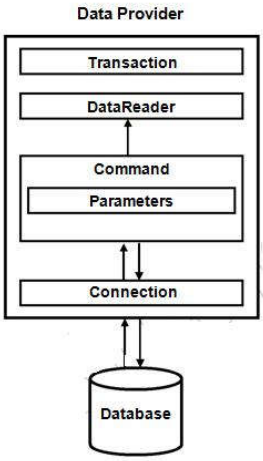
ADO.NET Basics

- ADO.NET provides its functionality in two groups of classes
    - Those that are used to contain and manage data (such as DataSet, DataTable, DataRow, and DataRelation)
    - Those that are used to connect to a specific data source (such as Connection, Command, and DataReader).

Connected Architecture of ADO.NET

The architecture of ADO.net, in which connection must be opened to access the data retrieved from database is called as connected architecture. Connected architecture was built on the classes: connection, command, datareader and transaction.



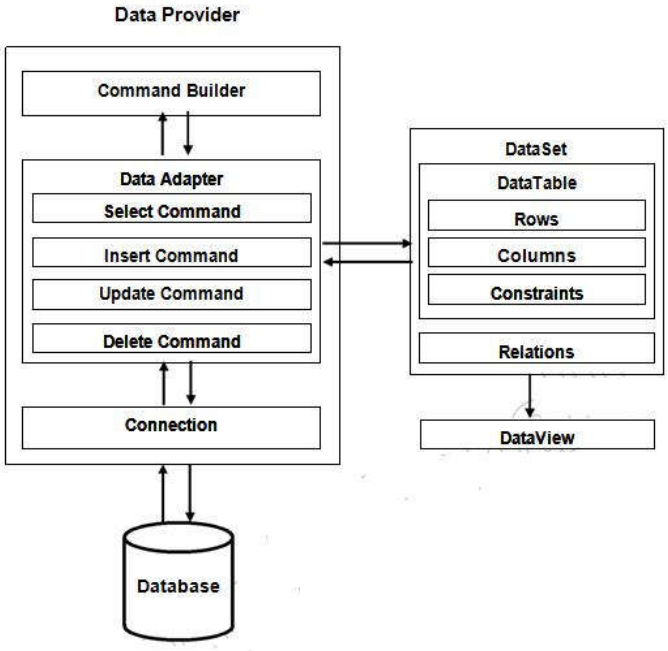
Connection : in connected architecture also the purpose of connection is to just establish a connection to database and itself will not transfer any data.

DataReader : DataReader is used to store the data retrieved by command object and make it available for .net application. Data in DataReader is read only and within the DataReader you can navigate only in forward direction and it also only one record at a time.

To access one by one record from the DataReader, call **Read()** method of the DataReader whose return type is **bool.** When the next record was successfully read, the Read() method will return true and otherwise returns false.

Disconnected Architecture in ADO.NET

The architecture of ADO.net in which data retrieved from database can be accessed even when connection to database was closed is called as disconnected architecture. Disconnected architecture of ADO.net was built on classes `Connection`, `DataAdapter`, `CommandBuilder` and `DataSet` and `DataView`.



Connection: `Connection` object is used to establish a connection to database and connection itself will not transfer any data.

DataAdapter: `DataAdapter` is used to transfer the data between database and dataset. It has commands like select, insert, update and delete. Select command is used to retrieve data from database and insert, update and delete commands are used to send changes to the data in dataset to database. It needs a connection to transfer the data.

CommandBuilder: by default DataAdapter contains only the select command and it doesn't contain insert, update and delete commands. To create insert, update and delete commands for the `DataAdapter`, `CommandBuilder` is used. It is used only to create these commands for the `DataAdapter` and has no other purpose.

DataSet: `DataSet` is used to store the data retrieved from database by `DataAdapter` and make it available for .net application.

To fill data in to dataset `fill()` method of dataadapter is used and has the following syntax.

```
Da.Fill(Ds,"TableName");
```

When fill method was called, `DataAdapter` will open a connection to database, executes select command, stores the data retrieved by select command in to `DataSet` and immediately closes the connection.

As connection to database was closed, any changes to the data in dataset will not be directly sent to the database and will be made only in the dataset. To send changes made to data in dataset to the database, `Update()` method of the dataadapter is used that has the following syntax.

```
Da.Update(Ds,"Tablename");
```

When Update method was called, dataadapter will again open the connection to database, executes insert, update and delete commands to send changes in dataset to database and immediately closes the connection. As connection is opened only when it is required and will be

automatically closed when it was not required, this architecture is called disconnected architecture.

