PRACTICAL INSIGHT AND ADVICE FROM THE EXPERTS

ASP.NET 1.1

INSIDER SOLUTIONS

SAMS

Alex Homer, Dan Kent, Dave Sussman, Dan Wahlin

Alex Homer
Dan Kent
Dave Sussman
Dan Whalin

ASP.NET 1.1

SAMS 800 East 96th Street, Indianapolis, Indiana 46240

ASP.NET 1.1 Insider Solutions

International Standard Book Number: 0-672-32674-4

Library of Congress Catalog Card Number: 2004091341

Printed in the United States of America

First Printing: June 2004

07 06 05 04 4 3 2 1

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

1-317-428-3341

international@pearsontechgroup.com

Associate Publisher
Michael Stephens

Acquisitions Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Charlotte Clapp

Project Editor
Dan Knott

Copy Editor
Kitty Jarrett

Indexer
Heather McNeill

Proofreader
Katie Robinson

Technical Editors
Dan Maharry
Justin Rogers

Team Coordinator
Cindy Teeters

Designer
Gary Adair

Page Layout
Bronkella Publishing

Contents at a Glance

Table of Contents

About the Authors

Alex Homer began his love/hate relationship with computers in 1980, with the Altair and Sinclair Z80, and he now lives and works in the idyllic rural surroundings of the Derbyshire Dales in England. Alex has written or contributed to more than 30 books on Web development topics for major publishers. He is a Microsoft MVP and INETA member, and he speaks regularly at conferences around the world. In what spare time is left, he runs his own software and consulting company, Stonebroom Limited (`http://stonebroom.com`).

Dave Sussman is a freelance writer, trainer, and consultant who lives in a rural village in England. He spends most of his time in betaland, a strange place inhabited by test software that changes daily and where there only seem to be 12 hours in a day. He strongly believes in the Douglas Adams view of deadlines. He can be contacted at `davids@ipona.com`.

Dan Wahlin, a Microsoft MVP, is the president of Wahlin Consulting and founded the XML for ASP.NET Developers Web site (`www.XMLforASP.NET`), which focuses on using XML and Web services in the .NET platform. In addition to consulting, Dan is also a corporate trainer/speaker, and he teaches XML and .NET training courses around the United States. Dan coauthored *ASP.NET: Tips, Tutorials, and Code* and authored *XML for ASP.NET Developers* (both from Sams Publishing).

Dan Kent currently edits the *Evolution* series for Sams Publishing, builds sites that support community regeneration, and performs cutting-edge video shows as half of VJ duo Syzygy.

After studying artificial intelligence, he went on to become part of the dot-com bubble, building online community sites that empowered newcomers to the Web to create Web presences. He decided to leave frontline programming and concentrate on passing on some of his know-how. His desire to be involved with books was kindled by some work as a technical reviewer for Wrox, which he went on to as a technical editor.

While at Wrox, Dan developed the Problem-Design-Solution concept, which pioneered the approach of presenting readers with real-world solutions in the context of real applications. He also worked with the Microsoft ASP.NET team to help programmers learn more about the fantastic technology they created and contributed as an author to the highly respected *Professional ASP.NET Security*, now sadly out of print. Two years, four job titles, and far too many books later, Dan decided to leave Wrox.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

For more information about this book or another Sams Publishing title, visit our Web site at `www.samspublishing.com`. Type the ISBN (0672326744) or the title of the book into the Search field to find the page you're looking for.

Introduction

Are you getting the most from ASP.NET? While it's easy to build quite complex pages quickly and easily with ASP.NET, if you acquire a more intimate knowledge of the .NET Framework as a whole, you can really take advantage of the great features it provides.

This book explores some of the more advanced topics that help you to build better, more efficient, and more attractive Web pages and Web applications. In fact, many of the examples in this book are designed to illustrate and provide solutions for questions and problems that appear regularly on the ASP.NET forums and newsgroups.

What This Book Covers

Topics include getting more from the `DataGrid` control, creating reusable content as both user and server controls, using page templating and cross-page posting, building secure applications, validating user input, integrating client-side script, providing great cross-browser support, and much more.

The book is divided into four sections:

- Part I, "Web Form User Interfaces," is a combination of many useful techniques for solving issues that ASP.NET developers regularly face. The chapters in this part include tips and tricks with Web forms and information on cross-page posting, displaying progress and status information, and working with nested ASP.NET list controls.
- Part II, "Reusability," demonstrates how you can create reusable content for Web pages and applications. Topics include client-side script integration; user and server control design and construction; adaptive controls; and master pages, templates, and page subclassing techniques.
- Part III, "Data Techniques," covers some of the issues that you should think about when working with both relational and XML data, including tips and tricks, protecting your server, and performance.
- Part IV, "Hosting and Security," covers topics that are mainly concerned with installing, setting up, and using ASP.NET. This includes side-by-side execution of different versions, ASP.NET forms authentication, and general security configuration issues.

Who This Book Is For

This book is for developers who are using ASP.NET and have a reasonable grasp of the basic topics for building Web pages and Web applications in ASP.NET. It is not designed to act as a beginner's guide or as a comprehensive reference to all the techniques available in ASP.NET. However, the topics that it does cover are introduced in sufficient depth that a reasonably experienced ASP.NET user will be able to learn and take advantage of the techniques described.

For example, Chapter 5, "Creating Reusable Content," explains what user and server controls are and how to build them—in such a way that the reader does not need to have any prior experience of these topics. It describes and illustrates properties and methods, how to expose functionality from a control, and how to use that control in Web pages and applications.

What You Need to Use This Book

This book covers ASP.NET 1.1, and you must be running this version of ASP.NET to use the sample code that is available for download. The examples are not designed for use in Visual Studio .NET, which means that you can use them (and edit them to suit your own projects) in tools such as Web Matrix or in any text editor. You can, of course, convert them yourself to run within Visual Studio .NET if you wish.

All the sample code for this book can be downloaded from the Sams Web site at `www.samspublishing.com`. It is also available at `www.daveandal.net/books/6744/`, where you can run many of the examples online without needing to download them and install them on your own server.

Many of the examples in this book rely on a database server to provide values for the pages. The database used in the book is the sample Northwind database provided with SQL Server and MSDE, and a suitable Access database is included with the downloadable samples for the book as well. You can use a different database server if you prefer, provided that you have a managed provider for the .NET Framework available, and you must edit the connections strings in the `web.config` file to specify your database server.

Other than that, you can run the examples and experiment with the techniques they illustrate without requiring any other special software or hardware.

Conventions Used in This Book

Special conventions are used to help you get the most from this book and from Web markup.

Text Conventions

Various typefaces in this book identify terms and other special objects.

- Screen messages, code listings, and command samples appear in `monospace type`.
- Uniform Resource Locators (URLs) used to identify pages on the Web and values for HTML attributes also appear in `monospace type`.

- Terms that are defined in the text appear in *italics*. *Italics* are sometimes used for emphasis, too.
- In code lines, placeholders for variables are indicated by using `italic monospace type`.
- User input information will appear in **`bold monospace type`**.

Special Elements

Throughout this book, you'll find best practices, sidebars, and cross-references. These elements provide a variety of information, ranging from warnings you shouldn't miss to ancillary information that will enrich your learning experience:

Sidebars for More Information

Sidebars are designed to provide information that is ancillary to the topic being discussed. Read these if you want to learn more about an application or a task.

BEST PRACTICE

Best Practices

Bets practices are designed to help you decide which is the best way to approach the task being discussed and which is the best way to make use of the technology or maximize its benefits.

PART I

Web Form User Interfaces

1

Web Forms Tips and Tricks

We start this chapter by looking at some of the more unusual ways you can use the ASP.NET validation controls, such as within a list control—something that comes up regularly on ASP.NET mailing lists and forums.

Next, we take a brief look at creating something other than the standard layout in a `DataGrid` control. We show a couple examples that demonstrate how you can specify the width of the columns, expose more than one editable value in a column, and display long text strings in scrollable cells.

Finally, we look at a topic that seems to regularly cause problems for users: creating instances of controls dynamically when a page is loaded. This technique can provide far more flexibility than just declaring all the controls within the HTML section of the page, but it means you have to be fairly organized when developing the page—and remember to re-create all the controls in the correct order on each postback.

IN THIS CHAPTER

Getting More from ASP.NET Validation Controls

Many developers do not realize just how versatile the validation controls provided with ASP.NET are. The common scenario is to use them to validate the contents of a text box, a task that they are ideally suited to. However, you can also use them to validate almost any Web Forms or HTML control as well. For example, if you have a list box or a drop-down list that has a "dummy" entry displayed by default, you can force users to select one of the other values in the list by using a validation control.

Suppose that the list is populated as follows:

```
<asp:ListBox id="TheListBox" runat="server">
  <asp:ListItem Text="Please select a value..." Value="" />
  <asp:ListItem Text="Value1" Value="1" />
  <asp:ListItem Text="Value2" Value="2" />
  <asp:ListItem Text="Value3" Value="3" />
</asp:ListBox>
```

The first entry in the list has a value for the text (the `Text` property and the corresponding content of the `<option>` element that is generated). However, it has no value (the `Value` property and the corresponding value attribute that is generated are empty strings). Therefore, you can use a `RequiredFieldValidator` control to force the user to select an entry in the list that does have a value:

```
<asp:RequiredFieldValidator id="ListValRequired" runat="server"
     ControlToValidate="TheListBox"
     ErrorMessage="You must select a value in the list">
  *
</asp:RequiredFieldValidator>
```

Likewise, you can use other validation controls to force a specific value to be selected. Of course, it's likely that your list control will contain only valid values anyway. However, one possible reason for validating the selected value might be to compare it to another control. This example uses a `CompareValidator` control to make sure the same selection is made in two list controls:

```
<asp:CompareValidator id="ListLimitValue" runat="server"
     ControlToValidate="TheListBox"
     ControlToCompare="AnotherList"
     Operator="Equal"
     Type="String"
     ErrorMessage="You must select the same value in both lists">
  *
</asp:CompareValidator>
```

Validating a `RadioButtonList` Control

A scenario that may arise is a situation in which you use a `RadioButtonList` control to generate a list of option buttons. You might decide that there is an obvious "default" option and preselect it by setting the `SelectedIndex` property when you generate the list. However, this can result in users submitting the value without actually considering whether it is the appropriate one—they might just click the Submit button without reading all the options.

To get around this, you can add a `RequiredFieldValidator` control to force the user to select one of the options, without having to specifically select one of the options as the default in your code. If the user makes no selection, the "value" of the `RadioButtonList` control is an empty string:

```
<asp:RequiredFieldValidator id="RadioListValRequired" runat="server"
     ControlToValidate="TheRadioButtonList"
     ErrorMessage="You must select a value in the radio button list">
  No value selected
</asp:RequiredFieldValidator>
```

And, of course, you can use another validation control to check the value that the user selected—just as in the earlier list control example. The following code uses a `RangeValidator` control with the comparison type set to `"String"` to perform a case-sensitive check that the `Value` property of the selected option button is between `"W"` and `"Z"`:

> **Validating Option Buttons**
>
> **If you are validating against text values for the `Value` properties of the option buttons, it generally makes sense to use a `RegularExpressionValidator` control instead of a `RangeValidator` control. This provides far wider opportunities for accurately specifying what is valid, rather than relying just on a specific range of character codes.**

```
<asp:RangeValidator id="RadioListValue" runat="server"
     ControlToValidate="TheRadioButtonList"
     MinimumValue="W"
     MaximumValue="Z"
     Type="String"
     ErrorMessage="You must select a value between W and Z">
  Invalid value selected
</asp:RangeValidator>
```

The Location of the Error Message

Notice in the preceding section that we avoid the common use of an asterisk (*) for the content of the validation control in both the examples of validating a `RadioButtonList` control. Normally, for text boxes and list controls, the content of the validation control is displayed next to the control when it contains an invalid value.

However, the default for the `RadioButtonList` control (and the `CheckBoxList` control) is to generate an HTML table. This means that the content of the validation control will appear below the list control rather than next to it. Specifying a meaningful message, rather than just an asterisk, makes it easier to see where the error is, as shown in Figure 1.1.

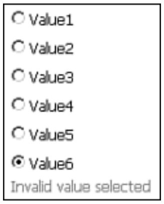


FIGURE 1.1 Displaying meaningful error messages below a `RadioButtonList` control.

> **Validating String and Numeric Values**
>
> **If you use the comparison type `"String"` when values are numeric, you'll get inaccurate results. For example, if the `MinimumValue` property of the validation control is `"5"`, the value 10 will be considered to be invalid because it comes before `"5"` in alphabetic (character-code) order.**

Performing Numeric Comparisons

You need to take some care if you are using numeric values for the `Value` property of items in a list control and then attaching a validation control. Remember to specify the correct comparison type because the `"String"` comparison type treats the values differently from the `"Integer"` type—the concepts of "less than" and "greater than" are different for strings and numbers:

```
<asp:RangeValidator id="RadioListValue" runat="server"
     ControlToValidate="TheRadioButtonList"
     MinimumValue="2"
     MaximumValue="5"
     Type="Integer"
     ErrorMessage="You must select a value between 2 and 5">
  Invalid value selected
</asp:RangeValidator>
```

Setting Validation Properties Dynamically

One of the prime reasons for using list controls rather than text boxes in a page is to limit the selections that a user can make. Therefore, in most cases, the list of values that users can select from only contain valid options, rendering most validation other than requiring a selection to be made (using a `RequiredFieldValidator` control) unnecessary.

However, bear in mind that you can set the properties of validation controls dynamically on the server side, just as you do for any other Web Forms control. This means that you can react to other conditions (such as values selected in other pages, the time of day, the user location, and so on) to specify which options in a list are valid when the page is generated—while still displaying all the options.

For example, if you had a custom function that discovered the weather conditions for a specified city, you could write code in the `Page_Load` event handler to set the maximum and minimum values of a `RangeValidator` control named `ValidateWeather` like this:

```
If GetWeather("Manchester") = "Raining" Then
  ValidateWeather.MinimumValue="2"
  ValidateWeather.MaximumValue="3"
End If
```

Validating a `CheckBoxList` Control

A `CheckBoxList` control can support validation, but not along the same lines as other list controls. If you try to attach any validation control other than `CustomValidator` to a `CheckBoxList` control, you'll get the compiler error "Control '*control-id*' referenced by the ControlToValidate property of '*validator-id*' cannot be validated." This is because the `CheckBoxList` control does not expose a "value" property as do the `RadioButtonList` control and most other Web Forms and HTML controls.

There is another factor to consider here. The reason for using a `CheckBoxList` control is to offer the user the opportunity to select more than one value. (If the user could select only one value, you would probably use a `RadioButtonList` control instead.)

However, you can use a `CustomValidator` control in conjunction with a `CheckBoxList` control to perform most kinds of validation, if required. For example, you can force the user to select one (or more) of the check boxes and prevent the form from being submitted with no check boxes selected. Or you can limit the number that can be checked or even perform tests against the captions of those that are checked or unchecked.

The `CustomValidator` control requires that you write server-side, and optionally client-side, functions to perform the actual validation. For example, if you have a `CheckBoxList` control with its `id` property set to `MyCheckBoxList`, you can attach a `CustomValidator` control to it like this:

```
<asp:CustomValidator id="ValidateCheckBoxList" runat="server"
     ClientValidationFunction="ClientValidateCheckboxList"
     OnServerValidate="ServerValidateCheckboxList"
     ErrorMessage="You cannot select more than five checkboxes">
  More than five checkboxes selected
</asp:CustomValidator>
```

Then it's just a matter of writing the server-side and client-side validation functions. The server-side function can use the `Items` collection exposed by the `CheckBoxList` control to count the number of check boxes that are set (their `Checked` property is `True`). If the result is five or fewer, you return `True` so that the validation control will return `True` for its `IsValid` property:

```
Sub ServerValidateCheckboxList(sender As Object, _
                               e As ServerValidateEventArgs)
  Dim iCount As Integer = 0
  For Each oCheck As ListItem In MyCheckBoxList.Items
    If oCheck.Selected Then
      iCount += 1
    End If
  Next
  e.IsValid = (iCount <= 5)
End Sub
```

You can also access the captions of each `CheckBox` control through the `Text` property of each entry in the `Items` collection, or you can simply use their index positions within the collection to see which are checked or unchecked.

The next section of code shows the function called by the CustomValidator control to perform the same validation test client side in JavaScript. To get a reference to the check boxes, it iterates through the first <form> element on the page, checking the name (ID) of each control it finds to see if it is one of the check boxes in the CheckBoxList control (whose names are all in the form MyCheckBoxList_*n*):

```
function ClientValidateCheckbox(source, args) {
  var iCount = 0;
  var aCtrls = document.forms[0].elements;
  for (var i=0; i < aCtrls.length; i++) {
    if (aCtrls[i].name.substring(0, 14) == 'MyCheckBoxList')
    {
      if (aCtrls[i].checked) iCount++;
    }
  }
  args.IsValid = (iCount <= 5);
}
```

BEST PRACTICE

Protecting Your Pages from Spoofing Attacks

You should always perform server-side validation—even if you perform it client side as well—to prevent any chance of the user spoofing your application by removing client-side validation code from the page or turning off script support on which the client-side validation depends.

Validating Nonstandard Control Values

Some controls, such as the CheckBoxList control we examined in the preceding section, don't fully support the use of validation controls. Another example is the Calendar control that is provided with ASP.NET. However, for all these types of controls, there is a simple technique you can use to perform server-side validation: You add an ASP.NET TextBox control to the page and then arrange for this to contain the current value of the control you want to validate when a postback occurs—by handling the appropriate On*Xxxxx*Changed event in your server-side code. All this event handler has to do is copy the current value from the control into the text box and then call the Validate method of the attached validation control(s).

For the CheckBoxList control, for example, you handle OnSelectedIndexChanged. You must also arrange for the control to cause a postback when the value changes by setting the AutoPostBack property:

```
<asp:CheckBoxList id="MyCheckBoxList" runat="server"
     OnSelectedIndexChanged="SetCBLTextbox"
     AutoPostback="True" />
```

Then you add the TextBox control, hiding it from view in the page by setting the visibility style selector to hidden:

```
<asp:Textbox id="CBLTextbox" Columns="1" runat="server"
     Style="visibility:hidden" />
```

Now you just add the validation controls you require to the page, specifying that they should validate the TextBox control and not the CheckBoxList control. For example, the following forces the user to select at least one check box:

```
<asp:RequiredFieldValidator id="RequireCBLSelection" runat="server"
     ControlToValidate="CBLTextbox"
     ErrorMessage="You must select at least one check box">
  No items selected
</asp:RequiredFieldValidator>
```

The server-side event handler that runs when the selection in the CheckBoxList control is changed is shown next. It just copies the value into the TextBox control and then calls the Validate method of the single validation control attached to the TextBox control:

```
Sub SetCBLTextbox(sender As Object, _
          e As EventArgs)
  CBLTextbox.Text =
   MyCheckBoxList.SelectedValue
  RequireCBLSelection.Validate()
End Sub
```

> **Hidden Controls and Validation**
>
> **You cannot use an HTML hidden-type <input> control here because that does not support the ASP.NET validation controls. This is why you instead use a TextBox control. You can also use Visible="False" to completely remove the TextBox control from the page, although it remains in the server-side control tree and still allows server-side validation to be performed. However, client-side validation will not be performed unless the text box is part of the page that is sent to the browser. Hiding it by using the visibility style selector allows the client-side validation to be performed when the page is first loaded (and no check boxes are selected) without the text box being visible in most modern browsers. You can always place it in some non-obvious position in the page in case a user's browser doesn't support CSS.**

Validating a Calendar Control

The technique described in the preceding section works with the ASP.NET Calendar control. This control automatically causes a postback to the server when a date is selected, so there is no AutoPostBack property to set this time, and you handle the OnSelectionChanged event. Listing 1.1 shows the declaration of the Calendar control, the hidden TextBox control, the two validation controls that are attached to the TextBox control, and the event handler for the OnSelectionChanged event.

LISTING 1.1 Validating a `Calendar` Control

```
<asp:Calendar id="TheCalendar" runat="server"
     OnSelectionChanged="SetCalendarTextbox" />

<asp:Textbox id="CalendarTextbox" Columns="1" runat="server"
     Style="visibility:hidden" />

<asp:RequiredFieldValidator id="RequireCalendarDate" runat="server"
     ControlToValidate="CalendarTextbox"
     ErrorMessage="You must select a date in the calendar">
  No date was selected
</asp:RequiredFieldValidator>

<asp:RangeValidator id="RangeCalendarDate" runat="server"
     ControlToValidate="CalendarTextbox"
     MinimumValue="01/01/2004"
     MaximumValue="31/01/2004"
     Type="Date"
     ErrorMessage="You must select a date in January 2004">
  An invalid date was selected
</asp:RangeValidator>

...

Sub SetCalendarTextbox(sender As Object, e As EventArgs)
  CalendarTextbox.Text = TheCalendar.SelectedDate
  RequireCalendarDate.Validate()
  RangeCalendarDate.Validate()
End Sub
```

This example requires the selection of a date, using a `RequiredFieldValidator` control, and it requires this date to be within the month of January 2004, by using a `RangeValidator` control. Notice that the comparison type is set to `"Date"` in this example and that a meaningful text string is used as the content of the validation controls because it will appear below the `Calendar` control (which is rendered as an HTML table).

Finally, the event handler copies the currently selected date into the `TextBox` control and then calls the `Validate` methods of the two validation controls that are attached to the `TextBox` control.

Using List and Validation Controls in a `DataGrid` Control

A situation that seems to cause a lot of questions on mailing lists and forums is the use of ASP.NET validation controls within a templated list control such as a `DataGrid`, `DataList`, or

`Repeater` control. The following sections show how easy it is to use these controls in a `DataGrid` control to validate the values entered by the user when a row is in "edit" mode.

The following sections also summarize the technique for using list controls within the rows of a `DataGrid` control, although this chapter uses only simple nested list controls such as the `DropDownList` and `RadioButtonList` controls.

To learn more about advanced topics for the `DataGrid` control, see Chapter 4, "Working with Nested List Controls."

The `DataGrid` Control Validation Sample Page

Figure 1.2 shows the sample page that the following sections explore. It lists 10 rows from the `Products` table in the Northwind database, with each row displaying a link to edit the row contents. Notice also that the Discontinued column, which is a bit column in the database (effectively a `Boolean` value), contains a graphic image in "normal" (non-edit) mode.

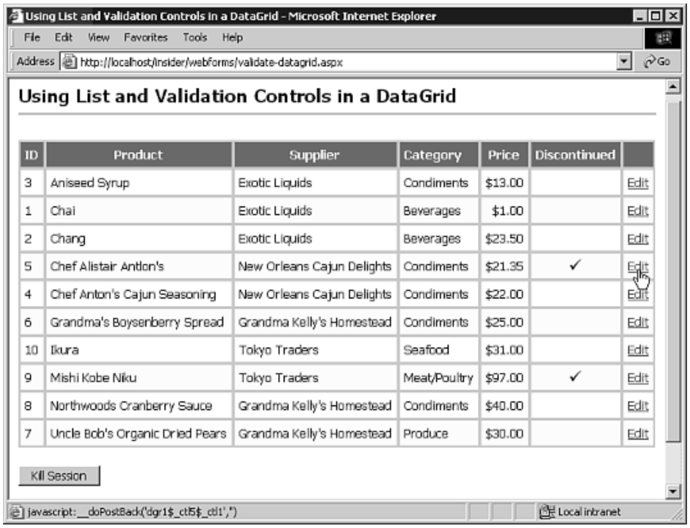


FIGURE 1.2
Using list and validation controls in a `DataGrid` control.

When an Edit link is clicked in the sample page, that row is displayed in edit mode, as shown in Figure 1.3. Notice that two of the columns display list controls, whose selections reflect the current value in the row. The user can only select a value from these list controls for these two columns when he or she edits a row.

Also notice that the Discontinued column now displays a check box where the user can effectively specify "yes" (checked) or "no" (not checked). To demonstrate the validation features of the page, there are some hints below the data grid on how to force an input error to occur. We'll come back and look at these features shortly.

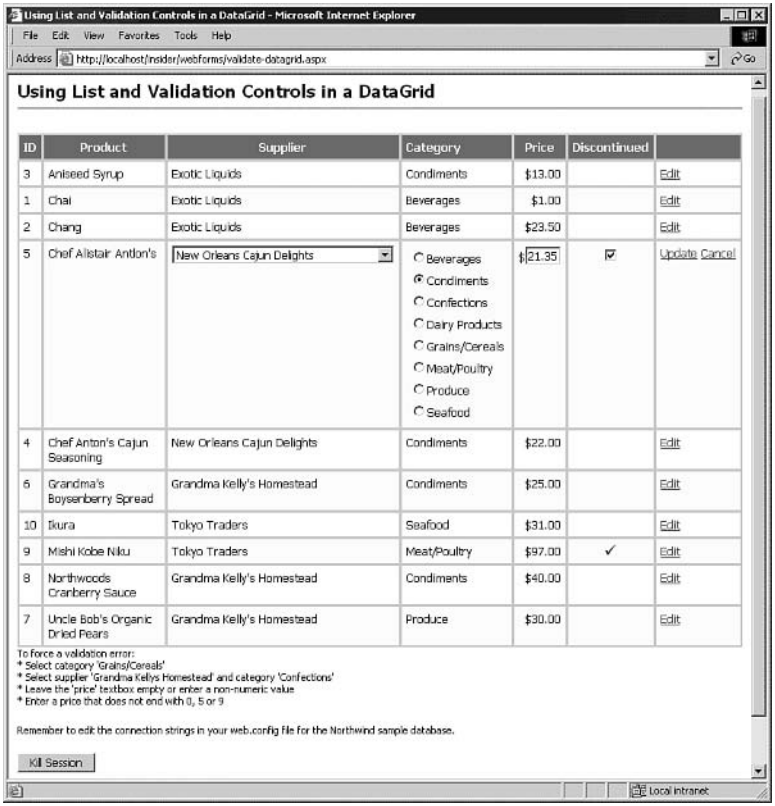


FIGURE 1.3
The sample page with one row in edit mode.

