Now that we have all of the pieces in place (the connection, the command to run, and a place to put the data), we can go ahead and fetch the data. For that we use the `Fill()` method of the data adapter, passing in the `DataSet`. This opens the database connection, runs the command, places the data into the `DataSet`, and then closes the database connection.

```
    myCommand.Fill(ds);
```

The data is now in our `DataSet` so we can use it as the `DataSource` for the grid, and bind the grid:

```
    DataGrid1.DataSource = ds;
    DataGrid1.DataBind();
  }
```

This may look like a complex set of procedures, but it's actually a simple set of steps that is used many times when you need to fetch data. You'll be seeing this many times during this book, and we'll go over its theory later so you really understand what's happening. For now though, let's look at another way to save time, by using the Web Matrix *Code Wizards*.

Displaying Data Using the Code Wizards

There are times where both the drag and drop from the Data Explorer and the template pages cannot provide you with exactly what you need. Perhaps you'd like to customize the query, or just add a routine to fetch data to an already existing page. The code wizards allow you to add code routines to a page, giving you a finer control of the data being fetched or updated. Let's give this a go.

Try It Out Creating a Data Page

1. Create a new blank ASP.NET page called `CodeWizard.aspx`.
2. Switch to Code view and you'll notice that the Toolbox now shows Code Wizards as shown in Figure 8-7:

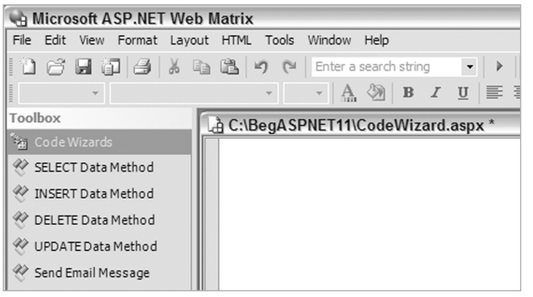


Figure 8-7

3. Pick the SELECT Data Method and drag it from the Toolbox, dropping it into your code window. This starts the wizard, and the first screen as shown in Figure 8-8 is where you pick the database to connect:

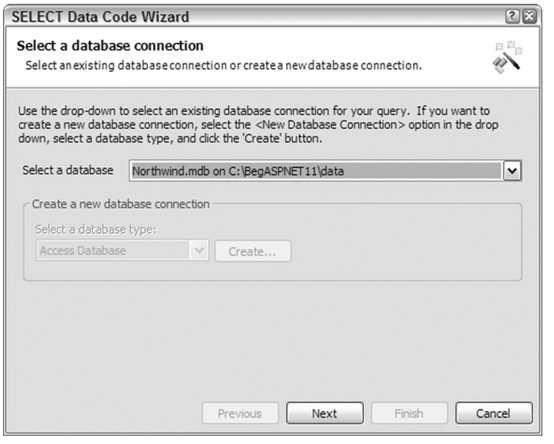


Figure 8-8

4. The drop-down list shows configured data sources (from the Data Explorer) as well as an option to create a new connection. Pick the existing connection and press Next to go to the screen shown in Figure 8-9:

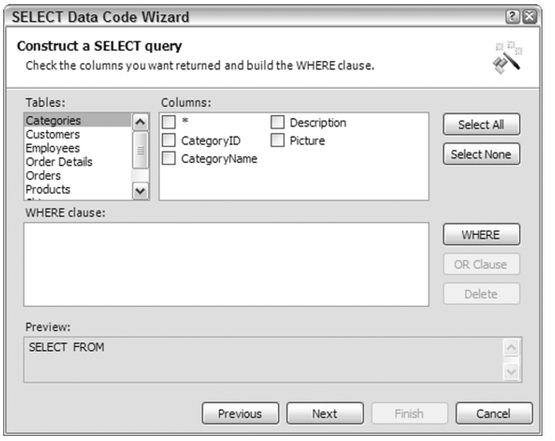


Figure 8-9

Now you can select the columns you wish to show. You can pick multiple columns (the * means all columns from the table) from multiple tables. You simply select them individually. However, when picking columns from multiple tables, you must join the tables. Remember our discussion of linked tables and keys from the beginning of the chapter – you need the primary and foreign key to join the tables.

5. Select the Products table and the ProductName column, and the Categories table and the CategoryName column. Notice the Preview pane at the bottom of the window shows the SQL statement, but without the tables joined together, as shown in Figure 8-10:

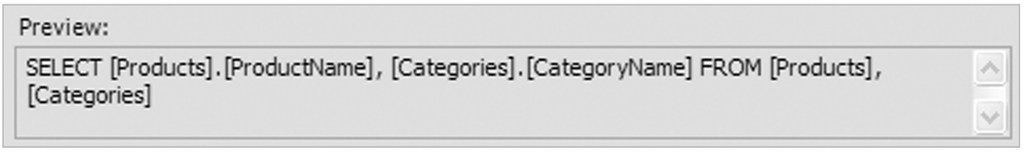


Figure 8-10

6. To join these tables together, we need a `WHERE` clause, so press the WHERE button to open the WHERE Clause Builder window.