A dataset can contain data in multiple tables.

Data Providers:

The second group of classes exists in several different 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 7 or later. Internally, it uses SQL Server's TDS (tabular data stream) protocol for communicating, thus guaranteeing the best possible performance. If you're using Oracle, you'll need to use the Oracle provider classes instead.

It's important to understand that you can use any data provider in almost the same way, with almost the same code. The provider classes derive from the same base classes, implement the same interfaces, and expose the same set of methods and properties.

In some cases, a data provider object will provide custom functionality that's available only with certain data sources, such as SQL Server's ability to perform XML queries. However, the basic members used for retrieving and modifying data are identical.

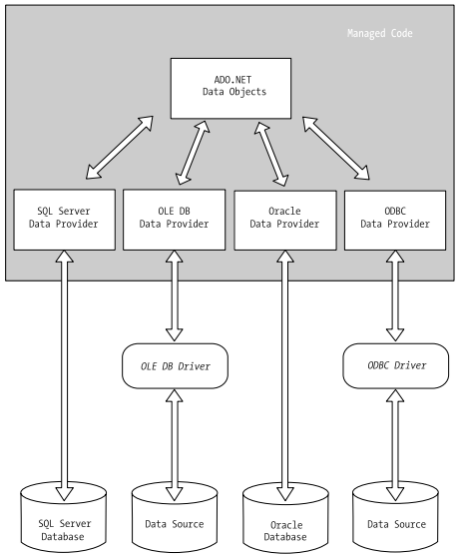
.NET includes the following four providers:

- *SQL Server provider*: Provides optimized access to a SQL Server database (version 7.0 or later)
- *OLE DB provider*: Provides access to any data source that has an OLE DB driver
- *Oracle provider*: Provides optimized access to an Oracle database (version 8i or later)
- *ODBC provider*: Provides access to any data source that has an ODBC (Open Database Connectivity) driver

In addition, third-party developers and database vendors have released their own ADO.NET providers, which follow the same conventions and can be used in the same way as those that are included with the .NET Framework.

If you can't find a suitable provider, you can use the OLE DB provider, as long as you have an OLE DB driver for your data source. The OLE DB technology has been around for any years as part of ADO, so most data sources provide an OLE DB driver including SQL Server, Oracle, Access, MySQL, etc. In the rare situation that you can't find a full provider or an OLE DB driver, you can fall back on the ODBC provider, which works in conjunction with an ODBC driver.

To help understand the different layers that come into play with ADO.NET, refer to Figure



The layers between code and the data source Data Namespaces

ADO.NET Namespaces	Namespace Purpose
System.Data	Contains fundamental classes with the core ADO.NET function-ality. This includes ataSet 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.
System.Data.Common	Not used directly in your code. These classes are used by other data provider classes that inherit from them and provide versions customized for a specific data source.
System.Data.OleDb	Contains the classes you use to connect to an OLE DB data source and execute commands, including OleDbConnection and OleDbCommand.
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database (version 7.0 or later) and execute commands. These classes, such as SqlCommand and SqlConnection, provide all the same properties and methods as their counterparts in the System.Data.OleDb namespace. The only difference is that they are optimized for SQL Server and provide better performance by eliminating the extra OLE DB layer (and by connecting directly to the optimized TDS interface).
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.OracleClient	Contains the classes you use to connect to an Oracle database and execute commands, such as OracleConnection and

	OracleCommand.
System.Data.Odbc	Contains the classes you use to connect to a data source through an ODBC driver and execute commands. These classes include OdbcConnection and OdbcComman

The Data Provider Classes

On their own, the data classes can't accomplish much. Technically, you could create data objects by hand, build tables and rows in your code, and fill them with information. But in most cases, the information you need is located in a data source such as a relational database. To access this information, extract it, and insert it into the appropriate data objects, you need the data provider classes described in this section. Remember, each one of these classes has a database-specific implementation. That means you use a different, but essentially equivalent, object depending on whether you're interacting with SQL Server, Oracle, or any other ADO.NET provider.

Regardless of which provider you use, your code will look almost the same. Often the only differences will be the namespace that's used and the name of the ADO.NET data access classes.

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.