Storing a `DataSet` Object in the User's ASP.NET Session

The source data for this example comes from an ADO.NET `DataSet` instance that is populated from the database when the page is first loaded. In a `DataGrid` control that allows users to select and edit rows, one issue is the fact that the control has to be rebound to its data source every time the user selects a row or places a row into edit mode, as well as when the user clicks the Update or Cancel button in a row that is in edit mode.

This example also demonstrates how you can store this `DataSet` object in the user's ASP.NET session to minimize the number of trips you need to make to the database. The only time that you need to go back to the database and refresh the `DataSet` object's contents is when the user changes a value in any row. This is useful because the `DataSet` object contains not just one but three tables—meaning that a lot of processing is required to fill it.

The first table in the `DataSet` object contains the 10 rows you want to display from the `Products` table (to which you have to join the `Suppliers` and `Categories` tables in order to get the supplier and category names). The second and third tables in the `DataSet` object contain all the rows from the `Products` table (used to populate the drop-down list of products) and the `Categories` table (used to populate the radio button list of categories).

A `Button` control labeled Kill Session is located at the bottom of the page, and a message is displayed next to it when the `DataSet` object has been filled or refreshed from the database. Then, as the user interacts with the `DataGrid` control after the initial page load, he or she sees that this message is displayed only when a row is updated.

However, the user can force the `DataSet` object to be discarded and refilled on the next page load by clicking the Kill Session button. All this does is remove the existing `DataSet` object from the session, and after the page reloads, the user sees the message that a new one has been created and filled from the database.

Using Cookieless Sessions in ASP.NET

If the user's browser does not support cookies, or if cookies are blocked in the browser's security settings, the application described here cannot by default take advantage of ASP.NET sessions. One solution is to use the cookieless sessions feature, by adding a `<sessionState>` element to the `<system.web>` section of the `web.config` file in the application root folder:

```
<system.web>
  <sessionState cookieless="true" />
</system.web>
```

Downloading and Running This Example

You can download this example, as well as the rest of the samples for this book, from our Web site, at `www.daveandal.net/books/6744/`. You can also run several of the examples online from the same URL. It contains a [view source] link at the bottom of the page so that you can view the source code.

Declaring the `DataGrid` Control

The example shown in Figures 1.2 and 1.3 contains a `DataGrid` control that is declared within the page in the usual way, but it uses templates for several of the columns. Listing 1.2 shows the outline declaration of the `DataGrid` control, with the contents of the `<Columns>` element removed. We'll look at the `<Columns>` element in the following section.

LISTING 1.2 The Outline Declaration of the `DataGrid` Control

```
<form runat="server">

<asp:DataGrid id="dgr1" runat="server"
     Font-Size="10" Font-Name="Tahoma,Arial,Helvetica,sans-serif"
     BorderStyle="None" BorderWidth="1px" BorderColor="#deba84"
     BackColor="#DEBA84" CellPadding="5" CellSpacing="1"
     DataKeyField="ProductID"
     OnEditCommand="DoItemEdit"
     OnUpdateCommand="DoItemUpdate"
     OnCancelCommand="DoItemCancel"
     OnItemDataBound="BindRowData"
     AutoGenerateColumns="False">
  <HeaderStyle Font-Bold="True" ForeColor="#ffffff"
               BackColor="#b50055" />
  <ItemStyle BackColor="#FFF7E7" VerticalAlign="Top" />
  <AlternatingItemStyle BackColor="#ffffc0" />
```

LISTING 1.2 Continued

```
  <Columns>
    ... column declarations here ...
  </Columns>

</asp:DataGrid>

...

<asp:ValidationSummary id="valSummary" runat="server"
     DisplayMode="BulletList"
     HeaderText="<b>The following errors were detected:</b>" />

<asp:Button Text="Kill Session" id="btnKill"
            OnClick="KillSession" runat="server" />  
<asp:Label id="lblErr" EnableViewState="False" runat="server" />

</form>
```

Notice that Listing 1.2 specifies the name of the primary key column in the source for the data rows (`ProductID`) and declares the names of the event handlers that will handle the `EditCommand`, `UpdateCommand`, and `CancelCommand` events that occur when the user edits the values in a row. The code also declares a handler for the `ItemDataBound` event that is raised automatically as the control is being bound to its data source; this is where you populate and set the selected index of the nested list controls in a row that is in edit mode.

Listing 1.2 also includes the `AutoGenerateColumns="False"` attribute because you will be creating the columns for the `DataGrid` control yourself. You must do this to incorporate the nested list controls and the validation controls that you want to include in this example.

Following the declaration of the `DataGrid` control is the `ValidationSummary` control, where a list of any validation errors is displayed when the user attempts to submit updates to the row values. Then comes the Kill Session button and the `Label` control that displays errors or help messages (such as hints on how to force a validation error to occur).

The Product Key, Name, and Supplier Columns

Listing 1.3 shows the start of the `<Columns>` element that is omitted from Listing 1.2. The first two columns (the product ID and product name) in the `data grid` cannot be edited, so you use `BoundColumn` elements for these and include the `ReadOnly="True"` attribute. The inclusion of this attribute means that when a row is placed in edit mode, the column will not display the value in a text box.

LISTING 1.3 The Declaration of the First Three Columns in the `DataGrid` Control

```
...
<Columns>

  <asp:BoundColumn DataField="ProductID" HeaderText="ID"
       HeaderStyle-HorizontalAlign="Center" ReadOnly="True" />

  <asp:BoundColumn DataField="ProductName" HeaderText="Product"
       HeaderStyle-HorizontalAlign="Center" ReadOnly="True" />

  <asp:TemplateColumn HeaderText="Supplier"
       HeaderStyle-HorizontalAlign="Center" >
    <ItemTemplate>
      <%# Container.DataItem("Supplier") %>
    </ItemTemplate>
    <EditItemTemplate>
      <asp:DropDownList id="lstSupplier" runat="server" />
      <asp:CompareValidator id="valSupplier" runat="server"
           ControlToValidate="lstSupplier"
           ControlToCompare="lstCategory"
           Operator="NotEqual"
           ErrorMessage="This combination ... is not valid"
           Display="Dynamic" Text="*" />
    </EditItemTemplate>
  </asp:TemplateColumn>
...
```

The supplier name can be edited, but rather than use a `BoundColumn` element (without the `ReadOnly="True"` attribute), you can use the standard technique of declaring an `<ItemTemplate>` section and an `<EditItemTemplate>` section directly. This gives you more control over the content of the column in both normal and edit modes, although you are actually just inserting the column value when in normal mode.

In edit mode, however, you want to display a list of suppliers from the database and select the one that corresponds to the value of this column in the current row. So you declare a `DropDownList` control here. It will be populated, and the appropriate value selected, at runtime by code that runs in response to the `ItemDataBound` event (you'll see this later in the chapter).

You also want to validate the user's selection for the column value if it is changed. The user can't select an "empty" or "null" value because the `DropDownList` will contain only valid supplier names. (If you'd used a `TextBox` control here, adding a `RequiredFieldValidator` control would be the appropriate way to ensure that it is not empty when the page is submitted.)

However, to simulate validation within a row, you can add a couple artificial constraints to the process. You can use a `CompareValidator` control to ensure that the `Value` property of the item selected in the `DropDownList` control (which will be the numeric key of the supplier) is *not* the same as the Value property selected in the RadioButtonList control that displays the category names. (Again, the Value property of each radio button is the numeric key for that category.)

Figure 1.4 shows the DropDownList and RadioButtonList controls within the row that is currently in edit mode. It turns out that (as suggested by the hint at the bottom of the page) the supplier Grandma Kelly's Homestead and the category Confections have the same row key values. Therefore, as you can see in Figure 1.4, the validation control displays its content (an asterisk) when this combination of values is selected.

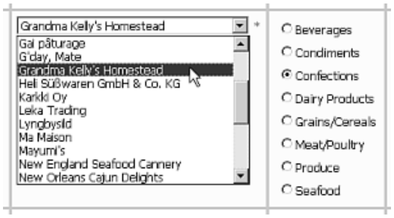


FIGURE 1.4 The DropDownList and RadioButtonList controls within the row that is in edit mode.

The Category and Price Columns

The next two column declarations within the <Columns> element are for the Category and Price columns—shown in full in Listing 1.4. Again, both are TemplateColumn controls, each with an <ItemTemplate> element and an <EditItemTemplate> element that defines the content in normal and edit modes. In normal mode, the column values are displayed directly, although in the case of the Price column you format the value to two fixed decimal places and prefix it with a $ character.

LISTING 1.4 The Declaration of the Category and Price Columns in the DataGrid Control

```
...
<asp:TemplateColumn HeaderText="Category">
  <ItemTemplate>
    <%# Container.DataItem("Category") %>
  </ItemTemplate>
  <EditItemTemplate>
    <asp:RadioButtonList id="lstCategory" runat="server" />
    <asp:CompareValidator id="valCategory" runat="server"
         ControlToValidate="lstCategory"
         Operator="NotEqual"
         ValueToCompare="5"
         ErrorMessage="This category is no longer available"
         Display="Dynamic" Text="*" />
  </EditItemTemplate>
</asp:TemplateColumn>

<asp:TemplateColumn HeaderText="Price"
     HeaderStyle-HorizontalAlign="Center"
     ItemStyle-HorizontalAlign="Right">
```

LISTING 1.4 Continued

```
  <ItemTemplate>
    <%# DataBinder.Eval(Container.DataItem, _
                        "UnitPrice", "${0:F2}") %>
  </ItemTemplate>
  <EditItemTemplate>
    $<asp:TextBox Columns="3" id="txtPrice" runat="server"
          Text='<%# DataBinder.Eval(Container.DataItem, _
                    "UnitPrice", "{0:F2}") %>' />
    <asp:RequiredFieldValidator id="valPrice1" runat="server"
         ControlToValidate="txtPrice"
         ErrorMessage="You must enter a price"
         Display="Dynamic" Text="*" />
    <asp:RangeValidator id="valPrice2" runat="server"
         ControlToValidate="txtPrice"
         MaximumValue="999.99"
         MinimumValue="0.99"
         ErrorMessage="Price must be between $0.99 & $999.99"
         Display="Dynamic" Text="*" />
    <asp:CustomValidator id="valPrice3" runat="server"
         ControlToValidate="txtPrice"
         ClientValidationFunction="ClientValidatePrice"
         OnServerValidate="ServerValidatePrice"
         ErrorMessage="The price must end with 0, 5 or 9"
         Display="Dynamic" Text="*" />
  </EditItemTemplate>
</asp:TemplateColumn>
...
```

BEST PRACTICE

Displaying the Correct Currency Symbol

When you are displaying currency values, it's tempting to format them using the standard currency symbols (for example, `{0:C}`). However, bear in mind that the actual currency symbol displayed depends on the locale of the server, so the application may display a different currency symbol if installed on a machine that is set to a different locale. While this is perhaps unlikely in the United States, it is more likely to be a concern in areas that support more than one "local" locale (pardon the play on words). Specifying fixed format to two decimal places and declaring the currency character explicitly protects against this happening.

The `<EditItemTemplate>` element for the Category column contains the `RadioButtonList` control shown in Figure 1.5, plus a single validation control that (in this example) prevents the category with key value 5 (Grains/Cereals) from being selected.

However, there are three validation controls attached to the text box in which the value of the Price column is displayed. A `RequiredFieldValidator` control prevents the text box from being left empty, and a `RangeValidator` control ensures that the entered value is a valid numeric value between 0.99 and 999.99. Then, to add an extra twist, a `CustomValidator` control is added as well. This uses both client-side and server-side functions (which we'll discuss soon) to implement good supermarket pricing practice by forcing the value to end with a 0, 5, or 9.

Figure 1.5 shows three rows of the `DataGrid` control, the last of which is in edit mode. You can see the way that the Price column is formatted with the currency symbol in normal mode. In edit mode, the value appears in the text box, and the currency symbol appears outside it.

FIGURE 1.5 The Price, Discontinued, and EditCommand columns in the `DataGrid` control.

The Discontinued and EditCommand Columns

Figure 1.5 shows the two remaining columns in the `DataGrid` control: the Discontinued and the EditCommand columns. The Discontinued column displays a "tick" image for normal-mode rows when that product is discontinued and a check box that allows the user to change the status when the row in is edit mode. In the final column, the EditCommand column, you can see that every row displays the Edit link when in normal mode, and the row that is in edit mode displays the Update and Cancel links.

Listing 1.5 shows the declaration of these two columns. The Discontinued column contains an `<ItemTemplate>` section that declares an ASP.NET `Image` control. Both the `AlternateText` and `ImageUrl` properties of the `Image` control are bound to the value in this column of the current row. If you hover the mouse pointer over the image in a normal-mode row, you see the ToolTip generated by the `AlternateText` property, which contains either `"True"` or `"False"`. ASP.NET formats values from a `Boolean` column like this when the general (`"G"`) format is specified.

Likewise, the `ImageUrl` property is automatically set to one of two values—`"True.gif"` or `"False.gif"`—during the data binding process, depending on the value in this row. This means that the appropriate one of the two images provided in the images folder within the root of the application will be displayed. (`False.gif` is just an empty transparent image.)

The `<EditItemTemplate>` element for the Discontinued column is simple compared to the `<ItemTemplate>` element. It contains just the declaration of the `CheckBox` control that is bound to the values in this column.

The `EditCommandColumn` control, as shown in Listing 1.5, creates the Edit, Update, and Cancel links shown in Figure 1.5. You simply specify an `EditCommandColumn` control and set the text values for the links, and it will automatically generate the appropriate links, depending on which mode the row is in.

LISTING 1.5 The Declaration of the Discontinued and EditCommand Columns

```
...
  <asp:TemplateColumn HeaderText="Discontinued"
       ItemStyle-HorizontalAlign="Center">
    <ItemTemplate>
      <asp:Image Width="12" Height="12" runat="server"
        AlternateText='<%# DataBinder.Eval(Container.DataItem,_
                           "Discontinued", "{0:G}") %>'
        ImageUrl='<%# DataBinder.Eval(Container.DataItem, _
                  "Discontinued", "~/images/{0:G}.gif") %>' />
    </ItemTemplate>
    <EditItemTemplate>
      <asp:CheckBox id="chkDiscontinued" runat="server"
         Checked='<%# Container.DataItem("Discontinued") %>' />
    </EditItemTemplate>
  </asp:TemplateColumn>

  <asp:EditCommandColumn EditText="Edit"
       CancelText="Cancel" UpdateText="Update" />

</Columns>
...
```

The Custom Validation Functions

Having looked at the declarations of the controls within the page, you can now look at the relevant features of the code that make it all work. The first thing to consider is the two functions required for the `CustomValidator` control that is attached to the text box in the Price column.

These functions look pretty similar to the ones discussed earlier in this chapter, in the section "Validating a `CheckBoxList` Control," but this time they enforce a different rule. They recognize the value as being valid only if it ends with a 0, 5, or 9 (see Listing 1.6). In each case, the current value of the control to which the validators are attached is available as the `Value` property of the object passed to the function as the second parameter.

LISTING 1.6 The Client-Side and Server-Side Custom Validation Functions

```
<script language="JavaScript">
<!--
// client-side validation function for CustomValidator
function ClientValidatePrice(source, args) {
  var bValid = false;
  var sValue = args.Value.toString();
  var sLast = sValue.substring(sValue.length - 1, sValue.length)
  if ('059'.indexOf(sLast) != -1)
    bValid = true;
```

LISTING 1.6 Continued

```
  args.IsValid = bValid;
  return;
}
//-->
</script>
...
<script runat="server">
Sub ServerValidatePrice(sender As Object, _
                        e As ServerValidateEventArgs)

  Dim bValid As Boolean = False
  Dim sValue As String = e.Value
  If  sValue.Length > 0 Then
    Dim sLast As String = sValue.Substring(sValue.Length - 1)
    If "059".IndexOf(sLast) <> -1 Then
      bValid = True
    End If
  End If
  e.IsValid = bValid

End Sub
...
</script>
```

Handling the `Page_Load` Event

As with almost all pages that use a `DataGrid` control, you need to handle the `Page_Load` event to populate the control the first time the page is loaded (see Listing 1.7). After that, the values the control displays are held in the viewstate of the page. If you try to repopulate the `DataGrid` control on every postback, it won't be possible to properly set the mode (normal or edit) or access the updated values.

LISTING 1.7 The `Page_Load` Event Handler and `BindDataGrid` Routine

```
' page level variable to hold a DataSet
Dim oDS As DataSet

Sub Page_Load()
  If Not Page.IsPostback Then
    BindDataGrid()
  End If
End Sub

...
```

LISTING 1.7 Continued

```
Sub BindDataGrid()

  ' try and get DataSet from current user's Session
  oDS = CType(Session("inx11vdgr"), DataSet)
  If oDS Is Nothing Then
    lblErr.Text &= "Loaded DataSet from the database<br />"

    ' declare SQL statements and use them to fill the three tables
    ' code not shown here  -  see downloadable samples for details
    ' ...

    ' save DataSet in current user's Session
    Session("inx11vdgr") = oDS

  End If

  ' bind DataGrid to Products table
  dgr1.DataSource = oDS
  dgr1.DataMember = "Products"
  dgr1.DataBind()

End Sub
```

Note that you use a page-level variable to store a reference to the `DataGrid` control so that it is available in all the routines in the page that may need to access it. The `BindDataGrid` routine shown in the `Page_Load` event handler is also shown in Listing 1.7. The bulk of the data access code has been removed from Listing 1.7 because there is nothing special about it; it just creates the three SQL statements to extract the data from the Northwind database and push it into three tables in the `data set`. You can download the sample code or use the [view source] link in the online version (see `www.daveandal.net/books/6744/`) to see all the code.

The important point about the `BindDataGrid` routine is the way that it caches the `DataSet` object in the user's ASP.NET session in between page loads. It looks for a session variable named `inx11vdgr` and attempts to convert that into a `DataSet` instance. If this is successful, the `DataSet` object is reused, without requiring a trip back to the database to be filled.

However, if it is not found in the user's session, it is filled and then stored there, ready for the next postback and page load. This means that the page will automatically cache the `DataSet` object where ASP.NET sessions are supported and gracefully fall back to re-creating and filling it on each postback if sessions are not supported.

The final step is to bind the `Products` table in the `DataSet` object to the `DataGrid` control and call the `DataBind` method to start the process of displaying the values.

Handling the `ItemDataBound` Event

When you declared the `DataGrid` control in the page, you specified that the `ItemDataBound` event should cause the routine named `BindRowData` to execute. You added this `OnItemDataBound` attribute to the declaration of the `DataGrid` control:

```
OnItemDataBound="BindRowData"
```

So, each time a row in the `DataGrid` control is bound to its data source row, the `BindRowData` routine (shown in Listing 1.8) will be executed. In this routine, you have to populate the `DropDownList` and `RadioButtonList` controls declared in the `<EditItemTemplate>` sections of the `DataGrid` control, but you only have to do this for the row that is currently being edited. So you check the `ItemType` property of the `DataGridItemEventArgs` instance passed to the event handler first.

If this row is in edit mode, you can get a reference to each control in turn and then bind it to the appropriate table in the `DataSet` object (`Suppliers` or `Categories`). You also have to select the correct value in these two lists, depending on the value currently in the relevant column of this row. You can extract the values of the columns in the current row from the `Item.DataItem` property of the `DataGridItemEventArgs` instance, specifying the column name you require. The `Item.DataItem` property is a reference to the `DataRowView` instance from the source data table that is being used to populate this row of the `DataGrid` control.

LISTING 1.8 The `BindRowData` Handler for the `ItemDataBound` Event

```
Sub BindRowData(sender As Object, e As DataGridItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  ' only process it if it's the row in Edit mode
  If oType = ListItemType.EditItem Then

    ' get a reference to DropDownList control in the edit row
    Dim oSupplier As DropDownList _
      = CType(e.Item.FindControl("lstSupplier"), DropDownList)

    ' bind it to the Suppliers table
    oSupplier.DataSource = oDS
    oSupplier.DataMember = "Suppliers"
    oSupplier.DataTextField = "CompanyName"
    oSupplier.DataValueField = "SupplierID"
    oSupplier.DataBind()

    ' set the current selection to the row value
    oSupplier.SelectedValue _
      = e.Item.DataItem("SupplierID").ToString()
```

LISTING 1.8 Continued

```
    'repeat for Categories RadioButtonList
    Dim oCategory As RadioButtonList _
      = CType(e.Item.FindControl("lstCategory"), RadioButtonList)

    ' bind it to the Categories table
    oCategory.DataSource = oDS
    oCategory.DataMember = "Categories"
    oCategory.DataTextField = "CategoryName"
    oCategory.DataValueField = "CategoryID"
    oCategory.DataBind()
    oCategory.SelectedValue _
      = e.Item.DataItem("CategoryID").ToString()

  End If

End Sub
```

At this point, the page is complete, as far as displaying the data is concerned. The combination of calling the `DataBind` method against the table bound to the `DataGrid` control and the intervention in the `ItemDataBound` event to populate and select the appropriate value for the nested list controls in any edit row (although there will be no row in edit mode when the page first loads, of course) means that the user will see what is shown in Figure 1.2.

BEST PRACTICE

Selecting the Current Value in a Nested List Control

When you nest list controls inside another list control, such as a `DataGrid`, `DataList`, or `Repeater` control, it's important to select the appropriate value in each nested list control to match the value already in the row. If you don't, when the user submits the page, the value in the row will be changed—even though the user hasn't selected a different value in the nested list control.

Editing and Updating the Data

You need to consider how to handle edits to the data in the `DataGrid` control. In fact, the process used in this example is basically the "standard" way that you see in documentation for the `DataGrid` control: When the user clicks the Edit link, that row is displayed in edit mode, as shown in earlier Figures 1.3 and 1.5. Listing 1.9 shows the `DoItemEdit` event handler that is executed when any of the Edit links are clicked (recall that you specified the event handlers in the declaration of the `DataGrid` control; for example, `OnEditCommand="DoItemEdit"`).

LISTING 1.9 The `DoItemEdit` Handler for the `EditCommand` Event

```
Sub DoItemEdit(sender As Object, e As DataGridCommandEventArgs)

  ' set the EditItemIndex of the grid to this item's index
  dgr1.EditItemIndex = e.Item.ItemIndex

  ' bind grid to display newly-loaded data
  BindDataGrid()

  ' display the validation error hints
  helptext.Visible = True

End Sub
```

In the `DoItemEdit` routine, you just set the `EditItemIndex` property of the `DataGrid` control to specify which row should be displayed in edit mode, and then you call the `BindDataGrid` routine to display all the rows in their correct modes. You also display the hints (for causing a validation error) shown at the bottom of the page, by setting the `Visible` property of the `<div>` element that contains them to `True`.

Handling the `UpdateCommand` Event

After the user changes the values in the row that is in edit mode, he or she clicks the Update link that is available in that row to push the changes back into the database. The Update link causes the `DoItemUpdate` event handler to be executed, which builds up the SQL statement required to update the original table in the database. You can see this in Listing 1.10. The code simply references the four controls in this row that allow editing, and it uses their values to create the SQL statement.

LISTING 1.10 The `DoItemUpdate` Handler and `ExecuteSQLStatement` Routine for the `UpdateCommand` Event

```
Sub DoItemUpdate(sender As Object, e As DataGridCommandEventArgs)

  If Page.IsValid Then

    ' remove existing DataSet from current user's Session
    Session("inx11vdgr") = Nothing

    ' get a reference to controls in the edit row
    Dim oSupplier As DropDownList _
      = CType(e.Item.FindControl("lstSupplier"), DropDownList)
    Dim oCategory As RadioButtonList _
      = CType(e.Item.FindControl("lstCategory"), RadioButtonList)
    Dim oPrice As TextBox _
      = CType(e.Item.FindControl("txtPrice"), TextBox)
```

LISTING 1.10 Continued

```
    Dim oDisc As CheckBox _
      = CType(e.Item.FindControl("chkDiscontinued"), CheckBox)

    ' create a suitable SQL statement and execute it
    Dim sSQL As String
    sSQL = "UPDATE Products SET SupplierID=" & oSupplier.SelectedValue & ", " _
         & "CategoryID=" & oCategory.SelectedValue & ", " _
         & "UnitPrice=" & oPrice.Text & ", " _
         & "Discontinued=" & CType(oDisc.Checked, Int16) & " " _
         & "WHERE ProductID=" & dgr1.DataKeys(e.Item.ItemIndex)
    ExecuteSQLStatement(sSQL)

    ' set EditItemIndex of grid to -1 to switch out of Edit mode
    dgr1.EditItemIndex = -1

    ' bind grid to display row in new mode
    BindDataGrid()

    ' hide the validation error hints
    helptext.Visible = False

  End If

End Sub

...

Sub ExecuteSQLStatement(sSQL)

  ' execute SQL statement against the original data source
  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindOleDbConnectString")
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSQL, oConnect)
    If oCommand.ExecuteNonQuery() <> 1 Then
      lblErr.Text &= "ERROR: Could not update the selected row<br />"
    End If
    oConnect.Close()

  Catch oErr As Exception
```

LISTING 1.10 Continued

```
    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text &= "ERROR: " & oErr.Message & "<br />"

  End Try

End Sub
```

BEST PRACTICE

Using a Stored Procedure to Update the Data Store

This section, of course, provides a demonstration of the use of the ASP.NET controls and is *not* the ideal real-world approach. You should consider using a stored procedure to push the changes back into the database, or at least use a parameterized SQL statement to prevent users from attacking the database by entering malicious text into edit controls. See Chapter 10, "Relational Data-Handling Techniques," for more details.

Also notice that this routine first removes any existing `DataSet` instance from the user's ASP.NET session so that the new data will be fetched from the database when the `BindDataGrid` routine is called toward the end of this routine. However, before calling this routine to repopulate the grid, you set the `EditItemIndex` property to `0` so that the current row will go out of edit mode. You end by hiding the hint text again because it applies only when there is a row in edit mode.

Listing 1.10 also shows the `ExecuteSQLStatement` routine that the `DoItemUpdate` routine uses to actually execute the SQL statement—with the `ExecuteNonQuery` method of a `Command` instance. If an error occurs, a message is displayed in the page. However, because you've validated all your values first, you should be protected from simple data update errors!