7. Select your options the same as shown in Figure 8-11:

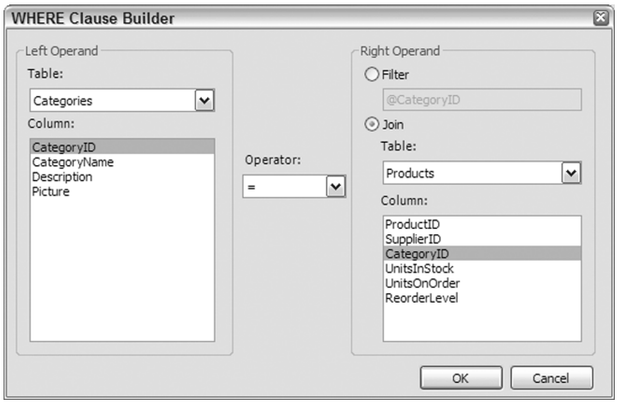


Figure 8-11

8. Click OK and you'll see the WHERE clause part of the window is filled in as shown in Figure 8-12:

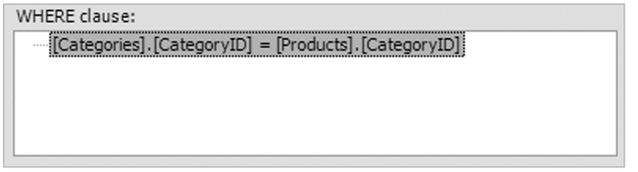


Figure 8-12

9. Press the Next button, and on the Query Preview window press the Test Query button:

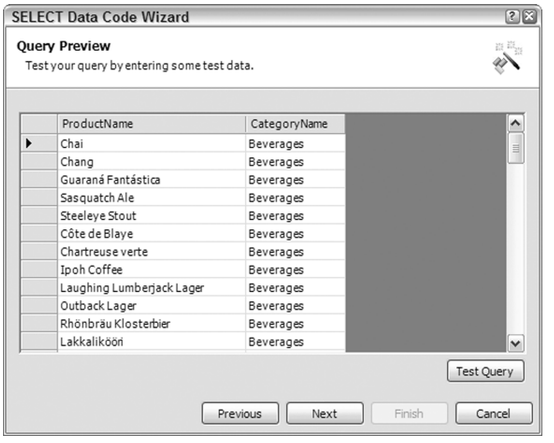


Figure 8-13

You can see just the required columns in Figure 8-13.

10. Press Next.

11. From the Name Method window, change the name textbox to `GetProductsDataSet`. Make sure the radio button at the bottom is set to `DataSet` and press Finish. We'll look at the `DataReader` later in the chapter.

12. Once the code has been added, you want a way to display it. You can do this by switching to Design view and dragging a DataGrid onto the page.

13. Switch to Code view and add the following code, after the `GetProductsDataSet` function:

```
void Page_Load(Object sender, EventArgs e)
{
  DataGrid1.DataSource = GetProductsDataSet();
  DataGrid1.DataBind();
}
```

14. Save the page and run it – you should see Figure 8-14:

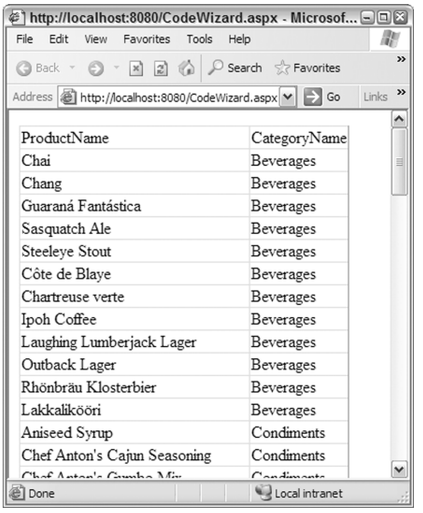


Figure 8-14

You can see how we now only have two columns and from two different tables. Let's see how this works.

How It Works

The key to this is the wizard that allows you to build up an SQL statement. This is great if you are a newcomer to SQL as you don't have to understand how the SQL language works. Perhaps the most important part of this wizard is the WHERE Clause Builder shown in Figure 8-11.

This is where (pun intended) we add the `WHERE` part of the SQL statement, and this is what filters the rows and joins tables together. We've selected the Join option allowing us to specify the primary key (`CategoryID` in the `Categories` table) and the foreign key (`CategoryID` in the `Products` table). The `WHERE` clause becomes:

```
WHERE [Categories].[CategoryID] = [Products].[CategoryID]
```

If we wanted to add a third table, perhaps `Suppliers`, we could use an `AND` clause. Once you've declared one `WHERE` clause, the WHERE button has a different name – AND Clause as shown in Figure 8-15:

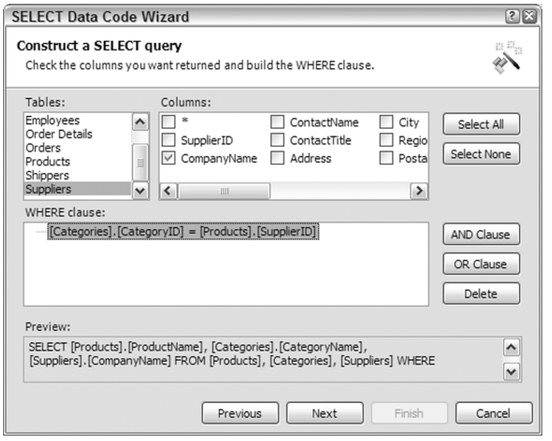


Figure 8-15

Pressing the AND Clause button shows the same WHERE Clause Builder, but this time you'd set the link between the `Suppliers` and `Products` tables as shown in Figure 8-16:

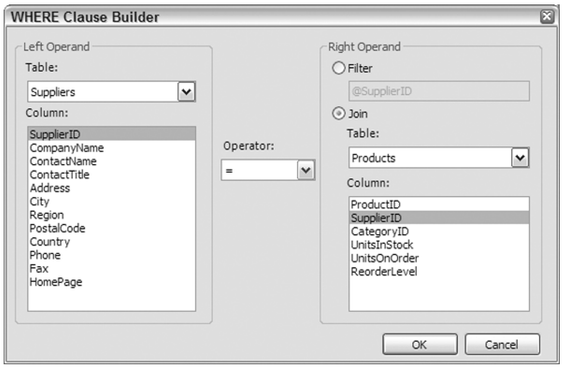


Figure 8-16

Now when you look at the WHERE clause section you see two tables joined together as in Figure 8-17:

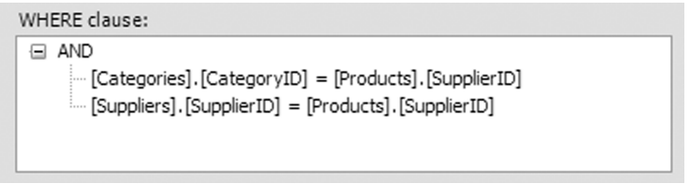


Figure 8-17

The WHERE Clause Builder can also be used to filter data so that only *selected* rows are shown; we'll look at that later. For now though, let's look at the code the wizard created for us (it may look slightly different in your page – we've wrapped it so it's easier to read):

```
System.Data.DataSet GetProductsDataSet() {
  string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
           "Ole DB Services=-4; Data Source=C:\\BegASPNET11\\" +
           "data\\Northwind.mdb";
  System.Data.IDbConnection dbConnection =
           new System.Data.OleDb.OleDbConnection(connectionString);

  string queryString = "SELECT [Products].[ProductName], " +
           "[Categories].[CategoryName] FROM [Products], [Categories] " +
           "WHERE ([Categories].[CategoryID] = [Products].[CategoryID])";
  System.Data.IDbCommand dbCommand = new System.Data.OleDb.OleDbCommand();
  dbCommand.CommandText = queryString;
  dbCommand.Connection = dbConnection;

  System.Data.IDbDataAdapter dataAdapter =
           new System.Data.OleDb.OleDbDataAdapter();
  dataAdapter.SelectCommand = dbCommand;
  System.Data.DataSet dataSet = new System.Data.DataSet();
  dataAdapter.Fill(dataSet);

  return dataSet;
}
```

Let's tackle this systematically. First, we have the function declaration:

```
System.Data.DataSet GetProductsDataSet() {
```

This is defined as type `System.Data.DataSet`, which means it's going to return a `DataSet` (we'll look at this in detail in the next chapter). You'll notice that the declaration has the `System.Data` namespace before it. This is done because, while declaring variables or functions, ASP.NET needs to know where the type is stored.

Normally we use the `<%@ import Namespace="..." %>` page directive to indicate the namespaces being used in a page, and thus we don't have to specify the namespace when declaring variables. The wizard isn't sure what namespaces have been set at the top of the page, so it includes the full namespace just-in-case, ensuring that the code will compile under all situations.

Next, we have the connection string that simply points to our existing database:

```
string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
        "Ole DB Services=-4; Data Source=C:\\BegASPNET11\\" +
        "data\\Northwind.mdb";
```

Notice that this uses two backward slash characters to avoid the problem of the single slash character being an escape sequence. In our earlier example we used the `@` symbol.

Now we have the connection object:

```
System.Data.IDbConnection dbConnection =
        new System.Data.OleDb.OleDbConnection(connectionString);
```

One thing that's immediately obvious is the fact that this example doesn't use the `OleDbConnection` type to define the connection to the database; it uses `IDbConnection`. If this seems confusing, refer to the discussion of interfaces in the previous chapter, where we talked about generic routines.

`IDbConnection` is an interface that defines what the `Connection` class must do, and since the wizard is building a generic routine, it uses this interface. This is because the wizard allows you to connect to different database types. This is seen on the first screen and is the same as the Data Explorer allowing you to pick either Access or SQL Server database. To make the wizard simpler, it uses the generic interface as the type rather than having to use the type for a specific database.

> *The Interface simply enforces the correct signature on a class implementing the interface. There's no actual requirement for the implementation to do anything. You could have a class that implements the `Open()` method but that actually does something else instead of opening a connection. It would be dumb, but it could be done.*

Next we have the SQL string, as built up by the wizard:

```
string queryString = "SELECT [Products].[ProductName], " +
        "[Categories].[CategoryName] FROM [Products], [Categories] " +
        "WHERE ([Categories].[CategoryID] = [Products].[CategoryID])";
```

Now we have the definition of the command object. In previous examples, we passed the command text directly into the `OleDbDataAdapter`. Underneath, ASP.NET actually creates another object – a `Command` object. However, you don't see that `Command` object, as it is used internally. The wizard creates the `Command` object directly, by making use of the `CommandText` property to store the SQL command, and the `Connection` property to store the database connection. As with the connection that used the interface as its type, the command is also defined as an interface type (`IDbCommand`).

```
System.Data.IDbCommand dbCommand = new System.Data.OleDb.OleDbCommand();
dbCommand.CommandText = queryString;
dbCommand.Connection = dbConnection;
```

Now we have the definition of the data adapter, and as with the connection, the type of the variable is the interface type:

```
System.Data.IDbDataAdapter dataAdapter =
        new System.Data.OleDb.OleDbDataAdapter();
```

We mentioned earlier that the data adapter is the link between our page and the data. As part of this link, the adapter provides not only data fetching, but also data modification. It does so with different command objects, exposed as properties of the adapter. These allow the different commands to run depending upon the action being performed. In this example, we are fetching data so we use the `SelectCommand` property (so named because we are selecting rows to view).

```
dataAdapter.SelectCommand = dbCommand;
```

If you use the data adapter directly, without explicitly creating a `Command`, this is what it does behind the scenes.