	SQL Server Data Provider	**OLEDB Data Provider**	**Oracle Data Provider**	**Odbc Data Provider**
Connection	SqlConnection	OleDbConnection	OracleConnection	OdbcConnection
Command	SqlCommand	OleDbCommand	OracleCommand	OdbcCommand
DataReader	SqlDataReader	OleDbDataReader	OracleDataReader	OdbcDataReader
DataAdapter	SqlDataAdapter	OleDbDataAdapter	OracleDataAdapter	OdbcDataAdapter

Remember, though the underlying technical details differ, the classes are almost identical. The only real differences are as follows:

- The names of the Connection, Command, DataReader, and DataAdapter classes are different in order to help you distinguish them.
- The connection string (the information you use to connect to the database) differs depending on what data source you're using, where it's located, and what type of security you're using.
- Occasionally, a provider may choose to add features, such as methods for specific features or classes to represent specific data types. For example, the SQL Server Command class includes a method for executing XML queries that aren't part of the SQL standard.

Connected Data Access

The easiest way to interact with a database is to use direct data access. When you use direct data access, you're in charge of building a SQL command and executing it. You use commands to query, insert, update, and delete information.

When you query data with direct data access, 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 than 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.

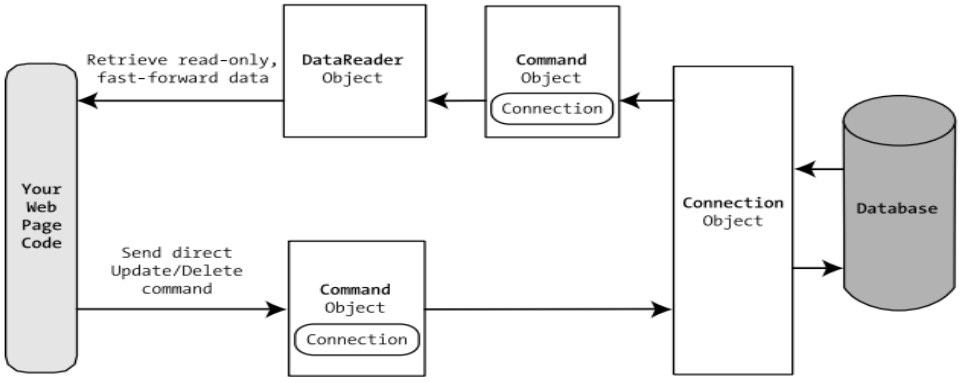
To query information with simple data access, follow these steps:

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 Connect ion and Command objects.
2. Execute the Command (with the appropriate SQL statement).

Direct data access with ADO.NET



Before continuing, make sure you import the ADO.NET namespaces:

```
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.

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.

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";
```

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. The only exception is if you're using a named instance of SQL Server. For example, if you've installed SQL Server 2005 Express Edition, you'll need to use the data source localhost\SQL EXPRESS, because the instance name is SQLEXPRESS. 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 you want to connect to SQL Server 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 how long your code will wait, in seconds, 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.

Storing the Connection String

Typically, all the database code in your application will use the same connection string. For that reason, 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 to ensure 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

Once you've created your connection, 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.

Here's the code with basic error handling:

```
'Define the ADO.NET Connection object.
String conStr = _
 WebConfigurationManager.ConnectionStrings("Pubs").ConnectionString;
SqlConnection con=new SqlConnection(conStr);

Try
{
   'Try to open the connection.
   con.Open();
  MessageBox.Show(con.State.ToString());
Catch (Exception Err){
   'Handle an error by displaying the information.
      MessageBox.Show("Error reading the database.");}
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.)
   Con.Close();}
}
```

The Command

Command is used to execute almost any SQL command from within the .net application. The SQL command like insert, update, delete, select, create, alter, drop can be executed with command object and you can also call stored procedures with the command object. Command object has the following important properties.

- Connection: used to specify the connection to be used by the command object.

- CommandType : Used to specify the type of SQL command you want to execute. To assign a value to this property, use the enumeration CommandType that has the members Text, StoredProcedure and TableDirect. Text is the default and is set when you want to execute ant SQL command with command object. StoredProcedure is set when you want to call a stored procedure or function and **TableDirect** is set when you want to retrieve data from the table directly by specifying the table name without writing a select statement.
- CommandText : Used to specify the SQL statement you want to execute.
- Transaction: Used to associate a transaction object to the command object so that the changes made to the database with command object can be committed or rollback.

Command object has the following important methods.