Handling the `CancelCommand` Event

The only other link in the `DataGrid` control that the user might click (available only for the row that is currently being edited) is the Cancel link. The event handler that executes when this occurs only has to reset the `EditItemIndex` property of the `DataGrid` control, repopulate the control (using the cached `DataSet` object, if one is available), and hide the hint text (see Listing 1.11).

LISTING 1.11 The DoItemCancel Handler for the CancelCommand Event

```
Sub DoItemCancel(sender As Object, e As DataGridCommandEventArgs)

  ' set EditItemIndex of grid to -1 to switch out of Edit mode
  dgr1.EditItemIndex = -1

  ' bind grid to display row in new mode
  BindDataGrid()

  ' hide the validation error hints
  helptext.Visible = False

End Sub
```

Removing Existing Session Data

The final section of code in the DataGrid control validation sample page runs when the Kill Session button is clicked. This button is outside the DataGrid control, so a standard event handler is used. In it, you just remove the existing DataSet object from the user's ASP.NET session and then call the BindDataGrid routine to re-create and fill it (see Listing 1.12).

LISTING 1.12 The KillSession Handler for the Click Event of the Button Control

```
Sub KillSession(sender As Object, e As EventArgs)

  ' remove existing DataSet from current user's Session
  Session("inx11vdgr") = Nothing

  ' bind grid to display newly loaded data
  BindDataGrid()

End Sub
```

Taking Control of Content Layout in a DataGrid Control

The majority of the previous sections of this chapter concentrate on the use of the ASP.NET validation controls, especially within a DataGrid control. However, we have also looked at a few other features of the DataGrid control along the way, including the use of nested list controls within a DataGrid control.

We'll come back to the issues of using nested list controls in a lot more detail in Chapter 4. However, there are a couple other interesting topics we cover next in this chapter that also use

DataGrid controls. These techniques are not concerned with nested controls but with how you can get more from the ASP.NET server controls when you build your applications and Web pages.

The following sections look at an example that moves away from the standard approach and appearance of the DataGrid control to provide something that might be more intuitive or useful to users. And it's no more complicated to implement than the "ordinary" approach. It should help remind you that you can often find solutions or develop new techniques just by thinking laterally—and by becoming familiar with all the properties, methods, and events of the ASP.NET server controls.

Controlling the Width of Columns in a DataGrid Control

In Figure 1.6, the Precis column contains a lot of text. Normally, the table that a DataGrid control creates would expand to fill the width of the browser window, making the Precis column a lot wider (and perhaps making the content harder to read). However, for this example, we've specified the width of each of the columns in the declaration of the DataGrid control so that they no longer expand to fill the available width.

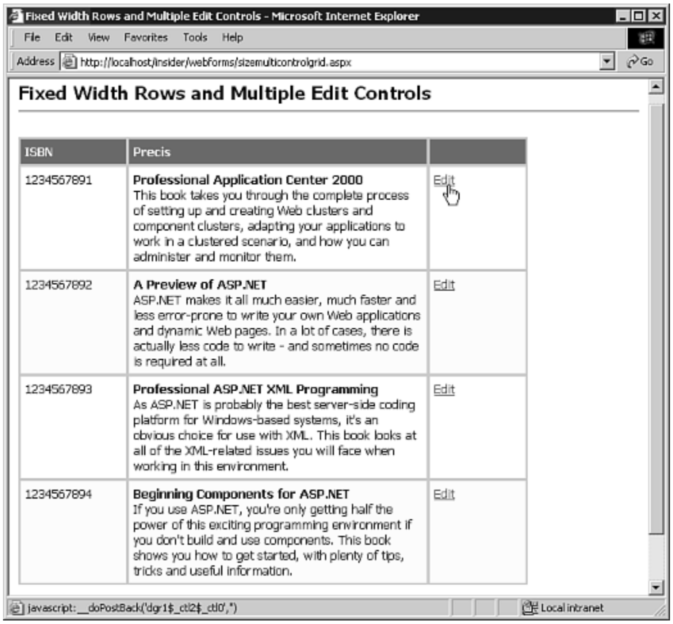


FIGURE 1.6
Controlling the width of columns in a DataGrid control.

To specify the width of a column, you simply add the ItemStyle-Width attribute to the declaration of the column:

```
ItemStyle-Width="300px"
```

In Internet Explorer, this causes the DataGrid control to add the width style selector to the opening <td> tag of the table cell that represents this column:

```
<td style="width:300px;">...</td>
```

However, like most of the ASP.NET Web Forms controls, the DataGrid control generates different output for "up-level" clients than for "down-level" clients. Only Internet Explorer 5.x and higher are classified as up-level, even though most other modern browsers understand CSS. But thankfully, the DataGrid control is clever enough to cope with this by adding the width attribute to the opening <td> tags in other browsers:

```
<td width="300">...</td>
```

It doesn't matter what type of column you're working with—BoundColumn, TemplateColumn, HyperlinkColumn, EditCommandColumn, or other type of column. The only limitation seems to be that the width cannot be made less than the longest "unbreakable" section of text or other content. In other words, the column won't shrink to less than the length of the longest unhyphenated word or the width of an image.

BEST PRACTICE

Setting the Width of All the Columns

You should set the width of all the columns in a DataGrid control which contain text that might disturb the layout you want for your page. The browser allocates the column widths dynamically, based on the width of the content, restrictions applied to each <td> element, and the width of the browser window. So just limiting the width of one column allows the other columns to grow to fill the available width.

Using Multiple Edit Controls in a DataGrid Control Column

The example shown in Figure 1.6 contains an Edit link in each row. Clicking the Edit link changes that row into edit mode, and the content of the Precis column is displayed in two TextBox controls, as shown in Figure 1.7.

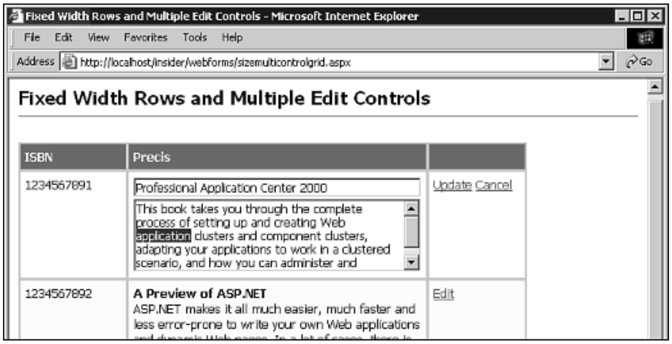


FIGURE 1.7
Editing the contents of the Precis column in two TextBox controls.

In fact, the content of the Precis column shown in the `DataGrid` control comes from two columns in the original data source. We've used code (not listed here) in the page to generate an ADO.NET `DataTable` instance that contains three columns: ISBN, Title, and Precis. Listing 1.13 shows the complete declaration of the `TemplateColumn` element that implements the Precis column in the `DataGrid` control. We use the `ItemStyle-Width` attribute, and we also add the `ItemStyle-VerticalAlign="Top"` attribute to every column to get the layout we want.

The `<ItemTemplate>` element simply specifies the contents of the two columns, Title and Precis, with the Title column in bold and followed by a `<br />` element. The `<EditItemTemplate>` element contains two `TextBox` controls, again separated by a `<br />` element. The second one is set to multiline mode, and we specify the number of rows (lines). This means that the layout of the content in the Precis column doesn't change when the user switches the row into edit mode, but the two separate values are available for editing.

LISTING 1.13 The `TemplateColumn` Declaration for the Precis Column

```
<asp:TemplateColumn HeaderText="Precis"
     HeaderStyle-HorizontalAlign="Left"
     ItemStyle-Width="300px"
     ItemStyle-VerticalAlign="Top">
  <ItemTemplate>
    <b><%# Container.DataItem("Title") %></b><br />
    <%# Container.DataItem("Precis") %>
  </ItemTemplate>
  <EditItemTemplate>
    <asp:Textbox id="txtTitle" runat="server"
         Width="300" Style="width:300px"
         Text='<%# Container.DataItem("Title") %>' /><br />
    <asp:Textbox id="txtPrecis" runat="server"
         TextMode="MultiLine" Rows="5"
         Width="300" Style="width:300px"
         Text='<%# Container.DataItem("Precis") %>' />
  </EditItemTemplate>
</asp:TemplateColumn>
```

The sample page contains code to handle the various edit and update events of the `DataGrid` control, exactly as is done in earlier examples, but it doesn't actually persist the changes. This is because the data is generated by code within the page and not taken from a database. However, the code in the page does generate a sample SQL statement that could be executed to update a database, and it displays the SQL statement at the bottom of the page when the Update link is clicked. (Figure 1.8 provides a combined before-and-after view of the results.)

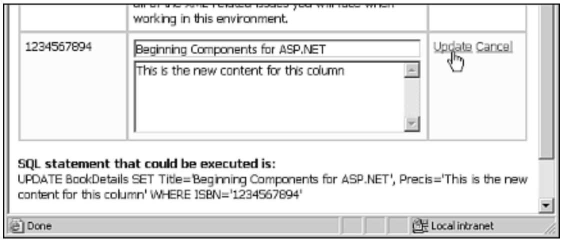


FIGURE 1.8
Displaying a SQL statement after editing the contents of the Precis column.

Just because there are two controls in the same column of each row doesn't change the way that you can access each control. The `FindControl` method, which is used in previous examples to get a reference to an edit control in the current row, works just the same:

```
Dim oTitle As TextBox _
    = CType(e.Item.FindControl("txtTitle"), TextBox)
Dim oPrecis As TextBox _
    = CType(e.Item.FindControl("txtPrecis"), TextBox)
```

You can use the [view source] link at the bottom of the sample page in the online version (see `www.daveandal.net/books/6744/`) to see all the code if you wish.

Controlling the Width of Edit Controls in a `DataGrid` Control

One more issue arises that you need to be aware of when specifying column widths in a `DataGrid` control. Although the `DataGrid` control is clever enough to handle column widths for both up-level and down-level clients, the same can't be said for many other Web Forms controls.

For example, the `TextBox` control has both a `Width` property and a `Columns` property. The value of the `Columns` property is used to set the `size` attribute of the `<input type="text">` element that is generated, which vaguely controls the width, based on the number of characters and the font size and style settings. However, the value of the `Width` property is used to generate a style attribute that accurately determines the width of the control, regardless of the font size and style settings.

Unfortunately, this style attribute is only output to up-level clients, so in even the latest of the non-Internet Explorer browsers (the down-level clients), the text box assumes some default width of its own. You need to declaratively add the `style` attribute you want. The `EditItemTemplate` section that follows specifies both the `Width` and `Style` properties of the two `TextBox` controls:

```
<EditItemTemplate>
  <asp:Textbox id="txtTitle" runat="server"
       Width="300" Style="width:300px"
       Text='<%# Container.DataItem("Title") %>' /><br />
  <asp:Textbox id="txtPrecis" runat="server"
       TextMode="MultiLine" Rows="5"
       Width="300" Style="width:300px"
       Text='<%# Container.DataItem("Precis") %>' />
</EditItemTemplate>
```

Now the page looks and works the same in most of the new browsers and in some not-so-new ones as well (for example, Netscape Navigator 4.5).

Providing Scrollable Content in a `DataGrid` Control

Limiting the width of columns in a `DataGrid` control is a useful way to exert extra control over the appearance of pages. However, if there is a lot of content in a cell, the `DataGrid` control expands vertically to accommodate all of it. You can easily prevent this by placing the content into scrollable containers in each column.

Figure 1.9 shows the sample page. You can see that in the Precis column, the rows that are in normal mode display the content inside a scrollable container. In edit mode, a multiline `TextBox` control allows the content to be edited just as in the previous example (although the controls that display the title have been removed from this example).

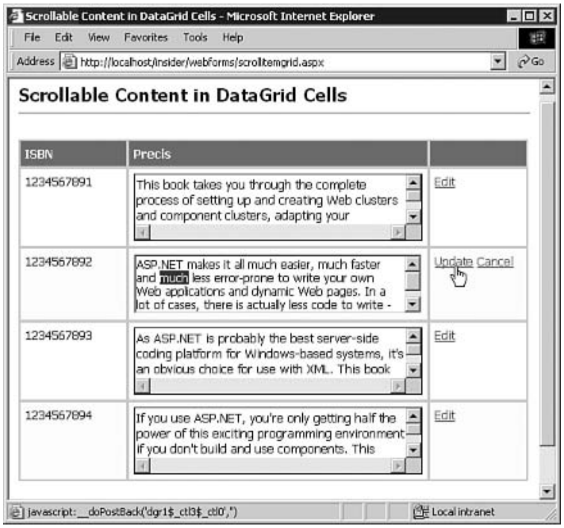


FIGURE 1.9
Using scrollable containers within a `DataGrid` control.

To enable scrolling for the contents of a column, you need to enclose it in a suitable container element and add the appropriate style selectors to that container element. The obvious choice of container element is a `<div>` element or an ASP.NET `Panel` control. If you don't intend to reference the content of the column in normal mode, the container doesn't have to be a server control (it will not contain the `runat="server"` attribute). In this case, it can be an ordinary HTML element and therefore not part of the ASP.NET control tree when the page is being generated on the server side.

> **When to Access the Controls**
>
> **Usually the only reason you would need to reference the controls in a row when it is not in edit mode is if you are modifying the content as the `DataGrid` control is generating its output (for example, during the `ItemDataBound` event).**

There is no need to reference the content of the Precis column in normal mode in this example, so we can use a `<div>` element. If you decide to use a server control, beware of the ASP.NET `Panel` control. Although it generates a `<div>` element in Internet Explorer 5.x and higher, it generates an HTML `<table>` element in all other browsers. The `<div>` element was not part of the original HTML recommendations, although almost all browsers in use today do recognize it.

Using the `HtmlGenericControl` Class

Instead of using the `Panel` control, you might prefer to generate a server-side `<div>` control explicitly in order to avoid the issue of the `Panel` control changing its output depending on the browser:

```
<div id="MyDiv" runat="server">Content goes here</div>
```

You implement this by using the ASP.NET `HtmlGenericControl` class, which is used for any element that contains the `runat="server"` attribute but is not implemented by a specific control type within the .NET Framework. The `HtmlGenericControl` class (located in the `System.Web.UI.HtmlControls` namespace) is descended from `Control` and `HtmlControl`, and it implements several of the common properties that all server controls provide.

It exposes an `Attributes` collection and a `Controls` collection, together with properties such as `ID`, `Disabled`, `EnableViewState`, `Page`, `Parent`, `Visible`, and `Style`. It also has a `TagName` property that is read/write and that reflects the actual HTML tag that is generated (such as `"DIV"` or `"P"`). (We'll be looking at the `Controls` collection in more detail later in this chapter.)

The only unusual feature is that there is no `Text` or `Value` property. Instead, you read or write the content of the element by using the `InnerHtml` property (which sets or returns all the content between the opening and closing tags of the control, including other elements), or the `InnerText` property (which sets or returns just the text content of the control).

Listing 1.14 shows the `TemplateColumn` declaration for the scrollable content example shown in Figure 1.9. Notice that the `ItemStyle-Width` attribute has been removed from the `TemplateColumn` itself because the container control will be of a fixed size and will restrain the content—so the column will not expand beyond the size of the container element.

You can see the various `style` selectors applied to the `<div>` element. As well as `width` and `height`, this code turns on scrollbars with the `overflow` selector. To give the appearance of a container, a thin black border is added, and the background is changed to white.

LISTING 1.14 The `TemplateColumn` Declaration for the Scrollable Content Version of the Precis Column

```
<asp:TemplateColumn HeaderText="Precis"
     HeaderStyle-HorizontalAlign="Left">
  <ItemTemplate>
    <div style="width:300;height:70;overflow:scroll;
                border:1 solid black;padding:2;
                background-color:white">
      <%# Container.DataItem("Precis") %>
```

LISTING 1.14 Continued

```
    </div>
  </ItemTemplate>
  <EditItemTemplate>
    <asp:Textbox id="txtPrecis" runat="server"
         TextMode="MultiLine" Rows="4"
         Width="300" Style="width:300px"
         Text='<%# Container.DataItem("Precis") %>' />
  </EditItemTemplate>
</asp:TemplateColumn>
```

You might like to experiment with the `CellPadding` and `CellSpacing` properties of the `DataGrid` control, as well as with different values of the `border` style selector, to get a different appearance for the scrollable regions. For example, `"style=border:3 inset"` when `CellPadding` and `CellSpacing` are both zero gives a very compact grid-like effect.

Loading Controls Dynamically at Runtime

When the ASP team at Microsoft was designing ASP.NET, it probably seemed obvious that the way forward was to compile the pages into some kind of executable code. This approach means that there is a distinct separation between the tasks (and the amount of processing work ASP.NET has to do) of generating a page the first time it is executed—when it has to be compiled and the resulting code written to disk—and subsequent executions of the compiled code.

As a result, the way that the structure and content of a page are discovered and created from a file containing declarative definitions and code in `<script runat="server">` sections only affects the "initial hit" performance and not the performance on subsequent requests. Consequently, this has provided a development environment that supports quite complex page creation techniques, such as the use of server controls and user controls, page and control state maintenance, and dynamic creation of a control tree for the page.

In particular, the use of a developer-accessible control tree has made it really easy to use ASP.NET to build pages that, in ASP 3.0 and many other Web development environments, would required complicated `Response.Write` statements, `#include` directives, and other tricks.

Being able to create controls dynamically at runtime, meanwhile, is extremely useful if you don't know beforehand how many instances of a particular control you need on the page. For example, you might need to create a number of text boxes or buttons, depending on the value entered by the user, which could therefore be different each time the page is executed.

The ASP.NET Control Tree

As ASP.NET processes a page, it generates a control tree that contains references to all the server controls on the page. Note that this only includes server controls—basically declarative elements that contain the `runat="server"` attribute. Figure 1.10 shows a conceptual view of a page that contains several server controls, including a server-side `<form>` element that contains many of the controls on the page. Notice also that the `Hyperlink` and `HtmlAnchor` (`<a>` element) controls in this example have child `Image` and `HtmlImage` controls. These represent the typical output that provides clickable images:

```
<a href="http://www.daveandal.net">
  <img src="click.gif" border="0" />
</a>
```

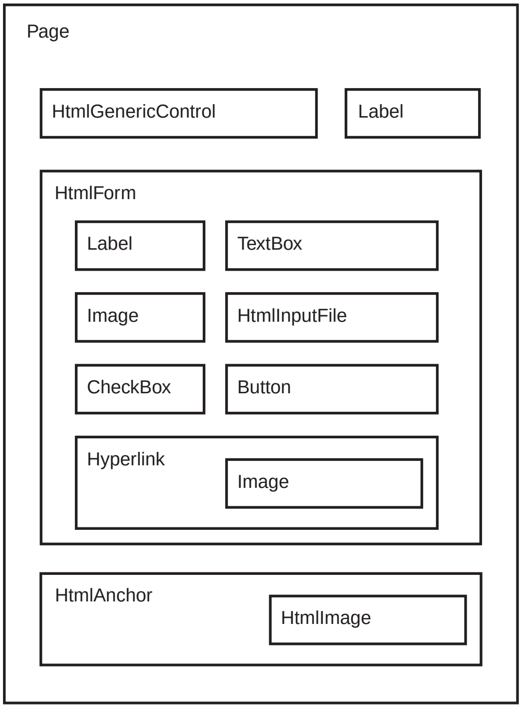


FIGURE 1.10 A conceptual view of an ASP.NET page that contains nested controls.

In more technical terms, the page consists of a *hierarchy* of control instances. Figure 1.11 shows this in tree form. Each object in the tree is a server control that is descended directly or indirectly from `System.Web.UI.Control` and thus exposes a `Controls` property that references a `ControlCollection` instance. Each `ControlCollection` instance is, as you might guess, a collection of references to all the child controls for that control.

You can manipulate the control tree by adding controls to and removing them from these `ControlCollection` instances. When the page is rendered, the control tree is used to build the HTML (or other output) that is sent to the client. Table 1.1 shows the properties and methods of the `ControlCollection` object that are useful when manipulating the control tree.

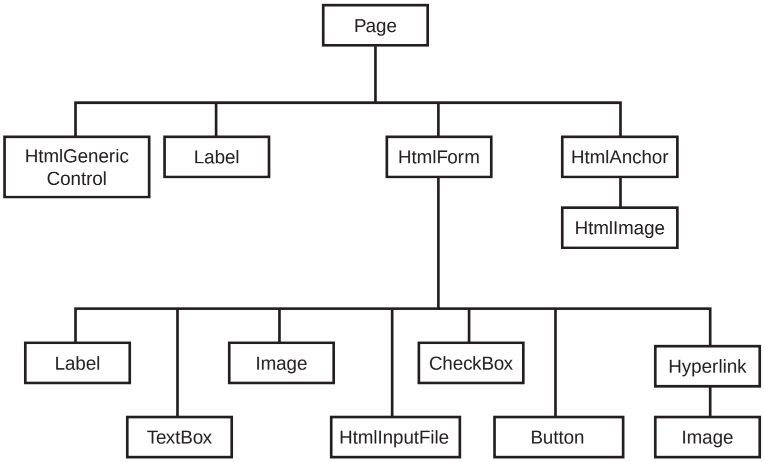


FIGURE 1.11
The ASP.NET control tree for the page shown in Figure 1.10.

TABLE 1.1
The Members of the `ControlCollection` Object for Working with, Adding, and Removing Controls

Property or Method	Description
`Count`	Returns the number of controls in the `ControlCollection` instance.
`Item`	Acts as the indexer for the zero-based `ControlCollection` instance, returning a reference to a control within the collection.
`Add(`*`control`*`)`	Adds the `Control` instance referenced by the *`control`* parameter to the end of the `ControlCollection` instance of this control.
`AddAt(`*`index`*`, `*`control`*`)`	Inserts the `Control` instance referenced by the *`control`* parameter into the `ControlCollection` instance of this control at the specified `Integer` *`index`*.
`Clear()`	Removes all the controls from the `ControlCollection` instance of this control.
`Contains(`*`control`*`)`	Returns a `Boolean` value indicating whether the `Control` instance referenced in the *`control`* parameter is a member of this control's `ControlCollection` instance.
`IndexOf(`*`control`*`)`	Returns the `Integer` index of the `Control` instance referenced in the *`control`* parameter within this control's `ControlCollection` instance.
`Remove(`*`control`*`)`	Removes the `Control` instance referenced in the *`control`* parameter from this control's `ControlCollection` instance.
`RemoveAt(`*`index`*`)`	Removes the `Control` instance at the specified `Integer` *`index`* from this control's `ControlCollection` instance.

Adding Controls to the Control Tree

In most cases, you can just use the `Add` method to add controls to the control tree in the correct order to produce the output you want. This is generally easier than trying to figure out where to insert a control within a collection, although the other methods are useful if you need to do any complex management of the child controls for a particular control.

Probably the easiest way to insert a control into a page at a specific point is to use an ASP.NET `PlaceHolder` control. This generates no output in the page, but it does expose a `ControlCollection` instance to which you can add other controls. When you use this approach, the newly added controls will always appear in the same position in the page, even if you later add or remove controls from elsewhere in the control tree or the parent control's `ControlCollection` instance.

The following code demonstrates the use of the `PlaceHolder` control. In the `Page_Load` event, you just create a new `Hyperlink` control and set the `NavigateUrl` property. Then you create a new `Image` control and specify its `ImageUrl` property. Next, you add the `Image` control to the `ControlCollection` instance of the `Hyperlink` control and add the `Hyperlink` control to the `ControlCollection` instance of the `PlaceHolder` control:

```
<asp:PlaceHolder id="ph1" runat="server" />
...
Sub Page_Load()
  Dim oLink As New Hyperlink()
  oLink.NavigateUrl = "http://www.daveandal.net"
  Dim oImage As New Image()
  oImage.ImageUrl = "~/images/True.gif"
  oLink.Controls.Add(oImage)
  ph1.Controls.Add(oLink)
End Sub
```

When the page is rendered, the following output is generated (ASP.NET automatically adds the `border="0"` attribute):

```
<a href="http://www.daveandal.net">
  <img src="../images/True.gif" border="0" />
</a>
```

The Actual Output Format

The output is not actually indented as shown here, but instead is generated as a single line with no spaces or carriage returns. In this example it is formatted with carriage returns and indented so that you can see the result more clearly.

Creating a `DataGrid` Control Dynamically at Runtime

Compared to the previous, somewhat trivial example, the following example generates a more complex page which contains a `DataGrid` control that supports inline editing. The result is shown in Figure 1.12.

FIGURE 1.12
A dynamically generated `DataGrid` control.

The page contains all the server-side code to handle the events in the DataGrid control, using the same techniques in the first example in this chapter. (We won't be looking at that code here.) With this example, we are interested in the way that the DataGrid control itself is generated. The HTML section of the sample page contains a server-side <form> element, but no other content:

```
<form id="frmMain" runat="server">
  <!-- DataGrid and Label will be dynamically inserted here -->
</form>
```

> **Choosing the Event when Adding Controls**
>
> **Instead of generating the controls during other ASP.NET page events, such as Init or Render, we had most success getting the process to work reliably, especially when wiring up event handlers, by using the Page_Load event. The controls must be generated on *every* postback (not just when the page is first loaded) and in exactly the same order and with the same ID values. Unlike control values, dynamically generated controls are not maintained in the viewstate of the page. However, values are maintained and will be reloaded after the controls have been created and added to the control tree.**

Instead, the DataGrid control and the Label control used to display any data access errors are added to the page dynamically during the Page_Load event. Also, the various events in the DataGrid control are wired to the appropriate event handlers already located in the <script> section of the page.

The previous example uses a PlaceHolder control as the container to which the new controls are added. However, a server-side <form> element works just as well, and in this example you can add the controls directly to the ControlCollection instance of the HtmlForm control that implements the server-side <form> control.