To fetch the data, we then create a `DataSet` and use the `Fill()` method of the adapter:

```
System.Data.DataSet dataSet = new System.Data.DataSet();
dataAdapter.Fill(dataSet);
```

And finally, we return the data:

```
    return dataSet;
}
```

This code is more complex than the previous example, but it follows a similar path. It creates a connection, creates a command, creates a data adapter, and then a `DataSet`. A look at these objects and their relationships in more detail will give you a clearer picture of how they work together.

To find out more about the `DataAdapter`'s properties and methods consult the .NET Documentation. The `OleDbDataAdapter` is in the `System.Web.OleDb` namespace and the `SqlDataAdapter` is in the `System.Web.SqlClient` namespace.

ADO.NET

All of the data access we've seen so far is based upon ADO.NET – the common name for all of the data access classes. We'll only be looking at a few of these, and the ones you'll use most are:

- `Connection`: Provides details of connecting to the database
- `Command`: Provides details of the command to be run
- `DataAdapter`: Manages the command, and fetchs and updates data
- `DataSet`: Provides a store for data
- `DataReader`: Provides quick read-only access to data

ADO.NET is designed to talk to multiple databases, so there are different objects for different database types. To keep the separation, ADO.NET classes are contained within different namespaces:

- `System.Data`: Contains the base data objects (such as `DataSet`) common to all databases.

- `System.Data.OleDb`: Contains the objects used to communicate to databases via OLEDB. OLEDB provides a common set of features to connect to multiple databases, such as Access, DBase, and so on.
- `System.Data.SqlClient`: Provides the objects used to communicate with SQL Server.

For some of the objects there are two copies – one in the `OleDb` namespace, and one in the `SqlClient` namespace. For example, there are two `Connection` objects – `OleDbConnection` and `SqlConnection`. Having two objects means they can be optimized for particular databases. Look at Figure 8-18 to see how they relate to each other:

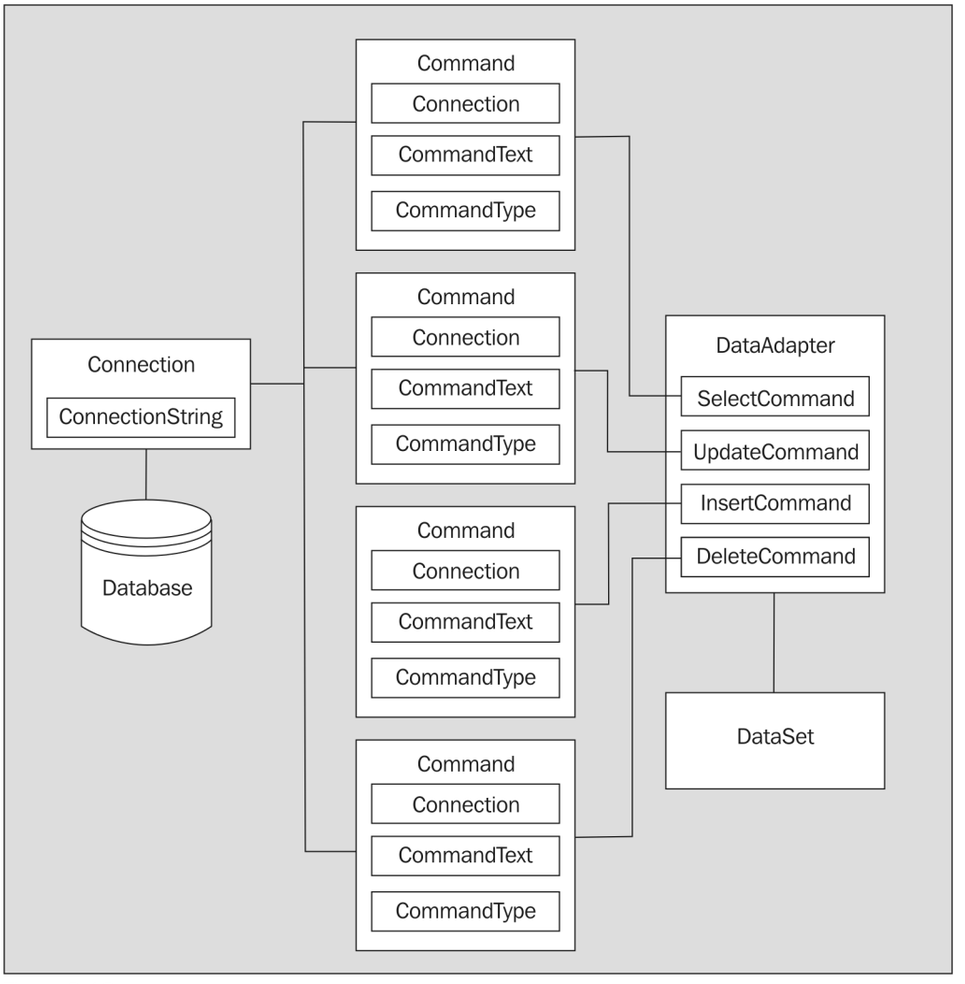


Figure 8-18

On the left we have the database and the connection, in the middle we have four `Command` objects, and on the right a `DataAdapter` and a `DataSet`. Notice that the `DataAdapter` contains four `Command` objects:

- `SelectCommand`: Fetches data
- `UpdateCommand`: Updates data
- `InsertCommand`: Inserts new data
- `DeleteCommand`: Deletes data

Each of these `Command` objects has a `Connection` property to specify which database the command applies to, a `CommandText` property to specify the command text to run, and a `CommandType` property to indicate the type of command (straight SQL or a stored procedure).

As we said earlier, if you don't explicitly create `Command` objects and use the `DataAdapter` directly, a `Command` is created for you using the details passed into the constructor of the `DataAdapter`, and this `Command` is used as the `SelectCommand`.

We'll be looking at the `UpdateCommand`, `InsertCommand`, and `DeleteCommand` in the next chapter.

Let's look at these objects in a bit more detail, concentrating on the `OleDb` ones as we're using Access. If you want to use SQL Server, you can simply replace `OleDb` with `SqlClient` in the object names; just change the connection string, and continue working.

The OleDbConnection Object

As we've said earlier, the `Connection` object provides us with the means to communicate to a database. Probably the only property you'll use is the `ConnectionString` property that can either be set as the object is instantiated:

```
string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
          "Data Source=C:\\BegASPNET11\\data\\Northwind.mdb";
OleDbConnection conn = new OleDbConnection(connectionString);
```

or it can be set with the property:

```
string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
          "Data Source=C:\\BegASPNET11\\data\\Northwind.mdb";
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = connectionString;
```

The two main methods you'll use are `Open()` and `Close()`, which (unsurprisingly) open and close the connection to the database. When used as we have so far, there is no need to do this explicitly since the `Fill()` method of a `DataAdapter` does it for you.

The OleDbCommand Object

The `OleDbCommand` has several properties that we'll be looking at:

Property	Description	
`CommandText`	Contains the SQL command or the name of a stored procedure.	
`CommandType`	Indicates the type of command being run, and can be one of the `CommandType` enumeration values, which are:	
	`StoredProcedure`	To indicate a stored procedure is being run.
	`TableDirect`	To indicate the entire contents of a table are being returned. In this case, the `CommandText` property should contain the table name. This value only works for `Oledb` connections.
	`Text`	To indicate a SQL text command. This is the default value.
`Connection`	The `Connection` object being used to connect to a database.	
`Parameters`	A collection or `Parameter` objects, which are used to pass details to and from the command.	

The three main methods of the command you'll use are the execute methods:

Method	Description
`ExecuteNonQuery()`	This executes the command but doesn't return any data. It is useful for commands that perform an action, such as updating data, but don't need to return a value.
`ExecuteReader()`	This executes the command and returns a `DataReader` object.
`ExecuteScalar()`	This executes the command and returns a single value.

In the examples so far, we haven't used these methods as the execution of the command is handled transparently for us. You'll see the `ExecuteReader()` method in action when you look at the `DataReader`, and the `ExecuteNonQuery()` method in action in the next chapter.

The Parameters Collection

A parameter is an unknown value – a value that ADO.NET doesn't know until the page is being run, and is often used to filter data based upon some user value. For example, consider a page showing a list

of products, with a drop-down list showing the product categories. The user could select a category so that only those categories are shown.

The `Parameters` collection contains a `Parameter` object for each parameter in the query. Thus, a command with three parameters would have objects looking like in Figure 8-19:

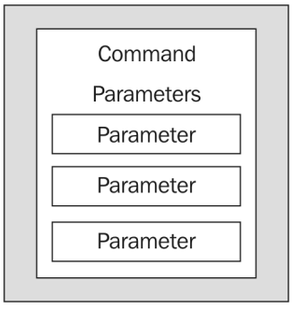


Figure 8-19

Let's look at an example to see how this works.

Try It Out Using Parameters

1. Create a new blank ASP.NET page called `Parameters.aspx`.
2. Add a Label and change the `Text` property to `Category:`.
3. Add a DropDownList next to the label and change the `ID` property to `lstCategory`.
4. Add a Button next to the DropDownList and change the `Text` property to `Fetch`.
5. Add a DataGrid underneath the other controls. Your page should now look like Figure 8-20:

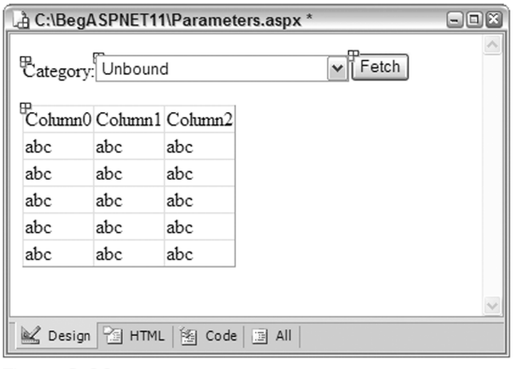


Figure 8-20

6. Double-click the Fetch button to switch to the `Click` event procedure. Add the following code:

```
void Button1_Click(object sender, EventArgs e) {
  DataGrid1.DataSource =
GetProducts(Convert.ToInt32(lstCategory.SelectedValue));
  DataGrid1.DataBind();
}
```

7. Underneath that procedure, add the following code:

```
void Page_Load(Object Sender, EventArgs e)
{
  if (!Page.IsPostBack) {
    lstCategory.DataSource = GetCategories();
    lstCategory.DataValueField = "CategoryID";
    lstCategory.DataTextField = "CategoryName";
    lstCategory.DataBind();
  }
}
```

8. Underneath the preceding block of code, drag a SELECT Data Method wizard from the toolbox onto the page. Pick the current database connection and select the `CategoryID` and `CategoryName` columns from the `Categories` table. Call the procedure that you have created `GetCategories` and have it return a `DataSet`.