- *ExecuteNonQuery*() : Used to execute an SQL statement that doesn't return any value like insert, update and delete. Return type of this method is int and it returns the no. of rows affected by the given statement.
- *ExecuteScalar*() : Used to execute an SQL statement and return a single value. When the select statement executed by executescalar() method returns a row and multiple rows, then the method will return the value of first column of first row returned by the query. Return type of this method is object.
- *ExecuteReader*() **:** Used to execute a select a statement and return the rows returned by the select statement as a DataReader. Return type of this method is **DataReader.**

The DataReader

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

```
con.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.
      String conStr = _
      WebConfigurationManager.ConnectionStrings("Pubs").ConnectionString;
      SqlConnection con=new SqlConnection(conStr);
      SqlCommand cmd=new SqlCommand(_
       "SELECT * FROM Authors ORDER BY au_lname ", myConnection)
      SqlDataReader rdr=new SqlDataReader();
      rdr = cmd.ExecuteReader();
      myReader.Read(); 'The first row in the result set is now available.
```

- These lines of code define a variable for a DataReader and then create it by executing the command. Once you have the reader, you retrieve a single row at a time 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 using the corresponding field names.
- 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:

```
Rdr.close();
Con.close();
```

Disconnected Data Access

When you use disconnected data access, you keep a copy of your data in memory using the DataSet. 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 (like a grid) 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.
- You need a convenient package to send data from one component to another.
- You want to store some data so it can be used for future requests.

DataSets

- Datasets store data in a disconnected cache.
- The structure of a dataset is similar to that of a relational database; it exposes a hierarchical object model of tables, rows, and columns.
- In addition, it contains constraints and relationships defined for the dataset.

DataAdapters

Data adapters are used to communicate between a data source and a dataset.

Adapters are used to exchange data between a data source and a dataset.

In many applications, this means reading data from a database into a dataset, and then writing changed data from the dataset back to the database. However, a data adapter can move data between any source and a dataset.

Visual Studio makes these data adapters available for use with databases:

- The `OleDbDataAdapter` object is suitable for use with any data source exposed by an OLE DB provider.

- The `SqlDataAdapter` object is specific to SQL Server. Because it does not have to go through an OLE DB layer, it is faster than the **OleDbDataAdapter**. However, it can only be used with SQL Server 7.0 or later.
- The `OdbcDataAdapter` object is optimized for accessing ODBC data sources.
- The `OracleDataAdapter` object is optimized for accessing Oracle databases.

Every DataAdapter can hold four commands:

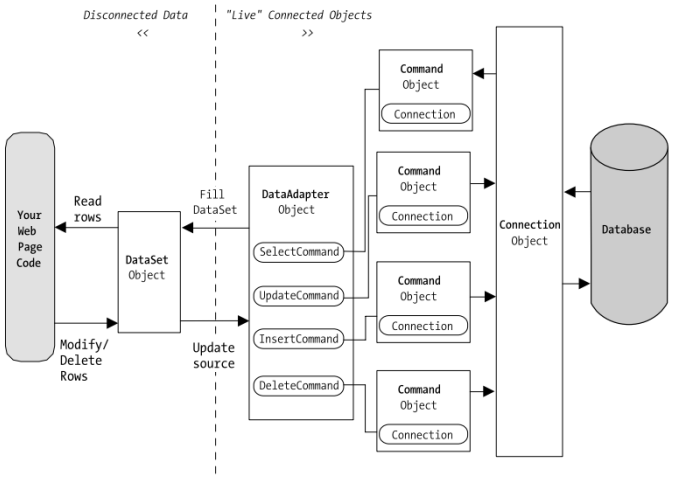
1. SelectCommand,
2. InsertCommand,
3. UpdateCommand, and
4. DeleteCommand.

This allows you to use a single DataAdapter object for multiple tasks. The Command object supplied in the constructor is automatically assigned to the DataAdapter.SelectCommand property.

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 using the field name, as it was with the DataReader.

Selecting Disconnected Data



With disconnected data access, a copy of the data is retained in memory while your code is running.

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.

```
private void Demo()
```

```
{
    // Define ADO.NET objects.
    string selstr;
    selstr = "SELECT au_lname, au_fname, au_id FROM Authors";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataAdapter dap = new SqlDataAdapter(cmd);
    DataSet ds = new DataSet();
    // Try to open database and read information.
    try
    {
        con.Open();
        // All the information in transferred with one command.
        // This command creates a new DataTable (named Authors)
        // inside the DataSet.
        dap.Fill(ds, "Authors");
    }
    catch (Exception err)
    {
        Interaction.MsgBox("Error");
    }
    finally
    {
        con.Close();
    }
}
```

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.

ASP.NET data binding, 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.

Types of ASP.NET Data Binding

1. *single-value binding* and
2. *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.