Setting Size and Color Properties Dynamically

You can generate values for some of the properties of Web Forms server controls. Properties that set the color of parts of the output, such as ForeColor and BackColor, accept references to a Color structure. Properties that accept sizes, such as Width and BorderWidth, accept references to a Unit structure. When declaring a server control in the HTML section of the page, you can use the color names or size values directly, as in this example:

```
<HeaderStyle ForeColor="#ffffff" BackColor="#b50055" />
<ItemStyle BorderWidth="1px" />
```

However, to set these properties dynamically, you have to provide an instance of the correct structure classes. Here's an example:

```
oGrid.HeaderStyle.ForeColor = Color.FromName("#ffffff")
oGrid.HeaderStyle.BackColor = Color.FromName("#b50055")
oGrid.ItemStyle.BorderWidth = Unit.Pixel(1)
```

The Unit structure is part of the System.Web.UI.WebControls namespace, so it is available by default in all ASP.NET Web pages. However, the Color structure is defined in the System.Drawing namespace, which is not imported into ASP.NET pages by default. Therefore, you have to include the appropriate `Import` directive in any pages that reference a `Color` structure:

```
<%@Import Namespace="System.Drawing" %>
```

Creating the `DataGrid` Control

Creating the `DataGrid` control itself is not difficult; it just requires quite a lot of repetitive code. You create an instance of a `DataGrid` control, set all the properties, and then you add it to the `Controls` collection of the `<form>` element in the page. You do the same with the `Label` control that will display any data access errors.

Listing 1.15 shows the declaration of two page-level variables that are used to hold references to the new controls (so that they can be accessed in routines other than the `Page_Load` event handler), followed by the start of the `Page_Load` event handler. Here, you create the `DataGrid` control and add all the properties that set the appearance and behavior of the control. You can reduce the amount of code required by taking advantage of the Visual Basic .NET `With` construct.

Notice how you set the properties of objects that are actually children of the `DataGrid` control, such as the `HeaderStyle`, `ItemStyle`, and `AlternatingItemStyle` objects. You can use a nested `With` construct or just reference them by using a period to access the child objects.

LISTING 1.15 Dynamically Generating a `DataGrid` Control

```
Dim oGrid As DataGrid
Dim oLabel As Label

Sub Page_Load()

  ' create a DataGrid control
  oGrid = New DataGrid()

  ' set control properties
  With oGrid
    .id = "dgr1"
    .BorderStyle = BorderStyle.None
    .BorderWidth = Unit.Pixel(0)
    .BackColor = Color.FromName("#deba84")
    .CellPadding = 3
    .CellSpacing = 0
    .DataKeyField = "CustomerID"
    .Width = Unit.Percentage(100)
    .AutoGenerateColumns = False
    With .HeaderStyle
      .Font.Bold = True
      .ForeColor = Color.FromName("#ffffff")
      .BackColor = Color.FromName("#b50055")
    End With
```

LISTING 1.15 Continued

```
  .ItemStyle.BackColor = Color.FromName("#fff7e7")
  .AlternatingItemStyle.BackColor = Color.FromName("#ffffc0")
End With

' create a column for the DataGrid control
' and set properties
Dim oCol1 As New EditCommandColumn()
With oCol1
  .EditText = "Edit"
  .CancelText = "Cancel"
  .UpdateText = "Update"
End With

' add column to DataGrid
oGrid.Columns.Add(oCol1)

' repeat for remaining columns
Dim oCol2 = New BoundColumn()
With oCol2
  .DataField = "CustomerID"
  .HeaderText = "ID"
  .ReadOnly = True
End With
oGrid.Columns.Add(oCol2)

...
' same for CompanyName, City, Country and Phone columns
...
```

Toward the end of Listing 1.15, you can see the columns being added. You create an instance of the appropriate type of column, set the properties, and then add the column to the `Columns` collection. Like the `Controls` property, the `Columns` property of a `DataGrid` control is a collection of references to the columns that make up the `DataGrid` control. Listing 1.15 does not contain the declarations of the Company Name, City, Country, and Phone columns because they are identical to the Customer ID column (except, of course, that they refer to different columns in the source data).

Wiring Up the `DataGrid` Control Events

With the `DataGrid` control complete, you can attach its events to the appropriate event handler routines already present in the page, as shown in Listing 1.16. In Visual Basic .NET you use the `AddHandler` statement, and in C# you just append the event delegates by using the `+=` operator. You can wire up the `EditCommand`, `UpdateCommand`, and `CancelCommand` events, targeting them at the event handlers named `DoItemEdit`, `DoItemUpdate`, and `DoItemCancel`. This achieves the same result as declaring them directly in the page, as in the `DataGrid` control example earlier in this chapter:

```
OnEditCommand="DoItemEdit"
OnUpdateCommand="DoItemUpdate"
OnCancelCommand="DoItemCancel"
```

LISTING 1.16 Wiring Up the Event Handlers, Creating the `Label` Control, and Populating the `DataGrid` Control

```
...
  ' add event handlers to the grid
  AddHandler oGrid.EditCommand, _
      New DataGridCommandEventHandler(AddressOf DoItemEdit)
  AddHandler oGrid.UpdateCommand, _
      New DataGridCommandEventHandler(AddressOf DoItemUpdate)
  AddHandler oGrid.CancelCommand, _
      New DataGridCommandEventHandler(AddressOf DoItemCancel)

  ' create new Label control and set properties
  oLabel = New Label()
  With oLabel
    .id = "lblErr"
    .EnableViewState = False
  End With

  ' add new controls to page as children of <form>
  frmMain.Controls.Add(oGrid)
  frmMain.Controls.Add(oLabel)

  ' only need to databind if it is not a postback
  ' viewstate used to populate dynamically added controls
  If Not Page.IsPostback Then
    oGrid.DataSource = GetCustomers()
    oGrid.DataBind()
  End If

End Sub
```

After attaching the event handlers, you generate a new `Label` control and set its properties. Then you add the `DataGrid` and the `Label` controls to the `ControlCollection` instance of the server-side `<form>` control declared in the page (as shown in Listing 1.16).

Populating the `DataGrid` Control

The final task in this example, shown at the end of Listing 1.16, is to populate the `DataGrid` control. As long as the control tree you generate is the same every time the page is loaded, the values of all the controls will be maintained through the viewstate of the page—even for dynamically added controls. So you only have to perform the data binding to the data source if this is not a postback, just as you would if you had declared the DataGrid control directly within the HTML section of the page.

The viewstate of the page also stores the values of many of the other properties of the controls on the page. So if you allow users to modify properties, such as whether specific columns are visible or the color of the text, you'll want these values to be preserved across page loads and not be reset every time you regenerate the control. In this case, you can set the values only the first time the page loads, at the same time as populating the DataGrid control. For example, this code sets the style of the header row only when the page first loads, but it is maintained across postbacks within the viewstate of the page:

```
If Not Page.IsPostback Then
  oGrid.DataSource = GetCustomers()
  oGrid.DataBind()
  oGrid.HeaderStyle.Font.Bold = True
  oGrid.HeaderStyle.ForeColor = Color.FromName("#ffffff")
  oGrid.HeaderStyle.BackColor = Color.FromName("#b50055")
End If
```

Loading User Controls Dynamically at Runtime

The final topic we'll briefly look at to finish this chapter is dynamically loading user controls at runtime. In theory, the principles are the same as for the DataGrid control; however, there are a couple things to be aware of with user controls. A user control is not strongly typed—in other words, it is usually generated as an instance of the generic UserControl class, whereas other server controls are specific classes from the .NET Framework class library.

You can use the LoadControl method of the Page object to load a user control. The following code takes the path and name of the .ascx disk file and returns a reference to the control as a UserControl instance that you can add to the ControlCollection instance of any other control:

```
Dim oNewCtrl As UserControl = LoadControl("path-to-ascx-file")
oExistingControl.Controls.Add(oNewCtrl)
```

This is fine if the user control is simply some static user interface content. However, if you want to access properties or other members of the user control, you have a problem because the UserControl class that represents the user control doesn't expose them. In that case, you have to add to the page a reference to the user control, and you have to specify the classname of the user control in the .ascx file.

When you insert a user control in the page declaratively, you use a Register directive to specify the tag prefix and tag name you'll be using, and you use the path and name of the .ascx file that implements the user control. Here's an example:

```
<%@Register TagPrefix="ahh" TagName="Spinbox"
            Src="..\ascx\user-spinbox.ascx" %>
```

When you want to insert a user control dynamically and be able to access it as a strongly typed object, you use the `Reference` directive instead. The following example just takes the path and name of the user control:

```
<%@Reference Control="..\ascx\user-spinbox.ascx" %>
```

However, this assumes that the user control itself declares a classname. In the user control, you have to add the `ClassName` attribute to the `Control` directive, as in this example:

```
<%@Control Language="VB" ClassName="UserSpinBox" %>
```

Now you can use the `CType` statement in Visual Basic (or a direct cast in C#) to convert the `UserControl` reference into a reference to the specific class. For example, you can load an instance of the `SpinBox` user control you'll be meeting later in this book and expose it as a `UserSpinBox` instance with the following code:

```
Dim oCtrl As UserControl = LoadControl("..\ascx\user-spinbox.ascx")
Dim oSpinBox As UserSpinBox = CType(oCtrl, UserSpinBox)
```

Alternatively, if you just want to set a property (such as the `Increment` property), you can use something like this:

```
Dim oCtrl As UserControl = LoadControl("..\ascx\user-spinbox.ascx")
CType(oCtrl, UserSpinBox).Increment = 3
oPlaceHolder.Controls.Add(oCtrl)
```

An Example of Loading a User Control

To briefly demonstrate the dynamic loading of a user control, the final example in this chapter loads instances of the custom `SpinBox` user control, as shown in Figure 1.13.

The page contains both a `Register` and a `Reference` directive for the `SpinBox` user control:

```
<%@Register TagPrefix="ahh" TagName="Spinbox"
            Src="..\spinbox\ascx\user-spinbox.ascx" %>
<%@Reference Control="..\spinbox\ascx\user-spinbox.ascx" %>
```

The first `SpinBox` instance you see in the page is inserted declaratively, which is possible because of the presence of the `Register` directive. However, as you can see from Listing 1.17, the remainder of the page is made up basically of three `PlaceHolder` controls where you can dynamically add the other instances of the `SpinBox` control.

Running the `SpinBox` Example

As you'll see in Chapter 8, "Building Adaptive Controls," which discusses the `SpinBox` control, you have to copy a file that we provide with the samples into the `aspnet_client` folder of your Web site for this example to work. You should copy the file `spinbox.js` from the samples into a new subfolder named `custom` within the `aspnet_client` folder of your Web site.

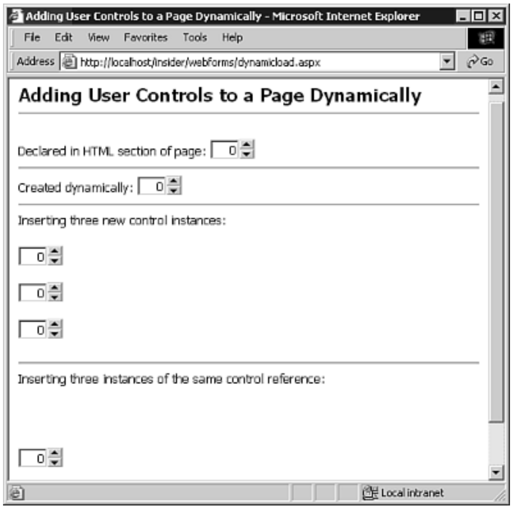


FIGURE 1.13 A sample page that demonstrates loading user controls dynamically.

LISTING 1.17 The `<form>` Section of the Page and Declaration of One `SpinBox` Control

```
<form runat="server">
  Declared in HTML section of page: <ahh:Spinbox runat="server" />
  <hr />
  Created dynamically: <asp:PlaceHolder id="ph1" runat="server" />
  <hr />
  Inserting three new control instances:<p />
  <asp:PlaceHolder id="ph2" runat="server" />
  <hr />
  Inserting three instances of the same control reference:<br />
  <asp:PlaceHolder id="ph3" runat="server" />
</form>
```

Listing 1.18 shows the `Page_Load` event handler. You generate a `SpinBox` control as a `UserControl` instance, and then you set the `Increment` property by converting the reference into a `UserSpinBox` instance, before adding it to the `ControlCollection` instance of the first `PlaceHolder` control.

Next, you add three separate new instances of the `SpinBox` control to the `ControlCollection` instance of the second `PlaceHolder` control. (You may have wondered if it is possible to use multiple instances of the same control.) You place each one on a new line by separating them with a `<br />` element. Notice how you generate this by using an `HtmlGenericControl` instance, as mentioned earlier in this chapter.

LISTING 1.18 The `Page_Load` Event Handler That Loads the `SpinBox` Control Instances

```
Sub Page_Load()

  Dim oCtrl1 As UserControl = LoadControl("..\spinbox\ascx\user-spinbox.ascx")
```

LISTING 1.18 Continued

```
  CType(oCtrl1, UserSpinBox).Increment = 3
  ph1.Controls.Add(oCtrl1)

  Dim oCtrl2 As UserControl
  For iCount As Integer = 1 To 3
    oCtrl2 = LoadControl("..\spinbox\ascx\user-spinbox.ascx")
    ph2.Controls.Add(oCtrl2)
    ph2.Controls.Add(New HtmlGenericControl("br"))
  Next

  Dim oCtrl3 As UserControl = LoadControl("..\spinbox\ascx\user-spinbox.ascx")
  For iCount As Integer = 1 To 3
    ph3.Controls.Add(oCtrl3)
    ph3.Controls.Add(New HtmlGenericControl("br"))
  Next

End Sub
```

Finally, the code demonstrates a common mistake that some people make when inserting controls into a page dynamically. Instead of creating a new instance of the control each time (as in the previous `For...Next` loop with the `LoadControl` method), it simply loads the user control and then inserts it into the `PlaceHolder` control's `ControlCollection` instance three times.

If you look back at Figure 1.13, you'll see that even though the control gets added three times, the sequence of actions that render the page remove all but the last instance. This is because the three references in the `ControlCollection` collection are all to the same instance of the user control. So remember to create new instances of your user controls if you want to insert multiple instances into the page.

Summary

This chapter covers quite a few different but interlinked topics. It starts with a look at how you can get more from the very clever ASP.NET validation controls. In particular, it looks at how you can use them with non-text controls and how you can validate other types of controls that do not directly support validation. This section of the chapter concludes with an example that uses the validation controls within a `DataGrid` control. Along the way, we looked at using images in a `DataGrid` control column to indicate `Boolean` values, populating and selecting values in simple nested list controls, and storing the source `DataSet` instance in the user's ASP.NET session to improve performance and efficiency.

Next, this chapter looks at some different issues with the `DataGrid` control, specifically aimed at exerting more control over presentation of the contents. It talks about how you can control the width of columns, edit more than one value in a cell, and provide scrolling in a cell to avoid having the `DataGrid` control expand vertically when long text strings are displayed.

This chapter also looks at the techniques for inserting controls into a page dynamically. As well as covering some of the basic theory of the ASP.NET control tree, this chapter provides an example that dynamically creates a `DataGrid` control, complete with inline editing. This involves considering when to populate the grid, as well as wiring up the events in the `DataGrid` control to the appropriate event handlers.

Finally, this chapter finishes with a look at the issues involved in loading user controls dynamically and being able to access their properties and methods as strongly typed objects.

2

Cross-Page Posting

IN THIS CHAPTER

Isn't it amazing how some people are never satisfied? In ASP 3.0, it was becoming the de rigueur approach to build pages that post back to themselves and include code that detects which button was clicked, extract the values of the HTML controls on the page, and then repopulate them. This required loads of fiddly work, inserting `value` and `selected` attributes into each control and building the decision constructs that decide which code to execute in response to the user's action.

Then along came ASP.NET, with its fiendishly clever postback architecture that does all the difficult stuff automatically. Hardly any code is required, there's no need to poke around in the `Request.Form` and `Request.QueryString` collections, and even proper event handling is provided.

So what do people keep asking how to do now? They want to post values back to a different page! A lot of programmers at Microsoft would be turning in their graves if they weren't still alive to see it. However, because there actually are some legitimate situations in which this is useful, this chapter looks at the possibilities and techniques for implementing ASP.NET server-side forms that post back to different pages. You might want to do this if you need to take advantage of pages in another application or site but still want to use server controls in your page. Or you might want to reuse pages so that each one can receive values from several sources. Whatever the reason, this chapter demonstrates how you can achieve it.

Techniques for Passing Values Between Pages

ASP.NET engenders a postback architecture, where pages containing a server-side form (a `<form>` element that contains the `runat="server"` attribute) are always posted back to themselves. In fact, this is effectively enforced by ASP.NET, which doesn't allow server-side code to set the `action` attribute of a server-side form (in other words, the `Action` property of the `HtmlForm` control instance that implements a server-side form) to any value other than the current URL.

However, there are basically four ways that you can force a `<form>` element to pass values to a different page:

- **Use a non-server control for the `<form>` element—in other words, omit the `runat="server"` attribute**—This means that the page will behave just as in ASP 3.0, and you can collect the values in the controls on the form from the `Request` collections in the traditional way. This also allows you to have multiple forms on the page, but it prevents you from using many of the ASP.NET server controls on the form. It also prevents the ASP.NET postback architecture from working, so you cannot access the controls on the original page—you can access only their posted values.
- **Use client-side script to change the `action` attribute of the `<form>` element after the page has loaded into the browser**—However, this method requires the target page to have the MAC encoding check on the viewstate disabled to prevent an error.
- **Use the `Response.Redirect` method to load the target page after the values have been posted back to the original page**—The submitted values must then be extracted from the `Request` collections, and you also have to use an intermediate page to handle the case where the `method` of the form is set to `POST` (the default for a server-side form) rather than `GET`.
- **Use the `Server.Transfer` or `Server.Execute` method to cause the second page to run within the context of the original page**—In this case, you can expose values and controls as properties of the original page and access them in the target page. The user does not see the URL of the target page in his or her browser.

This chapter does not look at the first of these techniques because it does not differ from traditional pre-ASP.NET methods. However, it does look at two sample pages that explore the concepts of the other three techniques.

Accessing Request Values in Another Page

The sample page `redirectpage.aspx`, shown in Figure 2.1, allows you to experiment with the second and third of the techniques listed in the preceding section. The page contains several server-side controls, hosted within a server-side `<form>` element. The first three (the text box, list, and drop-down list) are only there to provide values that will be passed to the target page. The option buttons allow you to select which method will be used: `POST` or `GET`.

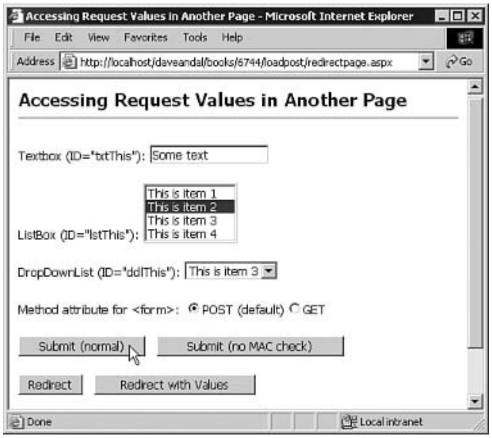


FIGURE 2.1 A sample page that demonstrates changing the `action` attribute and redirection.

Below these controls are four buttons that submit the form to the server. The first two use client-side code to change the `action` attribute of the `<form>` element before the page is submitted so that it is actually submitted to a different target page. The second two buttons are wired up to server-side event handlers that call the `Response.Redirect` method to load the target page.

Changing the `action` Attribute of a Form

The first two buttons in the sample page shown in Figure 2.1 are ordinary HTML `<input>` elements that call a client-side script function named `changeAction` and pass to it the name of the target page that will be loaded:

```
<input type="submit" name="btnChangeAction"
       value="Submit (normal)"
       onclick="changeAction('catchrequest.aspx')" />  
<input type="submit" name="btnActionNoMAC"
       value="Submit (no MAC check)"
       onclick="changeAction('catchnomac.aspx')" />
```

The `changeAction` function simply changes the `action` attribute of the server-side `<form>` element that contains all the controls on the page to the specified URL:

```
function changeAction(sURL) {
  var theForm = document.getElementById('frmMain');
  theForm.action = sURL;
}
```

In the case of the first button, the target page is `catchrequest.aspx`. This page contains an ASP.NET `Label` control and a server-side `<form>` element with an ASP.NET `Button` control:

```
<b>Values in the Request collections</b>:<br />
<asp:Label id="lblRequest" runat="server" />

<form runat="server">
  <asp:Button Text="Back" runat="server" OnClick="GoBack" />
</form>
```

The server-side code in this page, shown in Listing 2.1, implements a `Page_Load` event handler that simply iterates through the `Request.QueryString` and `Request.Form` collections, collecting any values stored there and displaying them in the `Label` control. The `GoBack` event handler, which executes when the button on the page captioned Back is clicked, redirects the browser back to the original page.

LISTING 2.1 The Page_Load and GoBack Event Handler Routines

```
Sub Page_Load()
  If Not Page.IsPostback Then

    ' display the values in the Request collections
    lblRequest.Text &= "* QueryString collection:<br />"
    For Each oValue As String In Request.QueryString
      lblRequest.Text &= "  " & oValue & " = " _
                      & Request.QueryString(oValue) & "<br />"
    Next
    lblRequest.Text &= "* Form collection:<br />"
    For Each oValue As String In Request.Form
      lblRequest.Text &= "  " & oValue & " = " _
                      & Request.Form(oValue) & "<br />"
    Next

  End If
End Sub

' return to previous page and end current response
Sub GoBack(sender As Object, args As EventArgs)
  Response.Redirect("redirectpage.aspx", True)
End Sub
```

When you try this example by clicking the Submit (Normal) button in the original page, you see an ASP.NET error page, indicating that the viewstate for the page is corrupted (see Figure 2.2). This is because ASP.NET encodes the viewstate it stores in the page along with the control tree and other details of the original page. When a postback occurs, ASP.NET validates this encoded data against the current page to act as a guard against malicious spoofing or other attacks. Because the page that is now executing is different from the original page, the validation check fails.

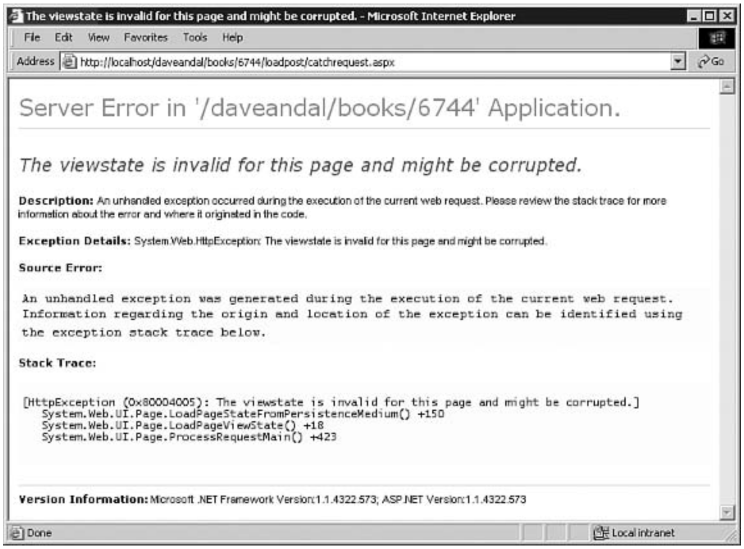


FIGURE 2.2
The error message displayed when the viewstate validation check fails.

Turning Off Viewstate Validation

You can get around the failed validation check problem by turning off viewstate validation in the target page. This means, of course, that the target page is no longer protected against spoofing, so if you use this technique, you must be sure to fully validate any submitted values to prevent malicious activity.

To turn off viewstate validation, you simply add the attribute `EnableViewStateMac="False"` to the `Page` directive. In this example, clicking the Submit (No MAC Check) button on the original page changes the `action` attribute of the form to point to the page `catchnomac.aspx`. This page is identical to the `catchrequest.aspx` page, except that it also contains the `EnableViewStateMac="False"` attribute. The result of clicking this button is shown in Figure 2.3, where you can see that now the values in the `Request.Form` collection are available and displayed.

Changing the `Method` Property of a Server-Side Form Control

In the example described in the preceding section, the values in the `<form>` element are posted to the server because the default for the `method` attribute of a server-side form is `POST`. However, you can change it to `GET` by adding the `method="get"` attribute to the declaration of the `<form>` control:

```
<form method="get" runat="server">
```

You can also change the `Method` property by using server-side code in your ASP.NET page or with client-side script code. The sample page allows you to change the `method` attribute of the form by using client-side script, which means that you can choose the method you want to use before submitting the form. Figure 2.4 shows the two option buttons for this, which are located on the page above the four submit buttons.

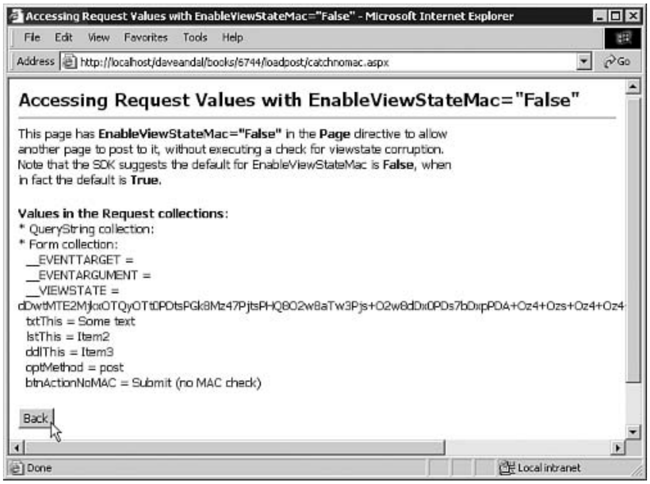


FIGURE 2.3 Turning off viewstate validation to allow the target page to execute.

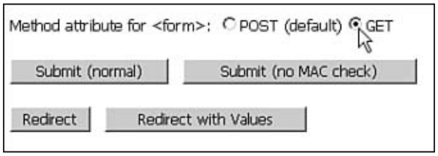


FIGURE 2.4 Changing the `Method` property of a server-side form.

Listing 2.2 shows the declaration of the `RadioButtonList` control that creates these option buttons. It has `AutoPostback` set to `True`, and any change to the selected index executes the server-side event handler named `ChangeMethod` (also shown in Listing 2.2). This just sets the `Method` attribute to the selected value.