9. Drag another SELECT Data Method wizard onto the page, underneath the SELECT Data Method wizard that you just created. Pick the current database connection, and select `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `UnitsInStock` from the `Products` table.

10. Click the WHERE button and pick the `CategoryID` from the `Products` table making it Filter on `@CategoryID`, as shown in Figure 8-21:

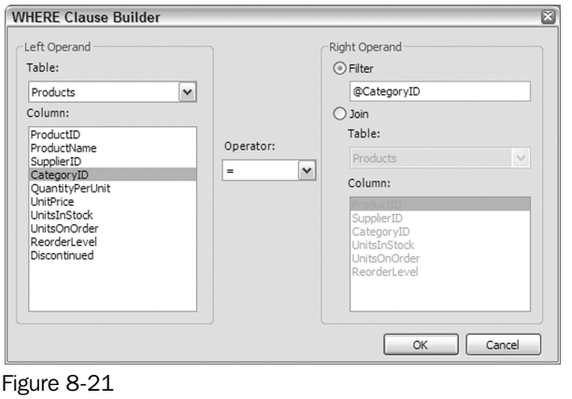


Figure 8-21

11. Click OK and Next to get to the Name Method screen.

12. Call the procedure `GetProducts` and have it return a `DataSet`. Press Finish to insert the code.

13. Save the file and run it.

14. Select a category and then click Fetch to see only the products for that category shown in Figure 8-22:

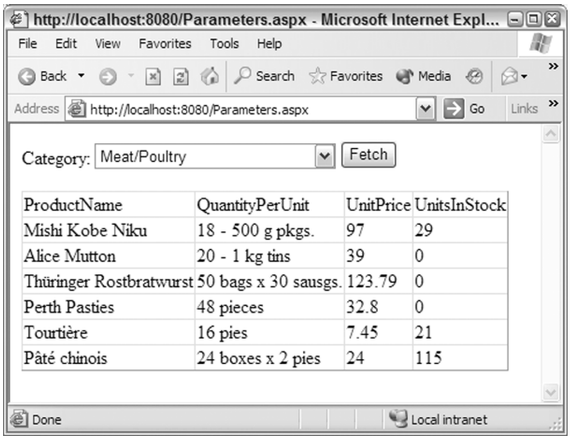


Figure 8-22

What you've achieved here is two things. First, you've used two controls that are bound to data – the list of categories and the grid of products. Second, you only fetched the products for a selected category – you've filtered the list. Let's see how this works.

How It Works

Let's start the code examination with the `Page_Load()` event, where we fill the `Categories` list:

```
void Page_Load(Object Sender, EventArgs e) {
```

We only want to fetch the data and bind it to the list the first time the page is loaded, so we use the `IsPostBack` property of the page to check if this is a postback. If it isn't, it must be the first load, so we fetch the data. We don't need to do this on subsequent page requests as the list itself stores the data.

```
  if (!Page.IsPostBack) {
    lstCategory.DataSource = GetCategories();
```

Instead of calling the `DataBind` straight away, we want to tell the list which columns from the data to use. A `DropDownList` stores two pieces of information – one is shown on the page (the text field), and the other is hidden (the value field). The text field is used for what the user needs to see, while the value field often contains an ID – what the user doesn't need to see. The `DropDownList` doesn't automatically know which columns contain these pieces of information, thus we use the `DataValueField` and `DataTextField` properties. The `DataValueField` is the `CategoryID`, the unique key for the category, and this will be used later in our code:

```
        lstCategory.DataValueField = "CategoryID";
        lstCategory.DataTextField = "CategoryName";
        lstCategory.DataBind();
      }
    }
```

When the Fetch button is clicked, we need to get the value from the `DropDownList`. For this, we use the `SelectedValue` property, which is new to ASP.NET 1.1. This contains the ID of the selected category, and we pass this into the `GetProducts` routine, which will return a `DataSet` of the products.

However, we can't pass this value directly into `GetProducts` as an integer value is expected, and the `SelectedValue` returns a string. So we have to convert it first using the `ToInt32` method of the `Convert` class. This is a static class method (which doesn't require an instance of the `Convert` class) and simply takes a single string argument. The return value from `ToInt32` is an integer representation of the string passed in.