1. ***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.

2. Repeated-Value, or "List," Binding

Repeated-value data binding allows you to display an entire table.

Unlike single-value data binding, this type of data binding requires a special control that supports it. Typically, this will be a list control such as CheckBoxList or List Box, but it can also be a much more sophisticated control such as the GridView.

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 in *single-value* and *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.

Once 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. Once 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.

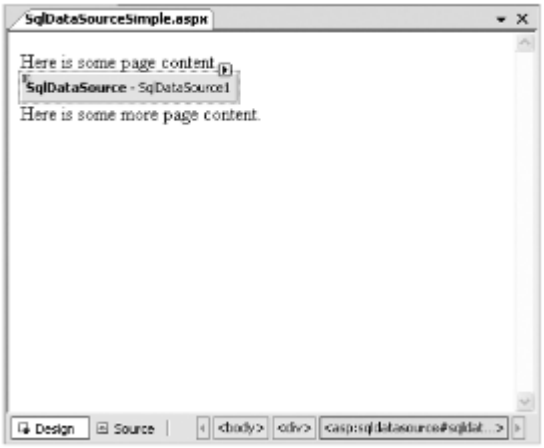
Data Source Controls

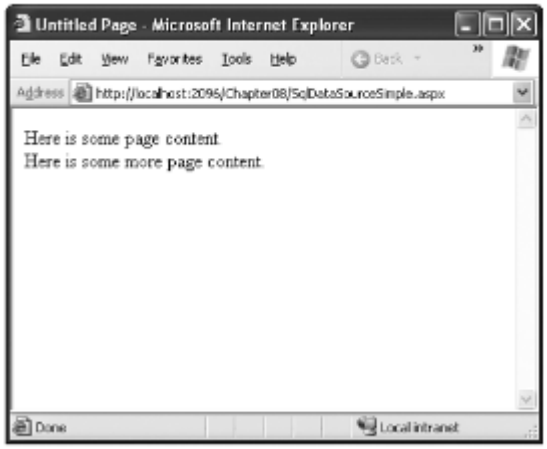
Data source controls allow you to create data-bound pages without writing any data access code at all.

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).
- *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.
- *XmlDataSource*: This data source allows you to connect to an XML file.
- *SiteMapDataSource*: This data source allows you to connect to a .sitemap file that describes the navigational structure of your website.

You can find all the data source controls in the Data tab of the Toolbox in Visual Studio. 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.





The Data Controls:

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.

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 three rich data controls 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. 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 t o 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 AD O.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").ConnectionStri
ng;
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.