LISTING 2.2 A `RadioButtonList` Control and an Event Handler to Change the Method of a Form

```
<asp:RadioButtonList id="optMethod" runat="server"
     RepeatDirection="Horizontal" RepeatLayout="Flow"
     OnSelectedIndexChanged="ChangeMethod" AutoPostback="True">
  <asp:ListItem Text="POST (default)" Value="post" Selected="True" />
  <asp:ListItem Text="GET" Value="get" />
</asp:RadioButtonList>
...
Sub ChangeMethod(sender As Object, args As EventArgs)
  frmMain.Method = optMethod.SelectedValue
End Sub
```

If you select the GET option button and then click the Submit (No MAC Check) button again, the values then appear in the `Request.QueryString` collection instead of in the `Request.Form` collection. They are also visible in the browser's address bar, appended to the URL as the query string.

Redirecting Postbacks to the Target Page

The two submit actions described in the preceding section work by fooling the browser and ASP.NET into working just like they do in ASP 3.0 and earlier. The browser automatically posts the values of the elements on the form or sends them as a query string, depending on the value of the method attribute of the <form> element.

The technique that this section examines uses a different approach. You allow ASP.NET to perform a postback in the usual way, but then you perform redirection to the target page by using the Response.Redirect method in the server-side code.

The server-side event handler named DoRedirect, which you attach to the Redirect button on the original page, looks like this:

```
Sub DoRedirect(sender As Object, _
       args As EventArgs)
  Response.Redirect("catchrequest.aspx", _
         True)
End Sub
```

It simply redirects the browser to the same catchrequest.aspx page (described in Listing 2.1) that displays the values in the Request.QueryString and Request.Form collections. However, if you click the Redirect button, you'll see that there are no values sent from the page following a call to the Response.Redirect method (see Figure 2.5).

> **Halting Execution by Using the Response.Redirect Method**
>
> **The Redirect method has two overloads. The first takes a single parameter—the URL of the page to redirect to. The second overload accepts an additional Boolean parameter, which indicates whether processing of the current page should be halted. Usually, when you perform a redirection, you set this second parameter to True. However, in some cases you might like to continue executing the original page code—even though the output from the page will not be sent to the client. For example, you might want to redirect the user to a different page when an error occurs but allow the original page code to clean up any resources it's using, such as closing database connections. You can always halt execution later by calling the Response.End method.**

FIGURE 2.5
The Response.Redirect method does not pass values to the target page.

Passing Form Values to a Target Page by Using Response.Redirect

Listing 2.3 shows the server-side event handler that is attached to the Redirect with Values button, the last of the four buttons on the original sample page. Similar to the Redirect button, it performs a redirection to the catchrequest.aspx page, which displays the values in the Request.QueryString and Request.Form collections. However, before it does this, it creates a query string containing the values of the text box, list, drop-down list, and option buttons on the page (you don't pass the values of the four buttons).

LISTING 2.3 The Event Handler for the Redirect with Values Button

```
Sub DoRedirectValues(sender As Object, args As EventArgs)

  ' create query string containing control values
  Dim sQuery As String = "?txtThis=" & txtThis.Text _
    & "&lstThis=" & lstThis.SelectedValue _
    & "&ddlThis=" & ddlThis.SelectedValue _
    & "&optMethod=" & optMethod.SelectedValue

  ' get setting of "method" option buttons
  If optMethod.SelectedValue = "post" Then

    ' redirect to a page that will post them to the "catch" page
    Response.Redirect("postrequest.aspx" & sQuery, True)

  Else

    ' redirect straight to the "catch" page
    Response.Redirect("catchrequest.aspx" & sQuery, True)

  End If
End Sub
```

The next decision depends on the value of the `RadioButtonList` control that creates the two option buttons—it will be either `POST` or `GET`. If it is `POST`, you redirect not to the `catchrequest.aspx` page, but to another page, named `postrequest.aspx` (which we'll examine shortly), that itself redirects to the `catchrequest.aspx` page.

However, if the GET option button is selected, you redirect straight to the `catchrequest.aspx` page. In this case, the control values added to the query string will be available in the `Request.QueryString` collection within the target page.

Posting Form Values with Redirect via an Intermediate Page

In this chapter you've seen that when you use the `Response.Redirect` method, the values in a `<form>` element are not passed to the target page. In the previous section you got around this by adding the values to the query string. However, if the target page requires the values to be posted as part of the form itself, you have to introduce a little subterfuge to achieve this.

In this case, you redirect to an intermediate page that captures the values from the query string and inserts them into hidden-type `<input>` controls within a `<form>` section. Then you arrange for the form to be posted to the actual target page, where the values will appear in the `Request.Form` collection.

Listing 2.4 shows the `Page_Load` event handler of the intermediate page, named `postrequest.aspx`. In it you simply iterate through the name/value pairs in the `Request.QueryString` collection, create a new `HtmlInputHidden` control for each one, set the name (using the `ID` property, which sets the `id` and `name` attributes), and specify the value. Then you add the hidden control to the `Controls` collection of an ASP.NET `PlaceHolder` control located within the `<form>` section of the page.

LISTING 2.4 The `Page_Load` Event Handler for the Intermediate Posting Page

```
Sub Page_Load()

 Dim oInput As HtmlInputHidden
 Dim sItem As String

 For Each sItem in Request.QueryString
   oInput = New HtmlInputHidden()
   oInput.ID = sItem
   oInput.Value = Server.HtmlEncode(Request.QueryString(sItem))
   phForm.Controls.Add(oInput)
 Next

End Sub
```

Listing 2.5 shows the HTML declarations for the `postrequest.aspx` page and the client-side script that submits it to the server automatically. The `<form>` element is not a server-side form in this case because you need to set the `action` attribute to the URL of the target page that will receive the values. And you must remember to set the method attribute to `POST` because the default for a non-server-side form is `GET`.

LISTING 2.5 The Client-Side Code to Submit the Sample Form

```
<script language="JavaScript">
  function submitForm() {
    document.forms[0].submit()
  }
</script>
...
<body onload="submitForm()">
  <form action="catchrequest.aspx" method="post">
    <asp:PlaceHolder id="phForm" runat="server" />
  <noscript>
    <input type="submit" value="Click to continue" />
  </noscript>
  </form>
</body>
</html>
```

Query String Considerations

Passing values in the query string is fine, as long as they are limited in size. Depending on the browser and server in use, the total length of the URL and query string will be limited to somewhere between 1KB and 4KB. If the pages are complex, and especially if they contain items such as a `DataGrid` control, the viewstate that is sent as part of the request can grow to alarming proportions. For example, the viewstate of the sample page, containing just a few simple controls, is over 120 bytes.

The other issues with including form values in the query string are that it makes these values highly visible to users and that it allows users to bookmark the page with these values included in the query string. For many applications, therefore, you will probably want to use the `Request.Form` collection to pass values between pages. However, the intermediate page method described in this section doesn't actually solve the problem because part of the process still involves the use of the query string. In this case, you can either use the approach described earlier in this chapter—changing the action of the form and turning off viewstate validation—or you can use the technique described in the following section, which uses the `Server.Transfer` method.

Notice that the page also contains a `<noscript>` section that will display a simple HTML submit button if scripting is not available. This is required for the user to be able to submit the values to the server in this case.

However, you want the values to be submitted automatically when possible, and the page contains a simple client-side script function that achieves this by calling the `submit` method of the `<form>` element. You call this function as soon as the page has finished loading by specifying it as the `onclick` attribute of the opening `<body>` tag in the page. The result is that this page will submit the values in the hidden `<input>` elements to the target page, using the `POST` method—without requiring user intervention unless client-side scripting is disabled or unavailable.

Figure 2.6 shows the results of clicking the Redirect with Values button. You can see that the values added to the query string in the original pager, before the call to the `Redirect` method, now appear in the `Request.Form` collection.

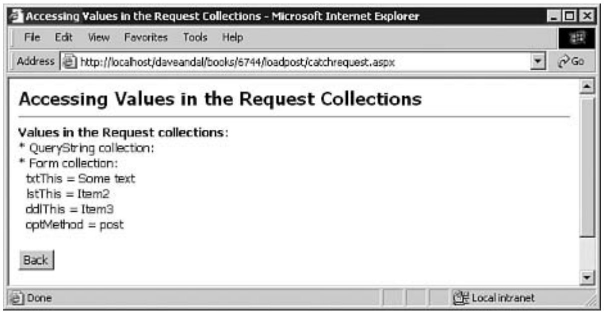


FIGURE 2.6
The result of using an intermediate page to post the values to the target page.

Client-Side Versus Server-Side Redirection

Earlier in this chapter we talked about how a Web server sends HTTP headers to the client in response to every HTTP request. These headers contain information about the Web server and the resource it is sending back. The HTTP redirection header "302 Object Moved" causes the browser to load the resource at a different URL from the one it originally requested. Of course, the browser responds to this header by issuing a new request, which specifies the new location of the resource.

The `Response.Redirect` method used in earlier examples in this chapter relies on this redirection process. It works by sending the HTTP redirection header back to the client. The client then loads the target page from the new location. So even though it is a server-based instruction, it is actually an example of *client-side* redirection.

The fourth technique for forcing a form to post to an alternative page and passing values between these pages described earlier in this chapter, in the section "Techniques for Passing Values Between Pages," involves the use of *server-side* redirection.

Limitations of the `Response.Redirect` Method

One limitation with the `Response.Redirect` method is that it must be executed before any content (that is, anything except other HTTP headers) has been sent to the client. In early versions of ASP, you had to enable buffering on the server by executing `Response.Buffer = True` to prevent any content from being sent until the page was complete or until you executed the `Response.Flush` or `Response.End` methods. Since version 3.0 of ASP, buffering has been on by default, and this is the case in ASP.NET as well. This means that you can usually call `Response.Redirect` anywhere in a page, as long as you haven't turned buffering off, executed the `Response.Flush` method, or executed one of the new ASP.NET methods that writes output directly to the response (such as `Response.WriteFile`).

Microsoft added server-side redirection to ASP in version 3.0 by providing two new methods for the `Server` object:

- **`Transfer`**—This method causes execution of the current page to end at the point where the method is called, and control passes to the new page. When that page ends, the response is complete. It is effectively a `GOTO` statement.
- **`Execute`**—This method causes execution to pause at the point where the method is called, and control passes to the new page. However, when execution of the new page ends, control passes back to the original page and continues from the point where it was paused. It is more like a `GOSUB` statement or a function call.

Note that these methods only accept a virtual path located on the same Web site.

These two methods continue to be available in ASP.NET, and in fact are more useful than ever before because you can now create a reference to the instances of the original page in code that is running in the new page. This effectively means that you can communicate with the original page, making it easy to pass values from one page to another.

Both the `Transfer` and `Execute` methods are completely server bound and do not involve the client. They don't rely on the client responding to HTTP headers, as does the `Response.Redirect` method, and there is no indication in the browser that redirection is taking place. The URL of the page doesn't change, and the user just sees output as though it were generated by the original page—regardless of how and when the `Transfer` and `Execute` methods were executed.

Because the new page is executed within the context of the original page, rather like a function or subroutine, all globally available (that is, `Public`) objects are accessible in the new page through the HttpContext object that ASP.NET uses to keep track of the original page. If you check out the properties of the HttpContext class, you'll see that it provides access to everything that you use in your ASP.NET code. You can access the current Request, Response, Server, Session, and Application objects, plus objects such as User (details of the current user), Cache (the ASP.NET global user cache), and Trace. So there are really no limitations on what code within the page that you transfer to, or execute, can do.

Exposing Values to Another Page via References

The ability to access the current context within a page that is being executed in response to the Server.Transfer method or the Server.Execute method is extremely useful. However, it becomes even more useful when combined with the fact that ASP.NET allows you to create a reference to the original page within the context of the page that you transferred to or executed.

For this to work, you must assign a classname to the original page by adding it as an attribute to the Page directive. For example, you can assign the name ReferencePage to the class that is created when the page is compiled. Although in this case the page file is named referencepage.aspx, the classname and filename do not have to be the same—any names can be used:

```
<%@Page Language="VB" ClassName="ReferencePage" %>
```

Then, in a page that will be executed using the Server.Transfer method or the Server.Execute method, you can create a reference to the current instance of the original page. First, you have to add a reference to the original page file that contains the class definition (that is, the page that declares the classname):

```
<%@Page Language="VB" %>
<%@Reference Page="referencepage.aspx" %>
```

Now code running in the new page can use the Handler property of the Context object to get a reference to the original page and cast it to the correct type:

```
Dim oPage As ReferencePage = CType(Context.Handler, ReferencePage)
```

In C#, this is the equivalent:

```
ReferencePage oPage = (ReferencePage) Context.Handler;
```

Then, through the reference you've created, you can access any Public content of the original page. You can do the following:

- Read and write the values of Public properties (unless they are declared as being ReadOnly or WriteOnly)
- Call Public functions and execute Public subroutines
- Access server controls that are referenced through a Public ReadOnly property

The sample page described in the following sections demonstrates many of these features for the `Server.Transfer` method. You'll learn more about the `Server.Execute` method later in this chapter, in the section "The `Server.Execute` Method."

> **Calling Event Handlers in the Original Page**
>
> In theory, you can call any `Public` event handlers in the original page, although it's not obvious where this would be directly useful. It's generally better to expose references to any server controls in the page that you want to interact with as `Public` properties.

The sample page named `referencepage.aspx` demonstrates the way that a reference to the original page can be used with the `Server.Transfer` method (see Figure 2.7). This page contains the same set of controls as the previous example, but it has only two submit buttons.

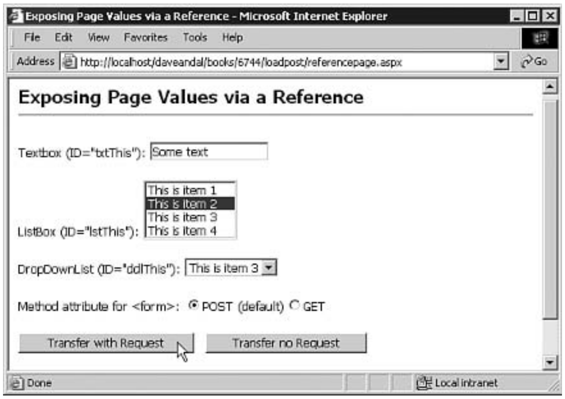


FIGURE 2.7
A sample page that uses the `Server.Transfer` method.

The reason for the two buttons is that the `Server.Transfer` method has two overloads. The first takes a single parameter, which is the URL or name of the page to transfer control to. The second overload accepts an additional `Boolean` parameter that determines whether the contents of the `Request` collections (`Form`, `QueryString`, `Cookies`, and `ServerVariables`) will *not* be cleared when the transfer takes place. The default value for this parameter is `True` (the Framework SDK says the default is `False`, but that is incorrect), which means that the collections will be preserved. If you set it to `False`, the collections will not be preserved—in other words, all the values in the collections will be removed.

The Event Handlers That Call the `Server.Transfer` Method

Listing 2.6 shows the event handlers for the two buttons in the sample page. The first is attached to the Transfer with Request button and specifies that the `Request` collections will be preserved. The second event handler is attached to the Transfer No Request button and specifies that the `Request` collections will not be preserved.

LISTING 2.6 Two Event Handlers That Initiate a `Server.Transfer` Method

```
Sub DoTransferTrue(sender As Object, args As EventArgs)

  ' use True to specify that Request collections are *not* cleared
  ' in fact True is the default anyway (the SDK is wrong on this)
  Server.Transfer("catchreference.aspx", True)

End Sub


Sub DoTransferFalse(sender As Object, args As EventArgs)

  ' use False so that no Request values are passed
  ' page properties are still available in target page
  Server.Transfer("catchreference.aspx", False)

End Sub
```

The Public Properties in the Main Page

The main page shown in Figure 2.7 defines a name for the class it creates as an attribute of the `Page` directive, just as is done earlier in this chapter:

```
<%@Page Language="VB" ClassName="ReferencePage" %>
```

Inside the `<script>` section of the page, you declare the `Public` properties that you want to expose to the page to which you'll transfer execution. You declare three `ReadOnly` properties, as shown in Listing 2.7. The `TextValue` property returns the `Text` property of the `TextBox` control; the `ListIndex` property returns the `SelectedIndex` property of the `ListBox` control; and the `DropList` property returns a reference to the `DropDownList` control itself.

LISTING 2.7 The `Public` Properties Declared Within the Main Sample Page

```
' public properties exposed to other pages
Public ReadOnly Property TextValue As String
  Get
    Return txtThis.Text
  End Get
End Property

Public ReadOnly Property ListIndex As Integer
  Get
    Return lstThis.SelectedIndex
  End Get
End Property
```

LISTING 2.7 Continued

```
Public ReadOnly Property DropList As DropDownList
  Get
    Return ddlThis
  End Get
End Property
```

BEST PRACTICE

Exposing Control Values or Control References As Properties

If you only want to access specific properties of a control or values that are used in the code within the page, exposing these as simple individual values is the best approach. When you do so, you maintain control over the values that can be accessed in the target page. In this example, all three are `ReadOnly` properties, so they cannot be changed in the target page. However, with the exception of the reference to the `DropDownList` control, you could declare the properties as read/write by omitting the `ReadOnly` keyword and including a `Set...End Set` section within the property declaration. Chapter 5, "Creating Reusable Content," describes the syntax for declaring properties in more detail.

Exposing a reference to a control itself, as you've done with the `DropDownList` control, is useful if the code in the target page will need to access (read and/or write) a range of properties of the control. For example, you could set multiple properties by using something like this:

```
With MyPage.DropList
  .DataSource = MyDataReader
  .DataTextField = "thiscolumn"
  .SelectedIndex = 3
  With .Items
    .Insert(0, New ListItem("First Option", "0"))
    .Add(New ListItem("Last Option", "999"))
  End With
End With
```

You can also call methods on the control, as in this example:

```
MyPage.DropList.DataBind()
```

The Target Page for the `Server.Transfer` Method

As shown in Listing 2.6, both the buttons in the main sample page transfer execution to a target page named `catchreference.aspx`. The only difference is that the second button clears the contents of the `Request` collections when the transfer takes place. The target page simply displays the values of the Public properties that are exposed by the main page, plus the contents of the Request.QueryString and Request.Form collections. Figure 2.8 shows the target page when the Transfer with Request button in the main page is clicked.

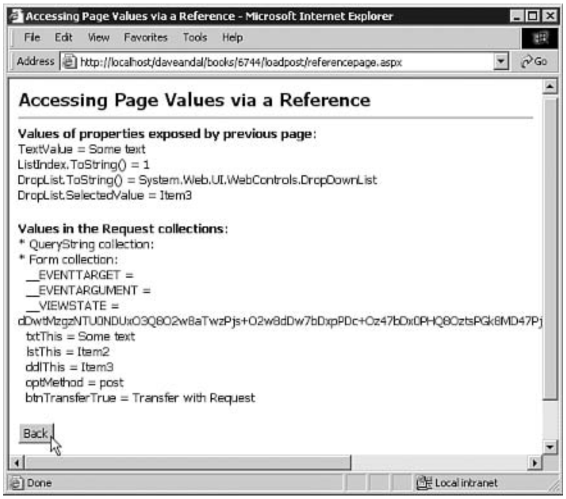


FIGURE 2.8
The target page for the transfer method, displaying the values of the properties and Request collections.

The HTML declarations in the target page are identical to those in the catchrequest.aspx page used to display the values in the Request collections in the earlier examples in this chapter, with the exception of an extra Label control that you see at the top of the page (refer to Figure 2.8). This control is used to display the values of the properties exposed by the main page.

There is the same Back button at the bottom of the page, and the same server-side event handler as in the catchrequest.aspx page is used to direct the browser back to the main page.

Where the catchreference.aspx page really differs from the catchrequest.aspx page is in the Page_Load event handler, as shown in Listing 2.8. Within a Try...Catch construct, you attempt to extract the values of the properties from the main page, and you display these in the first Label control. You use a Try...Catch construct in case the page is loaded directly, in which case it will not be able to create an instance of the original page.

LISTING 2.8 The Page_Load Event Handler for the Target Page

```
Sub Page_Load()
  If Not Page.IsPostback Then

    Try

      ' get a reference to the previous page
      Dim oRefPage As ReferencePage
      oRefPage = CType(Context.Handler, ReferencePage)

      ' display the property values from the previous page
```

LISTING 2.8 Continued

```
      lblProperties.Text = "TextValue = " _
                    & oRefPage.TextValue & "<br />" _
                    & "ListIndex.ToString() = " _
                    & oRefPage.ListIndex.ToString() & "<br />" _
                    & "DropList.ToString() = " _
                    & oRefPage.DropList.ToString() & "<br />" _
                    & "DropList.SelectedValue = " _
                    & oRefPage.DropList.SelectedValue

    Catch
      lblProperties.Text = "ERROR: Cannot reference previous page"
    End Try

    ' display the values in the Request collections
    lblRequest.Text &= "* QueryString collection:<br />"
    For Each oValue As String In Request.QueryString
      lblRequest.Text &= "  " & oValue & " = " _
                      & Request.QueryString(oValue) & "<br />"
    Next
    lblRequest.Text &= "* Form collection:<br />"
    For Each oValue As String In Request.Form
      lblRequest.Text &= "  " & oValue & " = " _
                      & Request.Form(oValue) & "<br />"
    Next

  End If
End Sub
```

Notice how you get a reference to the original page from the `Context.Handler` property and convert it onto the specific class `ReferencePage` that is declared in the main page, just as described earlier in this chapter. When you have this reference to the main page, you can access the properties within it in the same way that you access properties of any object.

The remainder of the code in Listing 2.8 extracts any values in the `Request.QueryString` and `Request.Form` collections and displays them in the second `Label` control, in exactly the same way as in earlier examples in this chapter.

Changing the Method and Clearing the `Request` Collections

Figure 2.8 shows that the values sent from the form on the main page appear within the `Request.Form` collection. However, this is only because the default for a server-side `<form>` element, as noted earlier, is `method="post"`. You can use the option buttons to change the method of the form, as in the examples earlier in this chapter, and then you'll see that the values appear in the `QueryString` collection as expected.

Just to prove another point, if you go back to the main page and click the second button, Transfer No Request, you'll see that the `Public` properties in the original page are still available but the `Request` collections are empty (see Figure 2.9). Of course, when you're using this technique, it's usually the values of the properties exposed by the main page that you're really interested in—not the `Request` collection contents.

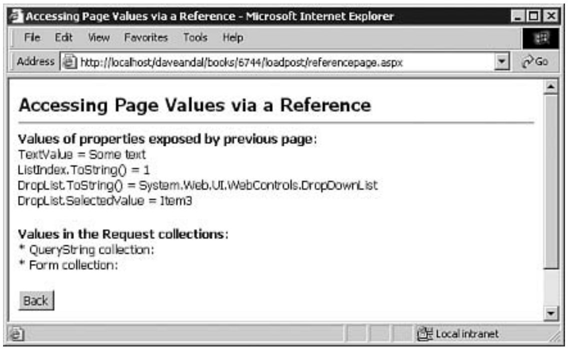


FIGURE 2.9
The result when the `Request` collections are cleared by the `Transfer` method.

BEST PRACTICE

Reducing Data Transfer Volumes by Using the `Server.Transfer` Method

One way you can reduce the amount of data you pass to the target page with the `Transfer` method is to set the second parameter of the method to `False`, to clear the values from the `Request` collections. If there are specific values in the `Request.QueryString` and `Request.Form` collections that you want to access in the target page, you can always extract them and expose them as `Public` properties of the main page.

The `Server.Execute` Method

The following sections briefly look at the `Server.Execute` method. This method works much like the `Server.Transfer` method, and you can access values in the original page in the same way. To try it out, you can simply replace this line:

```
Server.Transfer("catchreference.aspx", True)
```

with this line:

```
Server.Execute("catchreference.aspx")
```

in the sample page. You'll see the content generated by the page that is executed, followed by the content generated by the original page when execution returns to it.

The `Server.Execute` method differs from `Server.Transfer` in that it does not provide a `Boolean` parameter that determines whether the `Request` collections will be preserved. The `Request` collections are always preserved when the `Server.Execute` method is called, and they are always available in the original page when control returns to that page. If this were not the case, the ASP.NET postback architecture would fail to work correctly in cases where control properties or values had been changed.

Also, like functions or subroutines, you can use the `Server.Execute` method more than once in a page, and you can execute the same or different pages each time.

Creating Reusable Content by Using `Server.Transfer`

The `Server.Execute` method provides an interesting opportunity for creating reusable content because you can execute other pages just as though they were subroutines. You could, for example, build a library of such pages and then execute them from any of your main pages as required. Chapter 5 examines other ways of creating reusable content.

Capturing Output from the `Server.Execute` Method

A useful aspect of the `Server.Execute` method is that a second overload accepts a `StringWriter` instance as the second parameter. When the `Execute` method is called in this case, the output generated by the page that is executed is written to the `StringWriter` instance and not into the `Response` instance of the original page. This means that you can execute another page, capture the output it generates, and use it as required in the original page.

Figure 2.10 is a sample page that demonstrates this feature. It looks similar to the previous examples in this chapter, but notice that the two option buttons now allow you to specify whether the results should be HTML encoded.

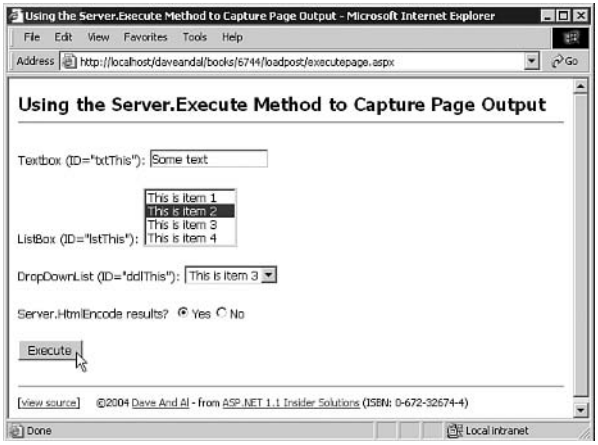


FIGURE 2.10
A sample page that demonstrates the `Server.Execute` method.

This page calls the `Server.Execute` method with a `StringWriter` instance as the second parameter, and then it displays the contents of the `StringWriter` instance in the original page. Note that the `StringWriter` class is defined within the `System.IO` namespace, so you must import that namespace into your page.

The option buttons allow you to decide whether to display the page as text with the HTML tags visible (that is, HTML encoded) or whether to display it in rendered form (as it would appear when loaded directly by a browser).