The `DataSet` returned from `GetProducts` is set to the `DataSource` of the grid and the `DataBind` method is called to bind the data:

```
void Button1_Click(object sender, EventArgs e) {
  DataGrid1.DataSource =
                    GetProducts(Convert.ToInt32(lstCategory.SelectedValue));
  DataGrid1.DataBind();
}
```

There are two routines to fetch data, but one of them is the same as we've already seen – using a simple `DataSet` to fetch data (in this case the `Categories`). What we want to see is the `GetProducts` routine, which gets *filtered* data. The first thing to notice is that it accepts an `int` argument – this will contain the `CategoryID`, passed in from the button click event:

```
System.Data.DataSet GetProducts(int categoryID) {
```

Next, we define the connection details, as we've seen in previous examples:

```
  string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
        "Ole DB Services=-4; Data Source=C:\\BegASPNET11\\" +
        "data\\Northwind.mdb";
  System.Data.IDbConnection dbConnection =
        new System.Data.OleDb.OleDbConnection(connectionString);
```

Then we define the query:

```
  string queryString = "SELECT [Products].[ProductName], "
                     "[Products].[QuantityPerUnit], [Products].[UnitPrice],"+
                     "[Products].[UnitsInStock] FROM [Products] " +
                     "WHERE ([Products].[CategoryID] = @CategoryID)";
```

Note that the `WHERE` clause is filtering on `CategoryID`. However, the value used for the filter (`@CategoryID`) is not a real value but a *placeholder*. This tells ADO.NET that the value will be supplied by a parameter.

Once the query string is set, we define our command to run the query, as follows:

```
System.Data.IDbCommand dbCommand = new System.Data.OleDb.OleDbCommand();
dbCommand.CommandText = queryString;
dbCommand.Connection = dbConnection;
```

Now we come to the definition of the parameter. Like many of the other examples, this uses a database specific object – an `OleDbParameter`, which defines what is being passed into the query:

```
System.Data.IDataParameter dbParam_categoryID =
     new System.Data.OleDb.OleDbParameter();
```

We then set the properties of the parameter. The `ParameterName` indicates the name of the parameter, and we set the value to be the same as the placeholder. The `Value` property stores the value for the parameter, and is set to the `CategoryID` passed into the procedure from the button click event – it's the ID of the category selected from the list. The `DbType` property indicates the database type – `Int32` is the database equivalent of an `Integer`:

```
dbParam_categoryID.ParameterName = "@CategoryID";
dbParam_categoryID.Value = categoryID;
dbParam_categoryID.DbType = System.Data.DbType.Int32;
```

At this point, even though we have a `Parameter` object, it's not associated with the command, so we add it to the `Parameters` collection of the command:

```
dbCommand.Parameters.Add(dbParam_categoryID);
```

When ADO.NET processes the command, it matches parameters in the collection with the placeholders in the query and substitutes the placeholder with the value in the parameter.

The rest of the code is as we've seen it before. We create a `DataAdapter` to run the command, and use the `Fill()` method to fetch the data into our `DataSet`:

```
System.Data.IDbDataAdapter dataAdapter =
            new System.Data.OleDb.OleDbDataAdapter();
dataAdapter.SelectCommand = dbCommand;
System.Data.DataSet dataSet = new System.Data.DataSet();
dataAdapter.Fill(dataSet);

return dataSet;
}
```

As you can see, there really isn't that much code; even though we've introduced a new object, much of the code remains the same.

Filtering Queries

There's a very important point to know about filtering data, as you may see code elsewhere that uses a bad method of doing it – it simply builds up the SQL string (as we've done), but instead of using parameters, it just appends the filter value to the SQL string. For example, you might see this:

```
string queryString = "SELECT [Products].[ProductName], "
                     "[Products].[QuantityPerUnit], [Products].[UnitPrice]," +
                     "[Products].[UnitsInStock] FROM [Products] " +
                     "WHERE ([Products].[CategoryID] = " + CategoryID + ")";
```

This simply appends the `CategoryID` value (from the function argument) into the SQL string. Why is this bad when it achieves the same objectives while using lesser code? The answer has to do with hacking. This type of method potentially allows what are known as *SQL Injection Attacks*, which are 'very bad things' (do a Web search for more details on SQL Injection). If you have a scale for 'bad things to do', then this is right up there, at the top!

Using Parameters protects you from this. Although it has the same effect, the processing ADO.NET does secure you against this type of attack.

Although using Parameters involves a little more work, it's much safer and should always be used.

The OleDataAdapter Object

The `OleDbDataAdapter` contains the commands used to manipulate data. The four `Command` objects it contains are held as properties; `SelectCommand`, `UpdateCommand`, `InsertCommand`, and `DeleteCommand`. The `SelectCommand` is automatically run when the `Fill()` method is called. The other three commands are run when the `Update` method is called – we'll be looking at this in the next chapter.

The DataSet Object

While the other objects we've looked at have different classes for different databases, the `DataSet` is common to all databases, and is therefore in the `System.Data` namespace. It doesn't actually communicate with the database – the `DataAdapter` handles all communication.

The `DataSet` has many properties and methods; we'll look at them in the next chapter. Since this chapter is concentrating on displaying data, all you need to remember is that when we fetch data it is stored in the `DataSet`, and then we bind controls to that data.

The DataReader Object

The `DataReader`, an object that we haven't come across yet, is optimised for reading data. When dealing with databases, connecting to them and fetching the data can often be the longest part of a page, therefore we want to do it as quickly as possible. We also want to ensure that the database server isn't tied up – we want not only to get the data quickly, but also stay connected to the database for as little time as possible.

For this reason we aim to open the connection to the database as late as possible, get the data, and close the connection as soon as possible. This frees up database resources, allowing the database to process other requests. This is the technique that the `DataAdapter` uses when filling a `DataSet`. If you manually open a connection, it isn't automatically closed.