Of course, you don't need to write this data access code by hand, 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 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= " source
Products" />
```

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.

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, what if 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 Auto `GenerateColumns` to False and define the columns in the <`Columns`> section of the GridView control tag. *Each column can be any of several column types:*

Class	Description
`BoundField`	This column displays t ext from a field in the data source.
`ButtonField`	This column displays a button in this grid column.
`CheckBoxField`	This column displays a check box in this grid column. It's

	used automatically for true/false fields.
`CommandField`	This column provides selection or editing buttons.
`HyperLinkField`	This column displays its contents (a field from the data source or static text) as a hyperlink.
`ImageField`	This column displays image data from a binary field.
`TemplateField`	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 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 t o 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.

BoundField Properties

Property	Description
`DataField`	Identifies the field (by name) that you want to display in

	this column.
`DataFormatString`	Formats the field. This is useful for getting the right representation of numbers and dates.
`ApplyFormatInEditMode`	If True, the DataFormat 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).
`FooterText, HeaderText, and HeaderImageUrl`	Sets the text in the header and footer region of the grid if this grid has a header (GridView.ShowHeader is True) and footer (GridView.ShowFooter 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 instead of text, set the HeaderI mageUrl property
`ReadOnly`	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.
`InsertVisible`	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.
`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.
`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 a better idea 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 hardcoded 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.

Generating Columns with Visual Studio

We 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 over 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.

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.

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.

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

- *Formatting: How to format rows and data values*
- *Selecting:* How t o 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).

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 (s o 3400.34 becomes $3,400.34). Here's a column that uses this format string:

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

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

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}	1.234.50E+004
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 12/30/08), you use the following column:

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

Time and Date Format Strings

Type	Format String	Syntax	Example
Short Date	{0:d}	M/d/yyyy	10/30/2008
Long Date	{0:D}	dddd, MMMM dd, yyyy	Monday, January 30, 2008
Long Date and Short Time	{0:f}	dddd, M MMM dd, yyyy HH:mm aa	Monday, January 30, 2008 10:00 AM
Long Date and Long Time	{0:F}	dddd, M MMM dd, yyyy HH:mm:ss aa	Monday, January 30 2008 10:00:23 AM
ISO Sortable Standard	{0:s}	yyyy-MM-ddTHH:mm:ss	2008-01-30T10:00:23
Month and Day	{0:M }	MMMM dd	January 30
General	{0:G}	M/d/yyyy HH:mm:ss aa	10/30/2008 10:00:23 AM

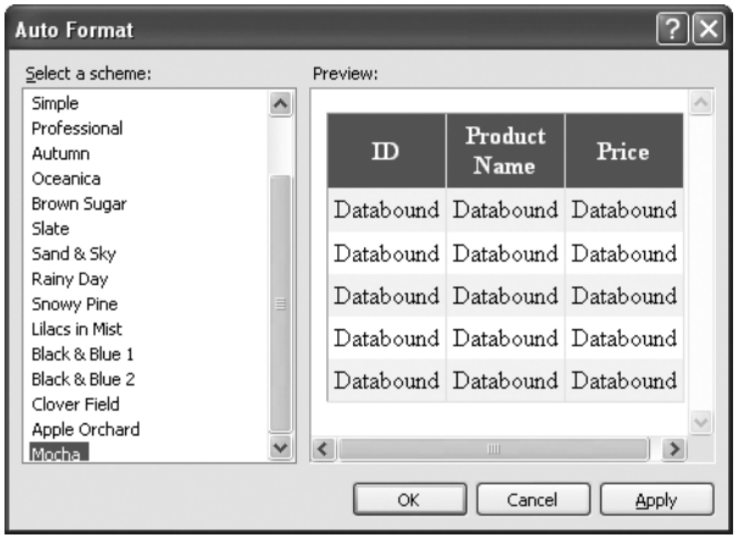
The format characters are not specific to the GridView. You can use them with other controls, with data-bound expressions in templates (as you'll see later in the "Using GridView Templates" section), and as parameters for many methods. For example, the Decimal and `DateTime` types expose their own `ToString`()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

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 t he header, expand the `HeaderStyle` property to show the nested Font property, and set that. The only limitation of 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 17-2) b y 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 shows the Auto Format dialog box with some of the preset styles you can choose. Select Remove Formatting to clear all the style settings.



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

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.

You can determine which row is selected in a `GridView` control by using any of the following properties:

- `SelectedDataKey` Returns the `DataKey` object associated with the selected row (useful when there are multiple data keys).

- `SelectedIndex` Returns the (zero-based) index of the selected row.
- `SelectedValue` Returns the data key associated with the selected row.
- `SelectedRow` Returns the actual row (`GridViewRow` object) associated with the selected row.

In most cases, you use the `SelectedValue` property to determine the value associated with a particular row. The `SelectedValue` property returns the data key associated with a row.

To find out what item is currently selected (or to change the selection), you can use the `GridView.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.

Adding a Select Button

You simply need to add a button 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" />
```

Using a Data Field as a Select Button

You can also 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.

```
<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 builds master-details pages—pages that let you navigate relationships in a database.

A typical master-details page has two GridView controls. The first 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 customer's 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.

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 `CustomerID` values in a list of customers:

```
<asp:GridView ID="gridCustomers" runat="server"
DataKeyNames="CustomerID" ... >
```

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.

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.

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 edit mode, you set the `EditIndex` property to the corresponding row number

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>
```

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. 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.

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.

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.

Sorting and Paging the GridView

The GridView has two features that make data more manageable: *sorting* and *paging*.

Sorting

The GridView sorting features allow the user to reorder the results in the GridView by clicking a column header.

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. A sort expression can use any syntax that's understood by the data source control. A sort expression almost always takes the form used in the ORDER BY clause of a SQL query.