Listing 2.9 shows the DoExecute event handler that runs when the Execute button in the main page is clicked. It creates a new StringWriter instance and passes it to the Execute method, along with the URL of the target page to execute. On return, the code checks the value of the RadioButtonList control to see whether the result should be HTML encoded. If it is being encoded, the output is wrapped in a <pre> element so that the source of the target page is displayed with the carriage returns and indenting visible.

LISTING 2.9 Calling the Execute Method with a StringWriter Instance

```
Sub DoExecute(sender As Object, args As EventArgs)

  ' create StringWriter and use it when executing target page
  Dim oWriter As New StringWriter()
  Server.Execute("catchexecute.aspx", oWriter)

  ' see if result should be HTML-encoded
  If optEncode.SelectedValue = "Yes" Then
    lblResult.Text = "<pre>" _
                   & Server.HtmlEncode(oWriter.ToString()) _
                   & "</pre>"
  Else
    lblResult.Text = oWriter.ToString()
  End if

End Sub
```

The Target Page for the Server.Execute Method

The target page used in this example is basically the same as the one used in the Server.Transfer example earlier in this chapter. One difference is that you have to reference the main page for this example, executepage.aspx:

```
<%@Reference Page="executepage.aspx" %>
```

You also have to change the line that accesses the Context.Handler property and specify the classname that is declared within the executepage.aspx page:

```
Dim oRefPage As ExecutePage = CType(Context.Handler, ExecutePage)
```

The other changes are prompted by the fact that you no longer need a Back button because control passes back to the main page after execution of the target page is complete. Therefore, the controls on the main page will still be visible.

Figures 2.11 and 2.12 show the results of clicking the Execute button in this example. Figure 2.11 specifies that the result should be HTML encoded, and you can see the output that is generated by the target page—including the values of the `Public` properties that it accesses within the main page. Effectively, you are looking at the same thing you would see if you loaded the page and then selected View, Source in the browser.

FIGURE 2.11
Displaying the results of the `Server.Execute` method in HTML-encoded form.

Figure 2.12 shows the result when the content of the `StringWriter` instance is simply written into the `Label` control, without being HTML encoded first. The output is rendered just as it would be if it were loaded directly into the browser. However, it looks a little odd and has lost some formatting because the content that is being rendered contains its own opening and closing `<html>` and `<body>` tags (as you can see if you refer to Figure 2.11).

FIGURE 2.12
Displaying the results of the `Server.Execute` method, as normally rendered.

Summary

This chapter focuses on how you can force ASP.NET to load alternative pages when the user submits a form that is implemented as a server control. Although it might seem simple, there are some interesting side effects and several useful opportunities, depending on the approach you decide to take. This chapter demonstrates three of the common approaches:

- Using client-side script to change the `action` attribute of the `<form>` element after the page has loaded into the browser
- Using the `Response.Redirect` method to load the target page after the values have been posted back to the original page
- Using the `Server.Transfer` method or the `Server.Execute` method to cause the second page to run within the context of the original page

All these methods have some features that recommend them, and there is no obvious single solution. Understanding the way that each works and the limitations it applies should make it easier to choose the appropriate one when you find that you need to implement this kind of behavior in your own Web pages and Web applications.

Changing the `action` attribute of the `<form>` element is a neat way to perform redirection to another page, but you have to disable viewstate validation in the target page in order for it to succeed. Using the `Response.Redirect` method avoids this problem, but the limit on query string length might be an issue.

In many cases, the third of the techniques examined, using the `Server.Transfer` method or the `Server.Execute` method, provides the best solution. You can access properties and even controls in the original page, and in fact you can access any other features of the page as well because the target page runs in the same context as the original page. However, you can't use this approach to send values to a page on to another Web site.

Finally, this chapter looks at an alternative use for the `Server.Execute` method. Because it can accept a `StringWriter` instance and write the content of the target page to that `StringWriter` instance, you can use it to fetch content and then process it before displaying it in your pages.

3

Loading Progress and Status Displays

IN THIS CHAPTER

ASP.NET is extremely fast when you're creating and delivering Web pages. However, no matter how fast and efficient your Web server and the software it runs (including your Web applications) are, the delay between the user clicking a button and seeing the results can vary tremendously. On a good ADSL or direct Internet connection, it might be a "wow, that was quick" few seconds. On a dial-up connection, especially when the server is on the other side of the world, it's more likely to be the seemingly interminable "did I remember to pay the phone bill?" response.

One feature that most executable applications offer but that is hard to provide in a Web application is accurate status information and feedback on a long-running process. However, this can be achieved in at least two different ways, depending on the process your application is carrying out and the kind of status or feedback information you want to provide.

One technique is a "smoke and mirrors" approach, in that it makes the user feel comfortable that something is happening—while in fact the information the user sees bears no real relationship to the progress of the server-based operation. The other approach, covered toward the end of the chapter, provides accurate status and feedback details but imposes limitations on client device type and the kinds of operation for which it is suitable.

In this chapter you will see what is possible regarding loading progress and status displays. You'll learn how to use and adapt a variety of techniques to suit your own applications and requirements. This chapter starts with a look at the theory of the process and examines the simplest way it can be achieved.

Displaying a "Please Wait" Page

Many ASP.NET developers find that despite their best efforts in producing efficient code that minimizes response times, the vagaries of database response times, the transit time over the Internet, and user input criteria that are not specific enough can result in a lengthy delay before a page appears in the browser. The result is that users often click the submit button several times to try to elicit a response from your server, sometimes causing all kinds of unfortunate side effects.

Chapter 6, "Client-Side Script Integration," looks at some specific solutions for creating a one-click button. However, an alternative approach is to provide a page that loads quickly and that displays a "please wait" message or some suitable graphic feature, while the real target page is being processed and delivered. In ASP 3.0 and other dynamic Web programming environments, it's common to handle this process with separate pages that implement the three execution stages shown in Figure 3.1.

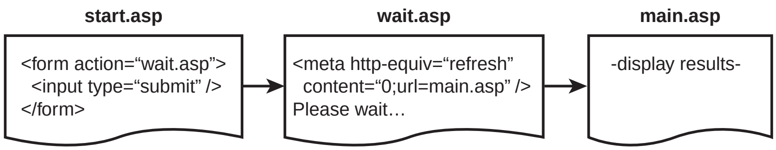


FIGURE 3.1 The traditional separate-pages approach to providing a "please wait" message.

> **Passing Values Between Requests**
>
> **Of course, what's missing from Figures 3.1 and 3.2 is how any values submitted by the user are passed from the "please wait" page to the code that creates the results. In ASP 3.0 and other dynamic Web page technologies, the usual technique is to include a placeholder within the `content` attribute of the `<meta>` element that gets replaced by a query string containing the values sent from the `<form>` section. You can then extract these from the query string in the page or section of code that generates the results. You'll see this discussed in more detail in the section "Displaying the "Please Wait" Message," later in this chapter.**

ASP.NET engenders the single-page postback architecture approach. However, you can build similar features into ASP.NET applications by implementing the three pages as separate sections of a single page. The server control approach to populating elements and attributes on the page also makes it easier to work with elements such as the `<meta>` element that you use as part of the process. Figure 3.2 shows the ASP.NET approach, as it is adopted in the example described in the following sections.

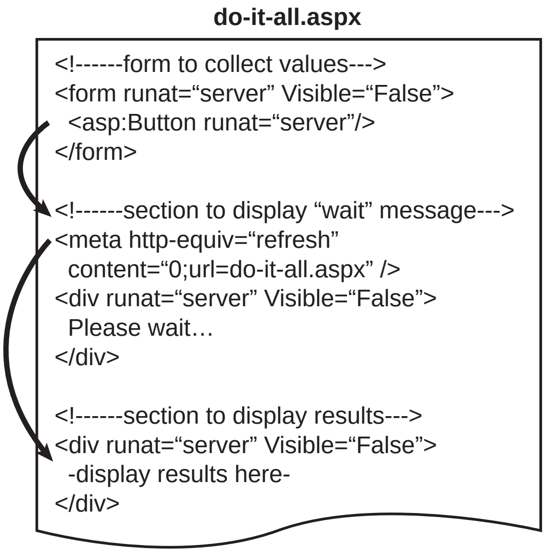


FIGURE 3.2 The ASP.NET single-page approach to providing a “please wait” message.

A Simple “Please Wait” Example

Figure 3.3 shows the initial display of a simple sample page that displays a “please wait” message while the main processing of the user’s request is taking place. The page queries the `Customers` table in the sample Northwind database that is provided with SQL Server. In the text box on the page, the user enters all or part of the ID of the customer he or she is looking for.

FIGURE 3.3 The initial page of the simple “please wait” example.

When the user clicks the Go button, the value in the text box is submitted to the server, and the page shown in Figure 3.4 is displayed. No complex processing is required to display this page, and the total size of the content transmitted across the wire is small, so it should appear very quickly. The user knows that his or her request is being handled, and there is no submit button for the user to play with in the meantime.

> **Obtaining the Sample Files**
>
> **You can download this example and the other examples for this book from the Sams Web site at `www.samspublishing.com`, or from `www.daveandal.net/books/6744/`. You can also run many of this book’s examples online at `www.daveandal.net/books/6744/`.**

FIGURE 3.4 The “please wait” message that is displayed while processing the main page.

After a short delay (about 3 or 4 seconds, in this example), the main page, which contains the results, is returned to the user and replaces the “please wait” message. You can see in Figure 3.5 that the main page contains a list of customers matching the partial ID value that was provided. At the bottom of the page is a New Customer link that takes the user back to the first page.

FIGURE 3.5 The main page, displaying the results of a search for matching customers.

The HTML and Control Declarations

Listing 3.1 shows the relevant parts of the sample page shown in the preceding section. Notice that although you include a `<meta>` element in the `<head>` section of the page, you don’t specify any attributes for it. Instead, you give it an ID and specify that it is a server control by including the `runat="server"` attribute. However, this `<meta>` element will have no effect on the page or the behavior of the browser until you specify the attributes for it in the server-side code.

LISTING 3.1 The HTML and Control Declarations for the Simple “Please Wait” Sample Page

```
<html>
<head>
<!----- dynamically filled META REFRESH element ----->
<meta id="mtaRefresh" runat="server" />
</head>
<body>

<!----- form for selecting customer ----->
```

LISTING 3.1 Continued

```
<form id="frmMain" Visible="False" runat="server">
  Enter Customer ID:
  <asp:Textbox id="txtCustomer" runat="server" />
  <asp:Button id="btnSubmit" Text="Go" runat="server" />
</form>

<!----- "please wait" display ----->
<div id="divWait" Visible="False" runat="server">
  <center>
  <p> </p>
  <p> </p>
  <b>Searching, please wait...</b><p />
  <p> </p>
  <span id="spnError"></span>
  <p> </p>
  </center>
</div>

<!----- section for displaying results ----->
<div id="divResult" Visible="False" runat="server">
  <b><asp:Label id="lblResult" runat="server" /></b><p />
  <asp:DataGrid id="dgrResult" runat="server" /><p />
  <asp:Hyperlink id="lnkNext" Text="New Customer" runat="server" />
</div>

</body>
</html>
```

The remainder of the page is made up of the three sections that implement the three pages shown in Figures 3.3 through 3.5. All three pages include a `Visible="False"` attribute in their container element—either the `<form>` element itself for the first one or the containing `<div>` element for the other two pages. So all three sections will be hidden when the page is loaded, and you can display the appropriate one by simply changing its `Visible` property to `True`.

Meta Refresh and Postback Issues

As you can see from the figures and code so far in this chapter, this example uses a `<meta>` element in the “please wait” page to force the browser to load the main page. This much-used technique is a handy way to redirect the browser to a different page, and it is supported in almost every browser currently in use today.

When you use the server-side `Response.Redirect` method in an ASP.NET (or ASP 3.0) page, the server sends two HTTP headers to the client to indicate that the browser should load a different page from the one that was requested. The `302 Object Moved` header indicates that the requested resource is now at a different location, and the `Location` *`new-url`* header specifies that the resource is located at the URL denoted by *`new-url`*.

The `<meta>` element supports the `http-equiv` attribute, which is used to simulate the effects of sending specific HTTP headers to the browser. To redirect the browser to a different URL, using a `<meta>` element, you can use this:

```
<meta http-equiv="refresh" content="[delay];url=[new-url] />
```

In this syntax, `[delay]` is the number of seconds to wait before loading the page specified in `[new-url]`. All browsers will maintain the current page they are displaying until they receive the first HTTP header sent by the server for the new page. So if the processing required for creating the new page takes a while and the server does not send any response until the processing is complete, the user will continue to see the page containing the `<meta>` element (the "please wait" message). By default, ASP.NET enables response buffering, so it does not generate any output until the new page is complete and ready to send to the browser.

BEST PRACTICE

Replacing the Existing Page in the Browser

Web browsers continue to display the existing page when you click a link in that page or enter a new URL in the address bar, while they locate and start to load the new page. However, as soon as the first items of the page that will be rendered are received (as opposed to the HTTP headers), the existing page is removed from the display, to be replaced by the progressive rendering of the new page.

One important point to note, however, is that if you disable output buffering by setting `Response.Buffer = False`, or if you force intermediate output to be sent to the response by using `Response.Flush`, the page currently displayed in the browser will be discarded as soon as the partial output of the new page is received.

You can delay the removal of the existing page in some browsers—for example, Internet Explorer supports page translations, which take advantage of the built-in Visual Filters and Transitions feature (see `http://msdn.microsoft.com/workshop/author/filter/filters.asp#Interpage_Transition`).

However, the issue here is that unlike when you submit an ASP.NET `<form>` element, the redirection caused by the `<meta>` element doesn't perform a postback. This means that viewstate for the page will not be maintained, and the values of any controls on the whole page (including the nonvisible sections) will be lost. So any values that you want to pass to the page the next time it loads (that is, when you display the results of processing the main section of the page) must be passed in the query string of the URL specified in the `<meta>` element.

Of course, this is what you would have to do in the pre-ASP.NET example shown in Figure 3.1 as well. Code in the page must collect the values from all the controls in the `<form>` section of the page when it is posted to the server, and it must build up a query string containing these within the `<meta>` element. You'll see how to do this in the following section.

The `Page_Load` Event Handler

The `Page_Load` event handler for the sample page first has to determine the current stage of the three-step process:

- **Stage 1**—The user has just posted the `<form>` element containing his or her input to the server.
- **Stage 2**—The "please wait" message is displayed, and the `<meta>` element has caused the browser to request the page containing the results.
- **Stage 3**—The user has clicked the New Customer link to go back to Stage 1.

The following sections describe the code and page content that is used in the example to implement these three stages.

Displaying the "Please Wait" Message

Listing 3.2 shows the first section of the `Page_Load` event handler for the sample page. The only time a postback will have occurred is at Stage 1 because the other two stages are initiated by a `<meta>` element or a hyperlink. (Code in a section of the `Page_Load` event handler makes the `<form>` element visible when the page first loads, as you'll see shortly.)

LISTING 3.2 The First Part of the `Page_Load` Event Handler

```
Sub Page_Load()

  If Page.IsPostback Then

    ' user submitted page with customer ID
    ' create URL with query string for customer ID
    ' next page will not be a postback, so viewstate will be lost
    Dim sRefreshURL As String = Request.Url.ToString() _
      & "?custID=" & txtCustomer.Text

    ' use META REFRESH to start loading next page
    mtaRefresh.Attributes.Add("http-equiv", "refresh")
    mtaRefresh.Attributes.Add("content", "0;url=" & sRefreshURL)

    ' hide <form> section and show "wait" section
    frmMain.Visible = False
    divWait.Visible = True

  Else
  ...
```

The `Page.IsPostback` property will be `True` only at Stage 1. At that point, you can extract the value of the text box (and any other control values that you might have in more complex examples) and build up the URL and query string for the `<meta>` element. You obviously want to reload the same page, so you get the URL from the `Url` property of the current `Request` instance. In this example, the only value you need to maintain as the page is reloaded is the value of the text box, and you use the name `custID` for this as you create the query string.

Then, as shown in Listing 3.2, you add the attributes you need to the `<meta>` element already declared in the page. You declare the `<meta>` element as a server control by using the following:

```
<meta id="mtaRefresh" runat="server" />
```

ASP.NET will implement this element as an instance of the `HtmlGenericControl` class because there is no specific control type within the .NET Framework class library for the `<meta>` element. However, the `HtmlGenericControl` type has an `Attributes` collection that you can use to add the attributes you need to it. You add the `http-equiv="refresh"` attribute and the `content` attribute, with a value that will cause the browser to immediately reload the page. If you view the source of the page in the browser (by selecting View, Source), you'll see the complete `<meta>` element:

```
<meta id="mtaRefresh" http-equiv="refresh" content="0;url=
➥/daveandal/books/6744/loadwait/simplewait.aspx?custID=a"></meta>
```

> **The `HtmlGenericControl` Class**
>
> **The `HtmlGenericControl` class is described in more detail in Chapter 1, "Web Forms Tips and Tricks," where it is used for another control type that is not part of the .NET Framework class library.**

The next line of code hides the `<form>` section of the page. Because this stage is a postback, the viewstate of the controls on the page is maintained, so the form will remain visible if you don't hide it. The final code line makes the section containing the "please wait" message visible.

Displaying the Results

Listing 3.3 shows the second section of the `Page_Load` event handler. This section is executed only if the `Page.IsPostback` property is `False`; however, you have to detect whether the page is being loaded by the `<meta>` element in the "please wait" page (Stage 2) or the hyperlink in the results page (Stage 3).

LISTING 3.3 The Second Part of the `Page_Load` Event Handler

```
...
Else

  ' get customer ID from query string
  Dim sCustID As String = Request.QueryString("custID")

  If sCustID > "" Then

    ' page is loading from META REFRESH element and
    ' so currently shows the "please wait" message
    ' a customer ID was provided so display results
    divResult.Visible = True
```

LISTING 3.3 Continued

```
      ' set URL for "Next Customer" hyperlink
      lnkNext.NavigateUrl = Request.FilePath

      ' get data and bind to DataGrid in the page
      FillDataGrid(sCustID)

    Else

      ' either this is the first time the page has been
      ' loaded, or no customer ID was provided
      ' display controls to select customer
      frmMain.Visible = True

    End If
  End If

End Sub
```

You've just seen how the code that runs in Stage 1, when the user submits the form, adds the customer ID to the query string as `custID=`*`value`*. (When the user loads the page by clicking the hyperlink in the results page, there will be no query string.) So you test for the presence of a customer ID value and, if there is one, you can make the section of the page that displays the results visible, set the URL of the hyperlink in that section of the page so that it will reload the current page, and then call a separate routine, named `FillDataGrid`, that calculates the results and fills the ASP.NET `DataGrid` control in this section of the page.

At the end of Listing 3.3 you can see the code that runs for Stage 3 of the process. In this case, you know that it's not a postback, and there is no customer ID in the query string. So either this is the first time the page has been accessed or the user did not enter a customer ID value in the text box. In either case, you just have to make the `<form>` section visible, and the user ends up back at Stage 1 of the process.

Viewstate and the `Visible` Property

Notice that because the page does not maintain viewstate for Stages 2 and 3, you don't need to hide the other sections of the page content. All three carry the `Visible="False"` attribute, so they will not be displayed unless you specifically change the `Visible` property to `True` when the page loads each time.

Populating the `DataGrid` Control

The only other code in the sample page is responsible for fetching the required data from the database and populating the `DataGrid` control on the page. The full or partial customer ID, extracted from the query string at Stage 2 of the process, is passed to the `FillDataGrid` routine, which is shown in full in Listing 3.4.

LISTING 3.4 The Final Part of the Page_Load Event Handler

```
Sub FillDataGrid(sCustID As String)

  Dim sSelect As String _
    = "SELECT CustomerID, CompanyName, City, Country, Phone " _
    & "FROM Customers WHERE CustomerID LIKE @CustomerID"
  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindSqlClientConnectString")
  Dim oConnect As New SqlConnection(sConnect)

  Try

    ' get DataReader for rows from Northwind Customers table
    Dim oCommand As New SqlCommand(sSelect, oConnect)
    oCommand.Parameters.Add("@CustomerID", sCustID & "%")
    oConnect.Open()
    dgrResult.DataSource = oCommand.ExecuteReader()
    dgrResult.DataBind()
    oConnect.Close()
    lblResult.Text = "Results of your query for Customer ID '" _
                   & sCustID & "'"

    ' force current thread to sleep for 3 seconds
    ' to simulate complex code execution
    Thread.Sleep(3000)

  Catch oErr As Exception

    oConnect.Close()
    lblResult.Text = oErr.Message

  End Try

End Sub
```

The code here is fairly conventional. It creates a parameterized SQL statement and then executes it with a `Command` instance to return a `DataReader` instance that points to the result set generated by the database. You use the customer ID passed to the routine as the value of the single `Parameter` instance you create, and the resulting `DataReader` instance is bound to the `DataGrid` control. See the section "Using Parameters with SQL Statements and Stored Procedures" in Chapter 10, "Relational Data-Handling Techniques," for more details on using parameterized SQL statements.

Simulating a Complex or Lengthy Process

The code used to populate the `DataGrid` control in this example is unlikely to qualify as a complex or lengthy operation. Unless someone pulls the network cable out, it won't take long enough for the user to see the "please wait" message in the demonstration page. So to simulate a long process, you can insert a call to the `Sleep` method of the static `Thread` object, specifying that the current thread should wait 3 seconds before continuing:

```
Thread.Sleep(3000)
```

The only point to watch for here is that you have to import the `System.Threading` namespace into the page to be able to access the `Thread` object:

```
<%@Import Namespace="System.Threading" %>
```

Displaying a Progress Bar Graphic

A static "please wait" message is fine, but it could not be described as eye-catching, and it gives no indication that anything is actually happening. The server could die while the message is being shown, leaving you still staring at the "please wait" message three days later. It's nice to have some kind of indication that the Web site is still alive and really is working furiously to generate the results you asked for.

Unfortunately, with the way that Web browsers and HTTP work, this isn't easy to achieve. Each request/response is treated as a single unit of operation, and there is no persistent connection over which status information can be passed. There are ways around this, of course, but they tend to hit performance and cause undue server loading. You'll see an example of this in the section "Implementing a Staged Page Load Process," later in this chapter.

An alternative is to display something that makes it look like the browser is working hard but actually bears no relationship to what's happening on the server. When you do this, you avoid the need for extra connections while the main process is taking place, and yet you still satisfy the user's desire to see something happening. The simplest solution is to use an animated GIF file in the page instead of or in addition to the "please wait" message.

Figure 3.6 shows the "please wait" page for this example. Instead of just a text message, you now also have a progress bar that appears to reflect the state of the long-running process that is generating the results the user is waiting for.

As intimated earlier, however, the progress bar is an illusion in that it will keep moving, regardless of whether the page takes a minute or a month to appear. But by carefully choosing the timing of the animation in the GIF file to match the anticipated average response times for average users, you can get it to look quite realistic.

FIGURE 3.6 Displaying a progress bar while loading another page.

> **Achieving True and Accurate Status Displays**
>
> To achieve a real page-loading status display, you can arrange for your server-side code to flush chunks of output to the client as it carries out the processing required to generate the results. These chunks of output could be client-side script that writes status details within the current browser page or even just simple `<img>` elements that load images to indicate progress of the operation. As an example, the MSN Expedia Web site (`www.expedia.com`) flushes partial page output to the browser, as you can see if you view the source of the page while it's searching for that holiday in Florida you keep promising your kids. However, it also uses a "dummy" animated graphic, just as this example does, which effectively indicates nothing about the actual underlying process of the operation.

Other than the appearance of the progress bar, the remainder of this example looks the same as the previous example, which displays just the "please wait" text message. Therefore, the following sections concentrate on what's different in the declaration of the HTML, the server controls, and the code used to drive this page compared to the previous example.

The Progress Bar Animated Graphic Files

We provide four different versions of the animated progress bar graphic in the `images` folder of the examples you can download for this book (from `www.daveandal.net/books/6744/`). The only difference between them is the speed at which the progress display moves from left to right. The details of these graphics files are summarized in Table 3.1.

TABLE 3.1

The Progress Bar Animated GIF Files for This Example

Filename	Description
`progressbar10.gif`	The indicator progresses at a steady speed from left to right in approximately 10 seconds, and it remains at the fully right (complete) position for 10 seconds before starting again.
`progressbar20.gif`	The indicator progresses at a steady speed from left to right in approximately 20 seconds, and it remains at the fully right (complete) position for 10 seconds before starting again.
`progressbar30.gif`	The indicator progresses at a steady speed from left to right in approximately 30 seconds, and it remains at the fully right (complete) position for 10 seconds before starting again.
`progressbarlog.gif`	The indicator progresses in logarithmic fashion from left to right in approximately 30 seconds, starting quickly and then getting slower. It remains at the fully right (complete) position for 10 seconds before starting again.

Displaying the Progress Bar Graphic

In theory, building the progress bar sample page should be easy. You just have to insert an `<img>` element into the section of the page that is displayed for Stage 2 of the process in the previous example, and you're done, right? However, most Web developers approach these trivial tasks with trepidation and with a knowledge gleaned from experience that nothing ever works quite as you expect when dealing with Web browsers—especially Web browsers from different manufacturers.

It turns out that trepidation is definitely justified here. Simply adding an `<img>` element fails to work properly because as soon as the redirection is initiated by the `<meta>` element, most browsers stop loading any images for the current page. In this case, unless the progress bar is already cached (and the server is extremely responsive when the browser checks whether the file has changed since it was cached), the result is a "missing image" placeholder instead of a progress bar.

The solution to this problem is to force the browser to delay for a few seconds—long enough to load the progress bar graphic—before beginning the refresh process that requests the next page. You can set this delay to 3 seconds in the sample page by changing the `content` attribute you add to the `<meta>` element in the `Page_Load` event handler:

```
mtaRefresh.Attributes.Add("content", "3;url=" & sRefreshURL)
```

Now the page works fine in recent Netscape, Mozilla, and Opera browsers. But it still doesn't work properly in Internet Explorer. It seems that Internet Explorer "turns off" the animation in animated GIF files as soon as a new page is requested. After the 3-second delay, the progress bar just stalls—which ruins the whole effect! So, for Internet Explorer, you have to find an alternative approach, as described in the following sections.