Many times, when fetching data we simply want to display it as it is, perhaps by binding it to a grid. The `DataSet` provides a local store of the data, which is often more than we need, so we can use an `OleDbDataReader` to stream the data directly from the database into the grid. Let's give this a go.

Try It Out Using a DataReader

1. Create a new blank ASP.NET page called `DataReader.aspx`.
2. Drag a DataGrid control from the Toolbox onto the page.
3. Switch to Code view and start the code wizard by dragging the SELECT Data Method onto the code page.
4. Select the existing database connection from the first screen and press Next.
5. Select the Products table, and from the Columns select ProductName, QuantityPerUnit, UnitPrice, and UnitsInStock.
6. Click Next, and Next again, to go past the Query Preview screen.
7. Enter `GetProductsReader` as the method name, and select the DataReader option on the Name Method screen.
8. Press Finish to insert the code into your page.
9. Underneath the newly inserted method, add the following:

```
void Page_Load(Object sender, EventArgs e) {
  DataGrid1.DataSource = GetProductsReader();
  DataGrid1.DataBind();
}
```

10. Save the page and run it.

You'll see a grid containing just the selected columns. This doesn't look very different from the other examples, but it's how the data is fetched that's important. Let's look at this.

How It Works

Let's start by looking at the code that the wizard generated for us. The declaration of the function returns an `IDataReader` – the interface that data readers implement:

```
System.Data.IDataReader GetProductsReader() {
```

Next we have the connection details – these are the same as you've previously seen (although they might look different in your code file, as this has been formatted to fit on the page):

```
string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; " +
            "Ole DB Services=-4; Data Source=C:\\BegASPNET11\\" +
            "data\\Northwind.mdb";
  System.Data.IDbConnection dbConnection =
         new System.Data.OleDb.OleDbConnection(connectionString);
```

Next, we have the query string and the command details:

```
  string queryString = "SELECT [Products].[ProductName], " +
       "[Products].[QuantityPerUnit], [Products].[UnitPrice], " +
```

```
        "[Products].[UnitsInStock] FROM [Products]";
    System.Data.IDbCommand dbCommand = new System.Data.OleDb.OleDbCommand();
    dbCommand.CommandText = queryString;
    dbCommand.Connection = dbConnection;
```

Once the command details are set, we can open the database connection:

```
    dbConnection.Open();
```

Even though the database connection has been opened for us when using a `DataSet`, we still have to open it manually because we are using an `OleDbCommand` and a data reader.

Next, we declare the data reader. It is of type `IDataReader` and the object is created by the return value of the `ExecuteReader()` method of the command:

```
    System.Data.IDataReader dataReader =
          dbCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

Remember that the command has the SQL statement, so `ExecuteReader()` tells ADO.NET to run the command and return a data reader. The argument indicates that as soon as the data is finished with the connection to the database, the connection should be closed. When using `ExecuteReader()`, you should always add this argument to make sure the connection is closed as soon as it is no longer required.

Finally, we return the reader object:

```
    return dataReader;
}
```

To bind to the grid, we simply use this function as the `DataSource` for the grid. Since the function returns a stream of data, the grid just binds to that data:

```
void Page_Load(Object sender, EventArgs e) {
  DataGrid1.DataSource = GetProductsReader();
  DataGrid1.DataBind();
}
```

DataReader Methods and Properties

The `DataReader` exists as `SqlDataReader` (for SQL Server) and `OleDbDataReader` (for other databases), as well as a common `IDataReader` interface. If you are not using generic code, you can create the reader as follows:

```
System.Data.OleDbDataReader dataReader = new _
dbCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

Using data readers is the most efficient way of fetching data from a database, but you don't have to bind to a grid. You can use the properties and methods to fetch the data directly. If you do this, it's best to use the `OleDbDataReader` rather than the interface, as the `OleDbDataReader` contains more properties that make it easier to use. For example, consider the following code:

```
System.Data.OleDbDataReader dataReader = new _
dbCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);

if (!dataReader.HasRows)
  Response.Write("No rows found");
else
  while (dataReader.Read())
    Response.Write(dataReader("ProductName") + "<br />";

dataReader.Close();
```

This first uses the `HasRows` property to determine if there are any rows, and then uses the `Read` method to read a row. This is done within a loop, with `Read` returning the `true` if there is a current row and moving onto the next, and `false` if there are no rows.

Summary

The results of the examples in this chapter have been relatively simple, but you've actually learned a lot. The first three main topics looked at how to use the Web Matrix to reduce your development time, taking away much of the legwork you'd normally have to do. We looked at the using the Data Explorer to drag and drop tables directly onto page, using the Web Matrix template pages, and using the code wizards.

After looking at the quick way of getting data, you saw the theory behind it, examining the objects. Even though we continued to use the wizards to generate code, you were now able to see how this wizard code worked (just because we understand how it works doesn't mean we abandon anything that makes our job easier).

Now it's time to look at taking your data usage one step further by showing how to update data, and how to manage your data-handling routines.

Exercises

1. In this chapter we created a page that showed only the products for a selected category. Try and think of ways to enhance this to show products for either a selected category or all categories.
2. In Exercise 1, we wanted to bind data from a database to a `DropDownList` as well as manually add an entry. There are two ways to solve this issue – using techniques shown in this chapter, and using techniques not yet covered. Try and code the solution using the known technique, but see if you can think of a way to solve it using a new technique.