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" />
```

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.

Not all data sources support sorting, but the `SqlDataSource` does, provided the `DataSourceMode` property is set to `DataSet` (the default), not `DataReader`. In DataSet mode, the entire results are placed in a DataSet and then the records are copied from the DataSet into the bound control.

Paging

When working with a large number of database rows, it is useful to be able to display the rows in different pages. You can enable paging with the `GridView` control by enabling its `AllowPaging` property.

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

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 num ber 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.

Paging Members of the GridView

Property	Description
`AllowPaging`	Enables or disables the paging of the bound records. It is False by default.
`PageSize`	Gets or sets the number of items to display on a single page of the grid. The default value is 10.
`PageIndex`	Gets or sets the zero-based index of the currently displayed page, if paging is enabled.
`PagerSettings`	Provides a PagerSettings 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.
`PagerStyle`	Provides a style object you can use to configure fonts, colors, and text alignment for the paging controls.

`PageIndexChanging` and `PageIndexChanged` events	Occur when one of the page selection elements is clicked, just before the PageIndex is changed (PageIndexChanging) and just after (PageIndexChanged).

This is enough to start using paging. Figure 17-11 shows an example with ten records per page (for a total of eight pages).

The DetailsView:

ASP.NET includes two controls: 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.

The DetailsView

The DetailsView displays a single record at a time. It places each field in a separate row of a table.

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.

One problem is that a separate postback is required each time the user moves from one record to another. 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.

Defining Fields

The DetailsView can use the same fields that the GridView uses. You can disable this automatic row generation by setting `AutoGenerateRows` to False. Its 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. The following code defines a DetailsView that shows product information.

```
<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>
```

The FormView:

You can use the FormView control to do anything that you can do with the DetailsView control. You can use the FormView control to display, page, edit, insert, and delete database records. However, unlike the DetailsView control, the FormView control is entirely template driven.

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 the model of the TemplateField in the GridView quite closely. This means you can work with the following templates:

- ItemTemplate
- EditItemTemplate
- InsertItemTemplate
- FooterTemplate
- HeaderTemplate
- EmptyDataTemplate
- PagerTemplate

You can use the same template content you use with a `TemplateField` in a GridView in the `FormView`. 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. 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.

A FormView that shows a single record

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
 ConnectionString="<%$ ConnectionStrings:Northwind %>"
 SelectCommand="SELECT * FROM Products">
</asp:SqlDataSource>
<asp:DropDownList ID="lstProducts" runat="server"
 AutoPostBack="True" DataSourceID="sourceProducts"
 DataTextField="ProductName" DataValueField="ProductID"
Width="184px">
</asp:DropDownList>
```

The Repeater:

The Repeater control is used to display a repeated list of items that are bound to the control. The Repeater control may be bound to a database table, an XML file, or another list of items.

Repeater is a Data Bind Control. Data Bind Controls are container controls. Data Binding is the process of creating a link between the data source and the presentation UI to display the data. ASP .Net provides rich and wide variety of controls, which can be bound to the data.

Repeater has five inline templates to format it:

- HeaderTemplate
- FooterTemplate
- ItemTemplate
- AlternatingItemTemplate
- SeperatorTemplate
- AlternatingItemTemplate

HeaderTemplate: This template is used for elements that you want to render once before your ItemTemplate section.

FooterTemplate: - This template is used for elements that you want to render once after your ItemTemplate section.

ItemTemplate: This template is used for elements that are rendered once per row of data. It is used to display records.

AlternatingItemTemplate: This template is used for elements that are rendered every second row of data. This allows you to alternate background colors. It works on even number of records only.

SeperatorTemplate: It is used for elements to render between each row, such as line breaks.

Some point about Repeater Control

- It is used to display backend result set. It is used to display multiple tuple.

- It is an unformatted control. The Repeater control is a basic templated data-bound list. It has no built-in layout or styles, so you must explicitly declare all layout, formatting, and style tags within the control's templates.

- The Repeater control is the only Web control that allows you to split markup tags across the templates. To create a table using templates, include the begin table tag (<table>) in the `HeaderTemplate`, a single table row tag (<tr>) in the `ItemTemplate`, and the end table tag (</table>) in the `FooterTemplate`.

- The Repeater control has no built-in selection capabilities or editing support. You can use the `ItemCommand` event to process control events that are raised from the templates to the control.

Handling Repeater Control Events

The Repeater control supports the following events:

- `DataBinding` Raised when the Repeater control is bound to its data source.
- `ItemCommand` Raised when a control contained in the Repeater control raises an event.
- `ItemCreated` Raised when each Repeater item is created.
- `ItemDataBound` Raised when each Repeater item is bound.

The DataList Control

The DataList control, like the Repeater control, is template driven. Unlike the Repeater control, by default, the DataList renders an HTML table. Because the DataList uses a particular layout to render its content, you are provided with more formatting options when using the DataList control.

```
<asp:DataList
        id="dlstMovies"
        DataSourceID="srcMovies"
        RepeatLayout="Flow"
        Runat="server">
        <ItemTemplate>
        <%#Eval("Title")%>
        </ItemTemplate>
</asp:DataList>
```

DataList control in above example includes a `RepeatLayout` property that has the value `Flow`. Each movie title is rendered in a `<span>` tag followed by a line-break tag.