An Alternative Page-Loading Technique for Internet Explorer

We experimented with several seemingly obvious approaches to loading the progress bar graphic and reloading the page using client-side script in Internet Explorer, all to no avail. It seems that the only way to circumvent the issue with the stalled animated graphic is to find a completely different way to load the next page (that is, reload the current page with the customer ID in the query string).

Internet Explorer 5 and higher have access to the MSXML parser component; it is part of a Windows installation and is distributed with Internet Explorer as well. Part of the MSXML parser component is an object named `XMLHTTP`, which you can use to request a resource from the server in the background while a page is loaded and displayed in the browser.

The `XMLHTTP` object is instantiated and manipulated with client-side script within a Web page, and it exposes properties and methods that allow you to make `GET` and `POST` requests to a server both synchronously and asynchronously. Although it is ostensibly designed for fetching XML documents, it works equally well fetching any type of resource, including HTML pages that probably aren't fully XML (or XHTML) compliant.

Loading Pages with the `XMLHTTP` Object

The process for using the `XMLHTTP` object is relatively simple, especially if you are happy to load the new page synchronously. You can create an instance of the `XMLHTTP` object by using the following:

```
var oHTTP = new ActiveXObject("Microsoft.XMLHTTP");
```

Next you open an HTTP connection, specifying the HTTP method (usually `"GET"` or `"POST"`), the URL of the target resource, and the value `false` to indicate that you want synchronous operation. Then you can use the `send` method to send the request:

```
oHTTP.open("method", target-url, false);
oHTTP.send();
```

After the response has been received from the server, you test the `status` property (the value of the HTTP status header) to see if it is `200` (which means `"OK"`) and extract the page as a string from the `XMLHTTP` object by using the following:

```
if (oHTTP.status == 200)
  sResult = oHTTP.responseText;
else
  // an error occurred
```

However, if you use synchronous loading, the browser will not respond to any other events (including animating the GIF file) while the request for the next page is executing. Instead, you need to use asynchronous loading to allow the browser to carry on reacting as normal while the server creates and returns the new page.

Asynchronous Loading with the `XMLHTTP` Object

For asynchronous loading, you first have to specify the name of a callback function that will be executed each time the `readystate` property of the `XMLHTTP` object changes and specify `true` for the third parameter of the `open` method:

```
oHTTP.onreadystatechange = myCallbackHandler;
oHTTP.open("method", target-url, true);
oHTTP.send();
```

The callback function you specify will be executed several times as the `XMLHTTP` object fetches the response from the server. When the response is complete, the value of the `readystate` property will be `4`, and at that point you can test for an error and extract the page as a string:

```
function myCallbackHandler () {
  if (oHTTP.readyState == 4) {
    if (oHTTP.status == 200)
      sResult = oHTTP.responseText;
    else
      // an error occurred
  }
}
```

Using the XMLHTTP Object in the Progress Bar Sample Page

Listing 3.5 shows the client-side code included in the progress bar sample page. It works exactly as just demonstrated, with the only additions being a test to see that an instance of the XMLHTTP object was successfully created and the display of any error messages in a <span> element, located below the progress bar graphic in the page.

> **Information on the XMLHTTP Object**
>
> **You can find a full reference to the XMLHTTP object (effectively the XMLHTTPRequest interface) in the MSDN library, at http://msdn.microsoft.com/library/en-us/xmlsdk30/htm/xmobjxmlhttprequest.asp.**

LISTING 3.5 Loading the Results Page with XMLHTTP

```
<script language='javascript'>
<!--
// variable to hold reference to XMLHTTP object
var oHTTP;

function loadTarget(sURL) {
  // create instance of a new XMLHTTP object
  oHTTP = new ActiveXObject("Microsoft.XMLHTTP");
  if (oHTTP != null) {
    // specify callback for loading completion
    oHTTP.onreadystatechange = gotTarget;
    // open HTTP connection and send async request
    oHTTP.open('GET', sURL, true);
    oHTTP.send();
  }
  else {
    document.all['spnError'].innerText
      = 'ERROR: Cannot create XMLHTTP object to load next page';
  }
}

function gotTarget() {
  // see if loading is complete
  if (oHTTP.readyState == 4) {
    // check if there was an error
    if (oHTTP.status == 200) {
      // dump next page content into this page
      document.write(oHTTP.responseText);
    }
    else {
      document.all['spnError'].innerText
        = 'ERROR: Cannot load next page';
```

LISTING 3.5 Continued

```
    }
  }
}
//-->
```

One interesting point about this listing is in the `gotTarget` callback handler. After you've extracted the complete content of the new page as a string, you simply write it into the current browser window, using the client-side `document.write` method. This replaces the current content, giving the same output as in the first example in this chapter, after the main customer lookup process has completed (refer to Figure 3.5).

What you've actually achieved here is to reload the same page again in the background, while still at Stage 2 of the process (displaying the "please wait" message and progress bar) and then use it to replace the current page. But because the URL you request contains the customer ID in the query string this time, the new page generated by the server will be the one for Stage 3 of the process (containing the `DataGrid` control, populated with the results of the database search). Altogether, this is a neat and interesting solution!

The Changes to the HTML and Server Control Declarations in This Example

The only remaining features of this example that we need to examine are how to initiate the client-side code that loads the results page and how to handle cases where client-side scripting is disabled in the browser. In the HTML section of the page, you declare the `<body>` element as a server control this time, by adding an ID and the `runat="server"` attribute—just as you did for the `<meta>` element earlier in this chapter:

```
<body id="tagBody" runat="server">
```

Then, in the `Page_Load` event handler, you can add an appropriate `onload` attribute to the opening `<body>` tag in the server-side code. Listing 3.6 shows the changed section of the `Page_Load` event handler. The only section that differs in this example from the first example is the part where the postback from Stage 1 occurs—where you are generating the "please wait" page for Stage 2 of the process.

LISTING 3.6 The `Page_Load` Event Handler for the Progress Bar Example

```
If Page.IsPostback Then

  Dim sRefreshURL As String = Request.Url.ToString() _
    & "?custID=" & txtCustomer.Text

  ' if it's IE, need to load new page using script because
  ' the META REFRESH prevents the animated GIF working
  If Request.Browser.Browser = "IE" Then
    tagBody.Attributes.Add("onload", "loadTarget('" _
                                   & sRefreshURL & "');")
```

LISTING 3.6 Continued

```
    ' set META REFRESH as well in case script is disabled
    ' use long delay so script can load page first if possible
    mtaRefresh.Attributes.Add("http-equiv", "refresh")
    mtaRefresh.Attributes.Add("content", "30;url=" & sRefreshURL)

  Else

    ' not IE so use META REFRESH to start loading next page
    ' allow 3 seconds for progress bar image to load
    mtaRefresh.Attributes.Add("http-equiv", "refresh")
    mtaRefresh.Attributes.Add("content", "3;url=" & sRefreshURL)

  End If

  frmMain.Visible = False
  divWait.Visible = True

Else
...
```

You use the ASP.NET `Request.Browser` object, which exposes a property also named (rather confusingly) `Browser`. This property indicates the browser type, and if it is `"IE"`, you know that you are serving to an Internet Explorer browser—so we can add the `onload` attribute to the `<body>` element by using the `Attributes` collection of the `HtmlGenericControl` class that implements it in ASP.NET. The result, when viewed in the browser, looks like this:

```
<body id="tagBody" onload="loadTarget('/daveandal/books/6744
➥/loadwait/progressbar.aspx?custID=a');">
```

You also add a "catch all" feature in case scripting is disabled, by setting the attributes of the `<meta>` element. In this case, the `<meta>` element will cause a page reload after 30 seconds. You can also see in Listing 3.6 the changed value of the `content` attribute that you apply for non–Internet Explorer browsers, to allow the progress bar graphic to load before the redirection commences (as discussed earlier in this chapter).

> **Checking for the Version of Internet Explorer**
>
> **In theory, you should test for the browser *version* as well as the *type* because the `XMLHTTP` object is available only in version 5 and higher of Internet Explorer. However, the "catch all" you build in for when scripting is disabled will also make the page work (after a fashion) on earlier versions of Internet Explorer. Whether anyone is still using version 4 or earlier, with all the security issues inherent in those versions, is open to discussion.**

Implementing a Staged Page Load Process

We hinted earlier in this chapter that there are ways you can generate "real" status messages in the browser while executing a complex or lengthy operation on the server. Although the technique of simply flushing chunks of content back to the browser as the process runs does work, it's not particularly efficient in terms of connection usage or server loading.

Web servers are designed to receive a connection and resource request, generate the required response, and disconnect as quickly as possible to allow the next user to connect and make a resource request. Because it's likely that most complex operations will involve database access on the server, holding open a connection to the database while you flush chunks of content back to the client is probably not a good idea.

> **Flushing Intermediate Content to the Client**
>
> **Of course, if the process has to access several different data sources to generate the resulting page, as is most likely the case with the MSN Expedia example mentioned earlier in this chapter, you can flush the individual chunks of "status" content to the browser in between opening each connection, extracting the data, and closing it again.**

However, if you can break down the complex or lengthy process into separate individual stages, it is possible to provide useful "real" status feedback in the browser. In fact, it's reasonably easy to do this in Internet Explorer 5 and higher, by using the `XMLHTTP` object used in the previous example.

The Steps in Implementing a Staged Page Load Process

Figure 3.7 shows a flowchart of a staged process that is implemented as the next example in this chapter. The main page, named `stagedloading.aspx`, uses the `XMLHTTP` component to request a separate operation page, named `stagedfetchpage.aspx`, four times. Each request contains, in the query string, a customer ID that the user provides and a step value that indicates which stage of the process is currently being performed. The operation page uses these values to collect the appropriate row set from the Northwind database at each stage and add to a `DataSet` instance a table that is stored in the user's ASP.NET session.

In between requests, the main page can display progress and status information, or it can display any error messages returned by the operation page. When the process is complete in this example, the value returned (the total for all matching orders) is displayed—together with a button that allows the user to view the list of orders. This data is in the `DataSet` instance stored in the user's ASP.NET session, so it can be extracted and displayed without requiring another trip to the database.

Of course, you can easily tailor this example to display different data at any stage and provide links to access any of the tables in the `DataSet` instance. In fact, this process opens up a whole realm of opportunities for collecting data of all kinds and combining and then querying it afterward. Figure 3.8 shows a screenshot of the sample page while it is collecting details of orders for all customers whose ID starts with *m* and building up the `DataSet` instance.

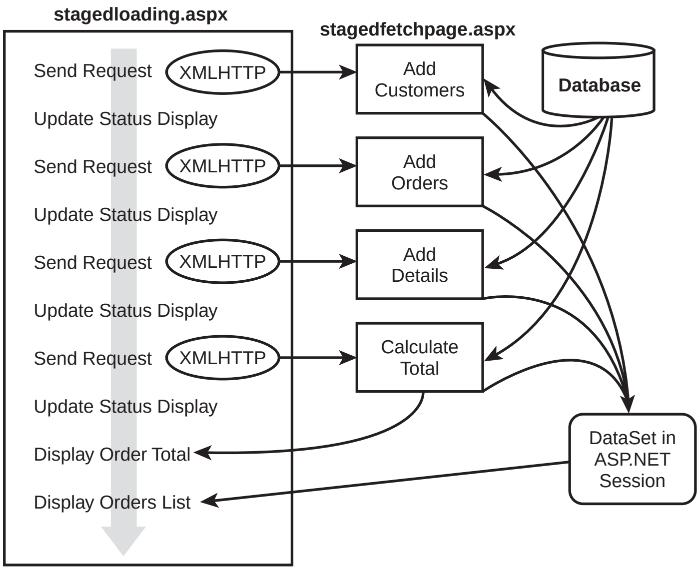


FIGURE 3.7 A flowchart of the steps in implementing a staged page load process.

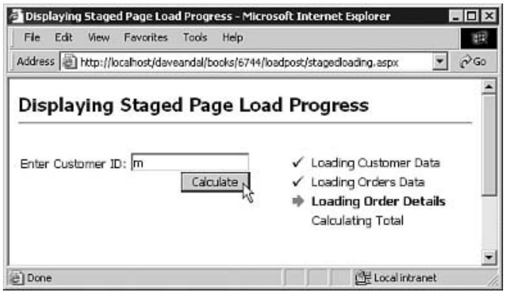


FIGURE 3.8 The staged processing and reporting sample page in action.

You'll learn about this page in more detail shortly, but first you need to see how you can pass status and other information back to the XMLHTTP object. Then you'll see how the operation page, which collects the data and stores it in the user's session, works. After that, you'll see how the main page calls this operation page and how it displays the status information and results.

Status Information in ASP.NET and the XMLHTTP Object

When a browser or any other client (such as XMLHTTP) requests an HTML page, the server

Accessing Physically or Geographically Separated Data Sources

The set of steps used in this example could easily be performed in one pass. However, using separate stages demonstrates how you could in a more complex scenario access multiple different data sources that could be physically and geographically separated. These data sources might be Web services, XML documents, or other types of data sources—and not just relational databases. For instance, take the MSN Expedia example mentioned earlier: It's likely that the data sources being accessed would be hosted by different airlines, hotels, rental car companies, and so on.

returns an HTTP status header, followed by the page that was requested. If there is no error (that is, the page can be found and executed by the server), it returns the status header `"200 OK"`.

However, even if the process of loading and executing the page succeeds, you can still control the status code that is returned by setting the `Status`, `StatusCode`, and/or `StatusDescription` properties of the current ASP.NET `Response` object. The values of these properties will be exposed by the `status` and `statusText` properties of the `XMLHTTP` object after it loads the page (see Table 3.2). You can find a full list of the standard HTTP status codes at `www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

TABLE 3.2

The Equivalent Status-Related Properties of the ASP.NET `Response` and `XMLHTTP` Objects

ASP.NET `Response` Object Property	`XMLHTTP` Object Property	Description
`Status`	No direct equivalent	A combination of the status code and status description (for example, `"200 OK"` or `"302 Object Moved"`)
`StatusCode`	`status`	The numeric part of the status information (for example, `200` or `302`)
`StatusDescription`	`statusText`	The text or description part of the status information (for example, `"OK"` or `"Object Moved"`)

By default, the server will automatically set the ASP.NET `Status` property to `"200 OK"` if there is no error or to the standard HTTP status code for any error that does occur (for example, `"500 Internal Server Error"` if there is an ASP.NET code execution error). However, if you trap ASP.NET errors in the code—for example, a failed database connection or a numeric calculation error—you must set the `Status` property (or the `StatusCode` and `StatusDescription` properties) if an error does occur.

The Staged Process Operation Page

The main page that the user sees makes repeated requests to the operation page (`stagedfetchpage.aspx`), passing the customer ID and the appropriate step number each time. Because it does this by using the `XMLHTTP` component, the operation page doesn't have to generate any HTML or output. All it has to do is indicate to the main page whether there was an error or whether this step of process succeeded.

However, not all the values you pass back to the `XMLHTTP` object in this example are strictly status messages; for example, the order value total that is displayed at the end of the process must be returned to the main page. So rather than use the `StatusDescription` property (`statusText` in `XMLHTTP`), you can write these messages directly into the page that is returned. The `XMLHTTP` object can retrieve this as the `responseText` property, as shown in the previous example.

The `Page_Load` Event Handler for the Staged Loading Example

Listing 3.7 shows the `Page_Load` event handler in the operation page, together with the page-level variable that holds a reference to the `DataSet` instance stored in the session. The values for the customer ID and the current step are collected from the query string each time the page loads.

LISTING 3.7 The Page_Load Event Handler for the Staged Loading Example

```
Dim oDS As DataSet

Sub Page_Load()

  Dim sCustID As String = Request.QueryString("custID")
  Dim sStep As String = Request.QueryString("step")
  Dim sSelect As String

  ' force current thread to sleep for 3 seconds
  ' to simulate complex code execution
  Thread.Sleep(3000)

  Select Case sStep
    Case "1"
      oDS = New DataSet()
      sSelect = "SELECT CustomerID, CompanyName, City, " _
        & "Country, Phone FROM Customers " _
        & "WHERE CustomerID LIKE @CustomerID"
      AddTable("Customers", sCustID, sSelect)
    Case "2"
      oDS = CType(Session("thedata"), DataSet)
      sSelect = "SELECT OrderID, OrderDate FROM Orders " _
        & "WHERE CustomerID LIKE @CustomerID"
      AddTable("Orders", sCustID, sSelect)
    Case "3"
      oDS = CType(Session("thedata"), DataSet)
      sSelect = "SELECT [Order Details].OrderID, " _
        & "Products.ProductID, Products.ProductName, " _
        & "[Order Details].Quantity, [Order Details].UnitPrice " _
        & "FROM [Order Details] JOIN Products " _
        & "ON [Order Details].ProductID = Products.ProductID " _
        & "WHERE [Order Details].OrderID IN " _
        & " (SELECT OrderID FROM Orders " _
        & "  WHERE CustomerID LIKE @CustomerID)"
      AddTable("OrderDetails", sCustID, sSelect)
    Case "4"
      oDS = CType(Session("thedata"), DataSet)
      CalculateTotal()
    Case Else
      Response.Status = "500 Internal Server Error"
      Response.Write("Error: Invalid Query String Parameter")
  End Select

End Sub
```

Accessing the Customer ID Value

The value of the customer ID entered into the text box cannot be extracted directly as the `Text` property of the ASP.NET `TextBox` control when this page is executed. The page is loaded with the `"GET"` method by the `XMLHTTP` object, with the customer ID appended to the query string, so it must be collected from there each time.

What Happens if Cookies Are Disabled?

The sample page will fail to work properly if the user has cookies disabled in his or her browser because ASP.NET will not be able to maintain a user session. One solution would be to enable cookieless sessions by adding the element `<sessionState cookieless= "true" />` to the `<system.web>` section of the `web.config` file for the application. In this case, you must also modify the `src` attribute of the non-server control `<img>` elements to specify the full path to the images because the inclusion of the session key in the page URL breaks the links to images that are specified only as relative paths from the URL of the page that hosts them.

Next, to simulate a long process, you force the current thread to sleep for 3 seconds (as you did in the "please wait" example) before using the step value from the query string to decide which action the page will carry out. The first three stages of the operation must create and execute a database query to extract the appropriate set of rows and then add these to the `DataSet` instance in the user's session. The `AddTable` routine, which you'll see shortly, achieves this. Obviously, you have to a create new `DataSet` instance at Stage 1, but the remaining stages can extract this `DataSet` instance from the user's session.

At Stage 4 in this example, the operation page has to calculate the order total and return it to the main page, using the routine `CalculateTotal` (which you'll see shortly). Any value greater than 4 for the step parameter is treated as an error, and the page returns the server-side execution error `"500 Internal Server Error"`. A more detailed error message is also sent back as the content of the returned page.

Adding Tables to the `DataSet` Instance

Adding a table to the `DataSet` instance you extract from the user's session is simple, and the code in Listing 3.8 demonstrates the traditional techniques you use. Notice that, in this code, you check whether you actually managed to find a `DataSet` instance in the session, and you return an error status and message if not. After adding the table, you push the updated `DataSet` instance back into the session. If there is an error while extracting the rows, a suitable error status and message are returned to the user instead.

LISTING 3.8 The `AddTable` Routine for the Staged Loading Example

```
Sub AddTable(sTableName As String, sCustID As String, _
             sSelect As String)

  If oDS Is Nothing Then

    Response.Status = "500 Internal Server Error"
    Response.Write("Error: Cannot access DataSet in session")

  Else
```

LISTING 3.8 Continued

```
    Dim sConnect As String = ConfigurationSettings.AppSettings( _
                             "NorthwindSqlClientConnectString")
    Dim oConnect As New SqlConnection(sConnect)
    Dim oDA As New SqlDataAdapter(sSelect, oConnect)
    oDA.SelectCommand.Parameters.Add("@CustomerID", sCustID & "%")

    Try

      ' fill table in DataSet and put back into session
      oDA.Fill(oDS, sTableName)
      Session("thedata") = oDS
      Response.Status = "200 OK"
      Response.Write("OK")

    Catch oErr As Exception

      Response.Status = "500 Internal Server Error"
      Response.Write("Error: " & oErr.Message)

    End Try

  End If

End Sub
```

Calculating the Total Value of the Orders

The final section of the operation page in the staged loading example is shown in Listing 3.9. This simply references the `OrderDetails` table in the `DataSet` instance and sums the values in each row by multiplying the quantity by the unit price. The result is written back to the response as a fixed-point number with two decimal places.

LISTING 3.9 The `CalculateTotal` Routine for the Staged Loading Example

```
Sub CalculateTotal()

  Dim dTotal As Decimal = 0

  Try

    For Each oRow As DataRow In oDS.Tables("OrderDetails").Rows
      dTotal += (oRow("Quantity") * oRow("UnitPrice"))
    Next

    Response.Status = "200 OK"
```

LISTING 3.9 Continued

```
    Response.Write(dTotal.ToString("F2"))

  Catch oErr As Exception

    Response.Status = "500 Internal Server Error"
    Response.Write("Error: " & oErr.Message)

  End Try

End Sub
```

The Staged Process Main Page in the Staged Loading Example

Now that you have seen how the operation page performs the updates to the `DataSet` instance and returns status and information messages, you can now look at the main page that calls this operation page at each stage of the overall process. Listing 3.10 shows the HTML content of the main page. You can see that there is an ASP.NET `TextBox` control for the user to enter the full or partial customer ID and an `<input>` element that creates the submit button captioned Calculate.

LISTING 3.10 The HTML Declarations for the Main Page in the Staged Loading Example

```
<form runat="server">

  <!----- form for selecting customer ----->
  <asp:Label id="lblEnter" runat="server"
             Text="Enter Customer ID:" />
  <asp:Textbox id="txtCustomer" runat="server" /><br />
  <input id="btnGo" type="submit" value="Calculate"
         onclick="return getResults();" runat="server"/>

  <!----- "please wait" display ----->
  <table border="0">
  <tr>
    <td><img id="img1" src="../images/False.gif" width="12"
             height="12" hspace="5" /></td>
    <td><span id="spn1">Loading Customer Data</span></td>
  </tr><tr>
    <td><img id="img2" src="../images/False.gif" width="12"
             height="12" hspace="5" /></td>
    <td><span id="spn2">Loading Orders Data</span></td>
  </tr><tr>
    <td><img id="img3" src="../images/False.gif" width="12"
             height="12" hspace="5" /></td>
    <td><span id="spn3">Loading Order Details</span></td>
```

LISTING 3.10 Continued

```
  </tr><tr>
    <td><img id="img4" src="../images/False.gif" width="12"
             height="12" hspace="5" /></td>
    <td><span id="spn4">Calculating Total</span></td>
  </tr>
  </table>

  <!----- section for displaying total ----->
  <div id="divResult">
    <b><span id="spnResult"></span></b><p />
  </div>

  <!----- section for displaying orders ----->
  <div id="divOrderList">
    <asp:Button id="btnOrders" style="visibility:hidden"
         Text="Show Orders" OnClick="ShowOrders" runat="server" />
    <asp:DataGrid id="dgrOrders" EnableViewState="False"
                  runat="server" /><p />
  </div>

</form>

<img id="imgTrue" style="visibility:hidden"
     src="../images/True.gif" />
<img id="imgThis" style="visibility:hidden"
     src="../images/This.gif" />
```

You use the HTML `<input>` element here because this is easier to connect to a client-side event handler than the ASP.NET `Button` element. (You don't have to add the `onclick` attribute on the server via the `Attributes` collection.) You always return `false` from the event handler that is attached to this button because you must prevent it from submitting the page to the server.

The HTML table that follows the text box and button contains an `<img>` element and a `<span>` element for each stage of the process. The client-side code that executes the operation page will update the `src` attribute of the `<img>` element to change the image that is displayed and the `font-weight` style selector of the text as each stage takes place.

Declaring the Button as a Server Control

You could omit the `runat="server"` attribute from the button. This would mean that the `<input>` element would not be a server control. However, you want to be able to hide the button if the browser is not Internet Explorer 5 or higher, and, because you perform this check on the server side when the page loads (as you'll see shortly), you need to be able to reference it in the server-side code.

You could also use the HTML `<button>` element instead of the `<input>` element. The `<button>` element is not supported in all browsers, but because this page will work only in Internet Explorer (where it is supported), this would not be an issue.

The other two sections of the page are a `<div>` section, where any error messages and the final order total will be displayed as each stage of the process executes, and another `<div>` section, where the list of orders is displayed if the user clicks the Show Orders button. You'll learn about this aspect of the sample page after you see how it performs the initial four stages of calculating the order total.

Finally, right at the end of the page are two more `<img>` elements that are hidden from view with the `visibility:hidden` style selector. You use these to preload the images for the list of operation stages. You display the image named `This.gif` (a right-pointing arrow) for each stage as it starts and then replace it with the image `True.gif` (a large check mark) if it completes successfully. You can see these two images in Figure 3.8.

Displaying the Current Operation Progress in the Staged Loading Example

Listing 3.11 shows the two client-side JavaScript functions you use to manipulate the progress indicators in the page. As each stage of the process is started, you make a call to the `setCurrent` function. As each stage completes, you call the `setCompleted` function. In both cases, you supply the stage number (a value from 1 to 4 in this example) as the single parameter.

LISTING 3.11 The Client-Side Routines to Display Operation Progress in the Staged Loading Example

```
function setCurrent(iStep) {
  // get reference to image and change to "arrow"
  // using image pre-loaded in hidden <img> element
  var oImg = document.getElementById('imgThis');
  var oElem = document.getElementById('img' + iStep.toString());
  oElem.src = oImg.src;
  // get reference to span and change text to bold
  oElem = document.getElementById('spn' + iStep.toString());
  oElem.style.fontWeight = 'bold';
}

function setCompleted(iStep) {
  // get reference to image and change to "tick"
  // using image pre-loaded in hidden <img> element
  var oImg = document.getElementById('imgTrue');
  var oElem = document.getElementById('img' + iStep.toString());
  oElem.src = oImg.src;
  // get reference to span and change text back to normal
  oElem = document.getElementById('spn' + iStep.toString());
  oElem.style.fontWeight = '';
}
```

The code in the `setCurrent` and `setCompleted` functions is very similar. It starts by getting a reference to the preloaded and hidden `<img>` element that contains either the arrow image (`This.gif`) or the check mark image (`True.gif`).