The `RepeatLayout` property accepts one of the following two values:

- `Table` Data Items are rendered in HTML table cells.
- `Flow` Data Items are rendered in HTML `<span>` tags.

Displaying Data in Multiple Columns

You can render the contents of a `DataList` control into a multi-column table in which each data item occupies a separate table cell. Two properties modify the layout of the HTML table rendered by the DataList control:

- `RepeatColumns` The number of columns to display.
- `RepeatDirection` The direction to render the cells. Possible values are `Horizontal` and `Vertical`.

For Example:

```
<asp:DataList
        id="dlstMovies"
        DataSourceID="srcMovies"
        RepeatColumns="3"
        GridLines="Both"
        Runat="server">
        <ItemTemplate>
       <%#Eval("Title")%>
        Directed by:
        <%#Eval("Director") %>
        <br />
        Box Office Totals:
        <%#Eval("BoxOfficeTotals","{0:c}") %>
        </ItemTemplate>
    </asp:DataList>
```

Using Templates with the DataList Control

The DataList control supports all the same templates as the `Repeater` control:

- **`ItemTemplate`** Formats each item from the data source.
- **`AlternatingItemTemplate`** Formats every other item from the data source.
- **`SeparatorTemplate`** Formats between each item from the data source.
- **`HeaderTemplate`** Formats before all items from the data source.
- **`FooterTemplate`** Formats after all items from the data source

In addition, the `DataList` supports the following templates:

- **`EditItemTemplate`** Displayed when a row is selected for editing.
- **`SelectedItemTemplate`** Displayed when a row is selected.

Security Fundamentals

Ordinarily, your ASP.NET website is available to anyone who connects to your web server, whether over a local network or the Internet. Although this is ideal for many web applications, it isn't always an appropriate design choice.

For example, an e-commerce site needs to provide a secure shopping experience to customers. A subscription-based site needs to limit content to extract a fee. And even a wide-open public site may provide some resources or features that shouldn't be available to all users.

ASP.NET provides an extensive security model that makes it easy to protect your web applications. Although this security model is powerful and very flexible, it can appear confusing because of the many different layers that it includes. Much of the work in securing your application isn't writing code, but determining the appropriate places to implement your security strategy.

There are two ways to authenticate users:

1. Using forms authentication (which is ideal for a public website that uses a custom database) and

2. Using Windows authentication (which is ideal for an intranet application on a company network).

Determining Security Requirements

- The first step in securing your applications is deciding where you need security and what it needs to protect.
  - For example, you may need to block access in order to protect private information. Or, maybe you just need to enforce a pay-for-content system. Perhaps you don't need any sort of security at all, but you want an optional login feature to provide personalization for frequent visitors. These requirements will determine the approach you use.
- Security doesn't need to be complex but it does need to be wide-ranging and multilayered.
  - For example, consider an e-commerce website that allows users to view reports of their recently placed orders. You probably already know the first line of defense that this website should use—a login page that forces users to identify themselves before they can see any personal information.
- However, it's important to realize that, on its own, this layer of protection is not enough to truly secure your system. You also need to protect the back-end database with a strong password, and you might even choose to encrypt sensitive information before you store it. Steps like these protect your website from other attacks that get beyond your authentication system.
  - For example, they can prevent dissatisfied employee with an account on the local network, a hacker who has gained access to your network through the company firewall, or a careless technician who discards a hard drive used for data storage without erasing it first.
- Furthermore, you'll need to hunt carefully for weaknesses in the code you've written. A

surprising number of websites fall prey to relatively simple attacks in which a malicious user simply tampers with a query string argument or a bit of HTML in the rendered page.

- For example, in the e-commerce example you need to make sure that a user who successfully logs in can't view another user's recent orders. Imagine you've created a ViewOrders.aspx page that takes a query string argument named userID, like this: http://localhost/InsecureStore/ViewOrders.aspx?userID=4191

- This example is a security nightmare, because any user can easily modify the userID parameter by editing the URL to see another user's information. A better solution would be to design the page so that it gets the user ID from the currently logged-on user identity, and then uses that to construct the right database command.
- When designing with security in mind, it's important to consider the different avenues for attack. However, you can't always anticipate potential problems. For that reason, it makes great sense to layer your security. The mantra of security architects can be summed up like this: "Don't force an attacker to do one impossible thing to break into your system—force them to do several."