The <img> and <span> elements that indicate the four process stages shown in the page have values for their id attributes that indicate which stages they apply to. For example, the first stage uses the id attributes "img1" and "spn1", respectively, for the <img> and <span> elements. So the code can get references to the correct elements by using the step number passed to it as a parameter.

With these references, it's then just a matter of updating the src property of the <img> element to display the appropriate image and setting the style.fontWeight property of the <span> element.

Executing the Operation Page with XMLHTTP

Listing 3.12 shows the code that executes the operation page discussed earlier in this chapter. Three page-level variables are declared to hold references to items that will be accessed from separate functions: the <span> element, where the status and any error messages are displayed, the XMLHTTP object, and the customer ID that the user entered.

LISTING 3.12 The Client-Side Routines to Execute the Operation Page

```
var oResult;
var oHTTP;
var sCustID;

function getResults() {
  // get reference to "result" label and texbox value
  oResult = document.getElementById('spnResult');
  var oTextbox = document.getElementById('txtCustomer');
  sCustID = oTextbox.value;
  if (! sCustID == '') {
    // hide DataGrid control
    var oElem = document.getElementById('dgrOrders');
    if (oElem != null) oElem.style.visibility = 'hidden';
    // get Customers data
    fetchData(1)
  }
  else
    oResult.innerText = 'No customer ID specified';
  // return false to prevent button from submitting form
  return false;
}

function fetchData(iStep) {
  // create instance of a new XMLHTTP object because we
  // can't change readystate handler on existing instance
  oHTTP = new ActiveXObject('Microsoft.XMLHTTP');
  if (oHTTP != null) {
    // update status display and build data page URL
```

LISTING 3.12 Continued

```
    setCurrent(iStep);
    var sURL = 'stagedfetchpage.aspx?custid=' + sCustID
             + '&step=' + iStep.toString();
    // set correct handler for XMLHTTP instance
    switch (iStep) {
      case 1: {
        oHTTP.onreadystatechange = gotCustomers;
        break;
        }
      case 2: {
        oHTTP.onreadystatechange = gotOrders;
        break;
        }
      case 3: {
        oHTTP.onreadystatechange = gotDetails;
        break;
        }
      case 4: {
        oHTTP.onreadystatechange = gotTotal;
        }
    }
    // open HTTP connection and send async request
    oHTTP.open('GET', sURL, true);
    oHTTP.send()
  }
  else
    oResult.innerText = 'Cannot create XMLHTTP object';
}
```

Next comes the main `getResults` function, which is executed when the Calculate button is clicked. It collects a reference to the `<span>` element that will hold the results, along with the customer ID that the user entered into the text box on the page. If there is a value here, it hides the `DataGrid` control that could still be displaying the list of orders from a previous query, and then it calls the `fetchData` function with the parameter set to `1` to perform Stage 1 of the process. If there is no customer ID, it just displays an error message instead.

The `fetchData` function (also shown in Listing 3.12) will be called at each stage of the process, starting—as you've just seen—with Stage 1. This function's task is to create an instance of the `XMLHTTP` object and execute the operation page with the correct combination of values in the query string. It first checks that an instance of `XMLHTTP` was in fact created, and then it calls the `setCurrent` function shown in Listing 3.11 to update the status display in the page. Then it creates the appropriate URL and query string for this stage of the process.

However, recall that you have to access the operation page asynchronously to allow the main page to update the status information, so you must specify a client-side event handler for the readystatechange event of the XMLHTTP object. The page contains four event handlers, and you select the appropriate one by using a switch statement before opening the HTTP connection and calling the send method of the XMLHTTP object to execute the operation page.

Handling the XMLHTTP readystatechange Events

Listing 3.13 shows the four event handlers that are declared in the switch statement in Listing 3.12. They are all very similar, and by looking at the first of them, gotCustomers, you can see that they do nothing until the loading of the operation page is complete (when the readystate property is 4). Then, if the status code returned from the operation page is 200 ("OK"), they call the setCompleted function shown in Listing 3.11 to indicate that this stage completed successfully. If any other status code is returned, the code displays the value of the responseText property (the content of the page returned, which will be the error details) in the page.

LISTING 3.13 The Event Handlers for the XMLHTTP readystatechange Event

```
function gotCustomers() {
  // see if loading is complete
  if (oHTTP.readyState == 4) {
    // check if there was an error
    if (oHTTP.status == 200) {
      // update status display and fetch next set of results
      setCompleted(1);
      fetchData(2);
    }
    else
      oResult.innerText = oHTTP.responseText;
  }
}

function gotOrders() {
  // see if loading is complete
  if (oHTTP.readyState == 4) {
    // check if there was an error
    if (oHTTP.status == 200) {
      // update status display and fetch next set of results
      setCompleted(2);
      fetchData(3);
    }
    else
      oResult.innerText = oHTTP.responseText;
  }
}

function gotDetails() {
  // see if loading is complete
  if (oHTTP.readyState == 4) {
```

LISTING 3.13 Continued

```
    // check if there was an error
    if (oHTTP.status == 200) {
      // update status display and fetch next set of results
      setCompleted(3);
      fetchData(4);
    }
    else
      oResult.innerText = oHTTP.responseText;
  }
}

function gotTotal() {
  // see if loading is complete
  if (oHTTP.readyState == 4) {
    // check if there was an error
    if (oHTTP.status == 200) {
      // update status display
      setCompleted(4);
      // display result in page and show Orders button
      oResult.innerText = 'Total value of all orders $ '
                        + oHTTP.responseText;
      var oElem = document.getElementById('btnOrders');
      oElem.style.visibility = 'visible';
    }
    else
      oResult.innerText = oHTTP.responseText;
  }
}
```

As each stage completes, the code must initiate the next stage. In the first three event handlers (shown in Listing 3.13), this just involves calling the `fetchData` function (shown in Listing 3.12) again—but with the next stage number as the parameter. The instance of the `XMLHTTP` object that is created will then have the event handler for the next stage attached to the `readystatechange` event.

At Stage 4, when the `gotTotal` function is called after the operation page has successfully calculated and returned the total value of matching orders, the `responseText` property will return the total as a string. The function displays this value in the page and then changes the `visibility` style selector of the Show Orders button to make it visible. However, if there is an error, the error message is displayed instead.

Figure 3.9 shows the sample page after the four steps have completed successfully. You can see that the order total is displayed and the Show Orders button is now visible as well.

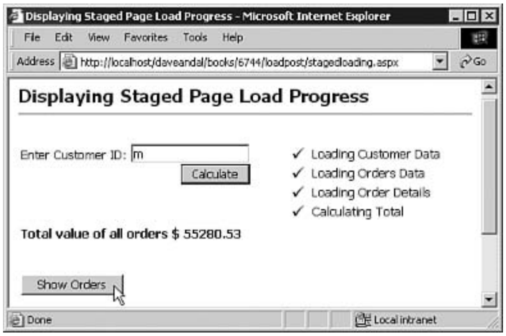


FIGURE 3.9 The sample page, after successfully processing all the stages.

Fetching and Displaying a List of Orders

After the four stages of the process in the staged loading example have completed successfully, the user's session contains a DataSet instance that is fully populated with lists of matching customers, orders, and order details rows from the database. This means that you can easily display some or all of the results of the four-stage process (as well as the total already displayed in the page) by querying this DataSet instance—without having to hit the database again.

> **Why Do the Check Mark Images Disappear?**
>
> **Notice that the check mark images disappear from the page following the postback that populates the `DataSet` instance. Remember that unlike changes made in server-side ASP.NET code, any changes made to the page displayed in the browser using client-side script are not persisted across postbacks.**

The Show Orders button (refer to Figure 3.9), which appears only after all four stages of the operation are complete, runs a server-side routine that extracts a list of order lines from the DataSet instance and displays them in the DataGrid control included in the HTML declarations of the page. Figure 3.10 shows the result.

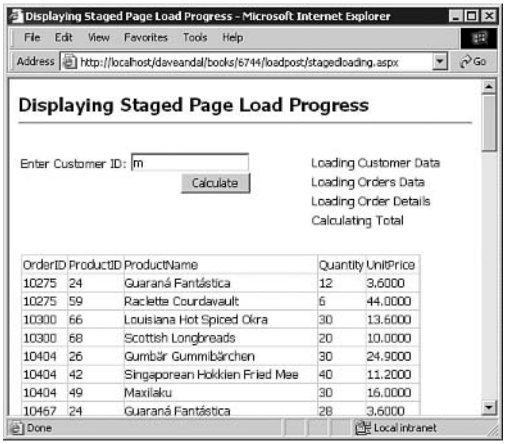


FIGURE 3.10 The sample page, displaying the list of orders from the cached DataSet instance.

The Server-Side Code in the Staged Process Main Page

Most of the action in the main page in the staged loading example is the result of the client-side script examined in the previous section. However, two tasks require server-side code. Because the page will work only in Internet Explorer 5 and higher, you really should make some attempt to test the browser type and display an error message in other browsers. Second, you need to handle `click` events for the Show Orders button and populate the `DataGrid` control that displays the list of order lines.

Listing 3.14 shows the complete server-side code for the main page. In the `Page_Load` event, you can access the `BrowserCapabilities` object that is exposed by the `Request.Browser` property and test the browser name and version. If the browser is not Internet Explorer 5 or higher, you display an error message and hide the text box and Calculate button so that the page cannot be used.

LISTING 3.14 The Server-Side `Page_Load` and `ShowOrders` Event Handlers

```
Sub Page_Load()
  ' check that the browser is IE 5 or higher
  If Request.Browser.Browser <> "IE" _
  Or Request.Browser.MajorVersion < 5 Then
    ' display message and hide input controls
    lblEnter.Text = "Sorry, this page requires Internet Explorer 5 or higher"
    txtCustomer.Visible = False
    btnGo.Visible = False
  End If
End Sub

Sub ShowOrders(sender As Object, args As EventArgs)

  ' bind DataGrid to contents of DataSet in user's Session
  dgrOrders.DataSource = CType(Session("thedata"), DataSet)
  dgrOrders.Datamember = "OrderDetails"
  dgrOrders.DataBind()

End Sub
```

When the Show Orders button is clicked (after the four stages of the process in the sample page are complete), the routine named `ShowOrders` is executed. This simply accesses the `DataSet` instance stored in the user's session, binds the `OrderDetails` table to the `DataGrid` control, and calls the `DataBind` method.

Catching and Displaying Errors from the Operation Page

The code shown in the preceding sections is designed to cope with any errors that might occur in the operation page, which does the real work of querying the database and building up the `DataSet` instance that contains all the results. As with any database operation, there is a possibility that something will go wrong—from a failed connection to changed permissions within the tables, changed column names, or even network failure if the database server is remote from the Web server.

As you've seen, the operation page returns one of the standard HTTP status codes each time, and it writes output into the page it generates. This content consists of just the text `"OK"` for the first three stages (where the `DataSet` instance is being created), but `this text` is not displayed in the main page. However, if there is an error within the operation page, the `XMLHTTP` object detects it because the status code is not `200`, and it displays the contents of the returned page.

As an example, if you change the SQL statement used for Stage 3 (extracting the order details) so that it references a non-existent column in the database, the `Try...Catch` construct in the operation page code (refer to Listing 3.8) catches the error. It returns the status code `"500 Internal Server Error"` and the text `"Error:"`, followed by the error message (as returned by ASP.NET when the data access operation failed) as the content of the page. The client-side code then displays the returned page content, as shown in Figure 3.11.

Making the Staged Process Work in Other Browsers

The staged loading example absolutely requires that the MSXML parser be available on the client and so it works only in Internet Explorer 5 and higher. However, it could be implemented in other browsers (and different types of clients), using other suitable client-side software components. There are Java applets available that could be used in other browsers, or you could create your own Java applet or ActiveX controls. The main issue will be persuading the user to install these. Although this solution would be fine on an intranet where you can install the code on each machine and keep control, users out there on the Internet might be less keen to download unknown components and allow them to run.

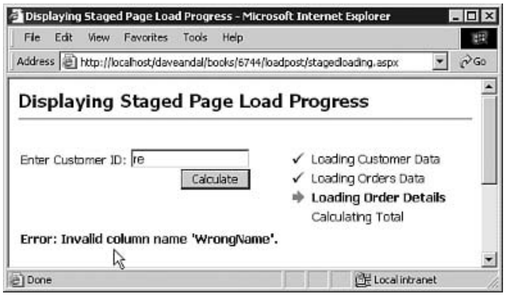


FIGURE 3.11 The sample page, reporting a data access error.

Although it's taken a while to examine the code used in this example, you can see that it is not really very complicated. It allows you to create and manage staged processes that provide accurate feedback to users and that can manage errors and display useful status information.

Summary

This chapter is devoted to the topic of finding ways to present users with status information while a complex or lengthy process is taking place. This chapter looks at two different approaches: displaying a simple "please wait" message or animated GIF image and implementing the server-side process as a series of staged individual operations.

The first of these techniques doesn't really provide feedback because the user is just looking at what is effectively the shadow of the last page that the browser displayed. Underneath, it is waiting for a response from the server. However, displaying a message indicating that the user should wait gives the impression that something really is happening. And removing from the page any buttons or other controls that the user might be tempted to play with prevents the page from being resubmitted and upsetting your server-side code.

This chapter also shows how you can improve on the simple "please wait" text message by using an animated GIF image—in this case, a progress bar. By choosing an image that progresses at a rate matching the average page load time, you can make it look as though your server is working flat out to satisfy their request.

Displaying a progress bar image should be a simple task, but as you discovered, there are issues that arise. (And they say that Web development is child's play!) You ended up having to find two different solutions: one for Internet Explorer and another for other types of browsers. This gave you the opportunity to look into how you can load pages in the background by using the `XMLHTTP` object that is part of the standard installation of Internet Explorer 5 and above.

Finally, this chapter looks at a process that uses the `XMLHTTP` object to implement a staged execution and page loading process. This is a really neat solution for an application that has to perform separate tasks to build up the final page that is returned to the client. And, of all the techniques examined in this chapter, this one alone has the advantage of providing accurate real-time status information as the server processes proceed.

If you decide to follow the asynchronous page-loading route, you might like to look at an implementation designed for the .NET Framework by Microsoft, called the Asynchronous Invocation Application Block for .NET. See `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/paiblock.asp` for more details.

4

Working with Nested List Controls

IN THIS CHAPTER

ASP.NET introduced many extremely useful server controls that can reduce development time and make it easier to create attractive Web pages with a lot less programming effort. Among these is the `DataGrid` control, which—for developers building pages that display and manage data—has become almost the de facto solution. However, many developers still have problems using the `DataGrid` control when stepping beyond the basic mode that it provides for displaying rows of data.

This chapter looks particularly at displaying hierarchical data from related tables or row sets. This is common in many applications, and this chapter investigates four alternative approaches. It also looks at the specific issue of providing a master/detail display where the user can choose to show or hide the related rows.

Displaying Related Data in Nested `DataGrid` Controls

Developers regularly find that they have to build pages that can display data from related tables in a data source and, at first glance, the `DataGrid` control doesn't seem to be able to do this. Many third-party grid controls are available for ASP.NET that are designed to provide this feature, but it's quite easy to achieve the same effect with a `DataGrid` control or a combination of ASP.NET list controls.

The process requires that the list controls be *nested* so that each row within the grid that displays the parent rows contains a list control bound to the related child rows. There are several oft-used approaches for selecting the correct set of child rows for each parent row. The following are the four most common:

- **Declarative nested binding to a `DataSet` instance**—This is the simplest approach, and it requires no code to be written except that required to generate and populate the `DataSet` instance the first time that the page is opened.
- **Filling nested `DataGrid` controls programmatically from a `DataSet` instance**—This technique allows you to extract all the data you want in one operation, while still maintaining control over the selection of child rows, and access or modify the row contents as required.
- **Declarative nested binding to a custom function that returns a row set**—This technique combines the previous two approaches, allowing custom handling of the data when creating the row set to be combined with the simple declarative approach to performing the binding.
- **Filling nested `DataGrid` controls from a `DataReader` instance**—This is a useful technique when you need to display only a few rows. It allows you to dynamically select the child rows you want for each parent row, and it gives you full control over the content at the point where the grid is being populated.

Declarative Nested Binding to a `DataSet` Instance

> **Running the Examples on Your Own Server**
>
> **You must edit the connection string in the `web.config` file provided in the root folder of the examples to suit your server and environment before running this example on your own server. Alternatively, you can run all the examples online at `www.daveandal.net/books`.**

The simplest way to populate nested `DataGrid` controls is to use syntax that allows the child rows to be specified using declarative techniques. In other words, you specify the binding properties of the nested grid at design time, and ASP.NET fetches the rows and performs the binding to generate the output at runtime.

The sample page in Figure 4.1 shows nested binding of three `DataGrid` controls, displaying data extracted from the Northwind sample database that is provided with SQL Server. The outer, or *root*, `DataGrid` control displays details from the `Customers` table, and the grid nested within it displays a list of orders (in the Order History column). However, this nested grid contains within its Details column another `DataGrid` control, which is bound to data extracted from the `Order Details` table. The result is a hierarchical display of all three sets of related data rows.

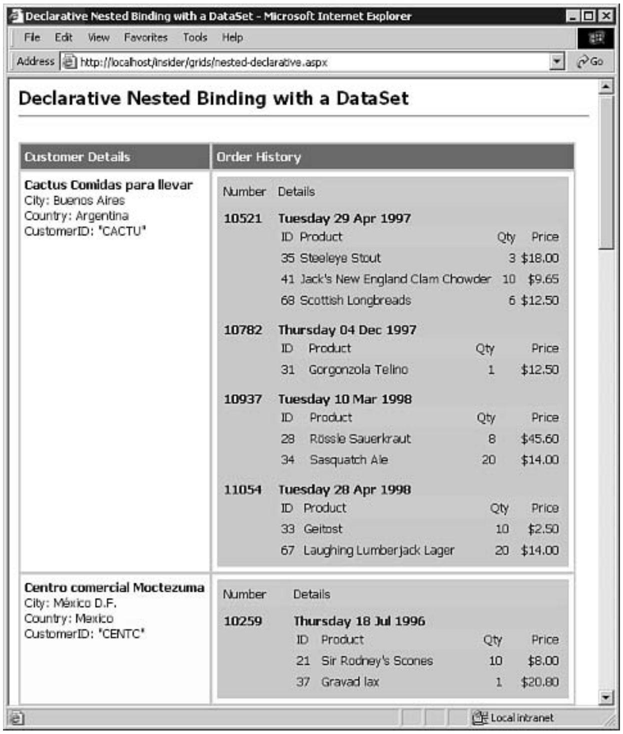


FIGURE 4.1
Nested `DataGrid` controls, using declarative data binding.

The page starts the usual `Page` and `Import` directives:

```
<%@Page Language="VB" EnableViewState="False" %>
<%@Import Namespace="System.Data" %>
<%@Import Namespace="System.Data.OleDb" %>
```

However, in this case you turn *off* viewstate for the page. You don't intend to perform postbacks, which means that you'll only generate the data once, and you don't need to preserve the values in the grid, so there is no point in storing it in the viewstate.

Declaring the `DataGrid` Controls

Listing 4.1 shows the declaration of the `<form>` section of the page and the three `DataGrid` controls. It also includes a `Label` control where you will display any data access

> **Saving Bandwidth by Disabling Viewstate**
>
> **To give you some idea of the savings in bandwidth and consequent download time, the resulting page contains 20,207 bytes of viewstate data with viewstate enabled in the `Page` directive. With viewstate disabled, this is reduced to 50 bytes. You could also omit the `<form>` tags from the page, as they are required only when you're performing a postback. However, if you place a Web Forms control such as a `TextBox` control on the page—perhaps to allow editing of the contents—you must use a server-side `<form>` tag. Most ASP.NET development tools insert a server-side `<form>` tag into every page by default.**

errors. The declaration of the DataGrid control includes a range of style and formatting attributes, including declarations of the <HeaderStyle>, <ItemStyle>, and <AlternatingItemStyle> elements.

LISTING 4.1 The Declaration of the DataGrid Controls

```
<form runat="server">

  <asp:Label id="lblErr" EnableViewState="False" runat="server" />

  <asp:DataGrid id="dgr1" runat="server"
       Font-Size="10" Font-Name="Tahoma,Arial,Helvetica,sans-serif"
       BorderStyle="None" BorderWidth="1px" BorderColor="#deba84"
       BackColor="#DEBA84" CellPadding="5" CellSpacing="1"
       AutoGenerateColumns="False" >
    <HeaderStyle Font-Bold="True" ForeColor="#ffffff"
                 BackColor="#b50055" />
    <ItemStyle BackColor="#FFF7E7" VerticalAlign="Top" />
    <AlternatingItemStyle backcolor="#ffffc0" />
    <Columns>

      <asp:TemplateColumn HeaderText="Customer Details">
        <ItemTemplate>
          <b><%# Container.DataItem("CompanyName") %></b><br />
          City: <%# Container.DataItem("City") %><br />
          Country: <%# Container.DataItem("Country") %><br />
          CustomerID: "<%# Container.DataItem("CustomerID") %>"
        </ItemTemplate>
      </asp:TemplateColumn>

      <asp:TemplateColumn HeaderText="Order History">
        <ItemTemplate>

          <asp:DataGrid id="dgr2" runat="server"
               BorderStyle="None" BorderWidth="0" Width="100%"
               BackColor="#deba84" CellPadding="5" CellSpacing="2"
               AutoGenerateColumns="False"
               DataSource='<%# CType(Container.DataItem, _
                 DataRowView).CreateChildView("CustOrders") %>' >
            <HeaderStyle BackColor="#c0c0c0" />
            <ItemStyle Font-Bold="True" VerticalAlign="Top" />
            <Columns>

              <asp:BoundColumn DataField="OrderID"
                   HeaderText="Number" />
```

LISTING 4.1 Continued

```
              <asp:TemplateColumn HeaderText="Details">
                <ItemTemplate>
                  <asp:Label runat="server"
                    Text='<%# DataBinder.Eval(Container.DataItem, _
                    "OrderDate", "{0:dddd dd MMM yyyy}") %>' />

                  <asp:DataGrid id="dgr3" runat="server"
                    BorderStyle="None" BorderWidth="0"
                    CellPadding="3" CellSpacing="0" Width="100%"
                    AutoGenerateColumns="False"
                    DataSource='<%# CType(Container.DataItem, _
                    DataRowView).CreateChildView( _
                                 "OrdersODetails") %>' >
                    <HeaderStyle BackColor="#c0c0c0" />
                    <Columns>
                      <asp:BoundColumn DataField="ProductID"
                           HeaderText="ID" />
                      <asp:BoundColumn DataField="ProductName"
                           HeaderText="Product" />
                      <asp:BoundColumn DataField="Quantity"
                           ItemStyle-HorizontalAlign="Right"
                           HeaderStyle-HorizontalAlign="Right"
                           HeaderText="Qty" />
                      <asp:BoundColumn DataField="UnitPrice"
                           DataFormatString="${0:f2}"
                           ItemStyle-HorizontalAlign="Right"
                           HeaderStyle-HorizontalAlign="Right"
                           HeaderText="Price"/>
                    </Columns>
                  </asp:DataGrid>

                </ItemTemplate>
              </asp:TemplateColumn>
            </Columns>
          </asp:DataGrid>

        </ItemTemplate>
      </asp:TemplateColumn>
    </Columns>
  </asp:DataGrid>

</form>
```

You can't rely on the autogeneration feature for the columns in the grid in this example because you want to include a DataGrid control in one of the columns. So you include the AutoGenerateColumns="False" attribute in the declaration of the main root DataGrid control and include a <Columns> element where you declare the columns you want.

Inside this <Columns> element, you specify a <TemplateColumn> element that displays a range of values extracted from the data rows that will be used to populate this grid. You include the company name, city, and country, as well as the value of the key column named CustomerID. You specify each by using the standard syntax for accessing the DataItem instance (the current row) of the Container object (the binding context that references the set of data rows) and specifying the column name:

```
<%# Container.DataItem("column-name") %>
```

The second column in this root grid is another <TemplateColumn> element, but this time it contains a nested DataGrid control (id="dgr2")—so each row in the root DataGrid control will contain an instance of the nested DataGrid control to display order details. This DataGrid control also disables auto-generation of columns and contains a <Columns> element. The important point to note here is that you declare the data source for this nested DataGrid control at design time. Later in this chapter you'll see how the declaration you've used works.

Meanwhile, the nested DataGrid control contains one BoundColumn element to display the value of the order ID (the row key) for each order for this customer and a <TemplateColumn> element that contains another nested DataGrid control (id="dgr3"). An instance of this third DataGrid control will be generated for every order and will be used to display the order lines for this order. In this case, you just use the normal BoundColumn elements to display the product ID, name, quantity, and unit price.

Declaring the DataSource Property for a Nested List Control

The interesting part of Listing 4.1, and the feature that makes it work, is the way you declare the DataSource attributes for the two nested DataGrid controls. Normally, as with the root DataGrid, you specify the DataSource property for the list controls at runtime. However, when you nest list controls (as in this example), you can specify a function that returns the set of data to populate the control within the declarative definition of that control.

The following is the first DataSource attribute used in the example:

```
DataSource='<%# CType(Container.DataItem, _
           DataRowView).CreateChildView("CustOrders") %>'
```

This statement converts the data source row that is providing the data to populate the current row of the root grid into a DataRowView instance, and then it calls its CreateChildView method. The name of a relationship between the current data source row set and the child row set must be provided, and the function returns a set of child rows that are related to the current row in the parent row set.

For this to work, the data must be stored in such a way that the relationship between the parent and child row sets is available, and the obvious way to meet this criterion is to populate tables in an ADO.NET DataSet instance with the rows from the data source. Then you create the relationship(s) between these tables within the DataSet instance.

In this example, you have three DataGrid controls, so there are three sets of data rows in the DataSet instance that you use to populate them: data extracted from the Customers, Orders, and Order Details tables in the Northwind database. The DataSet instance that contains these rows also contains two relationships (DataRelation objects), named CustOrders and OrdersODetails (see Figure 4.2).

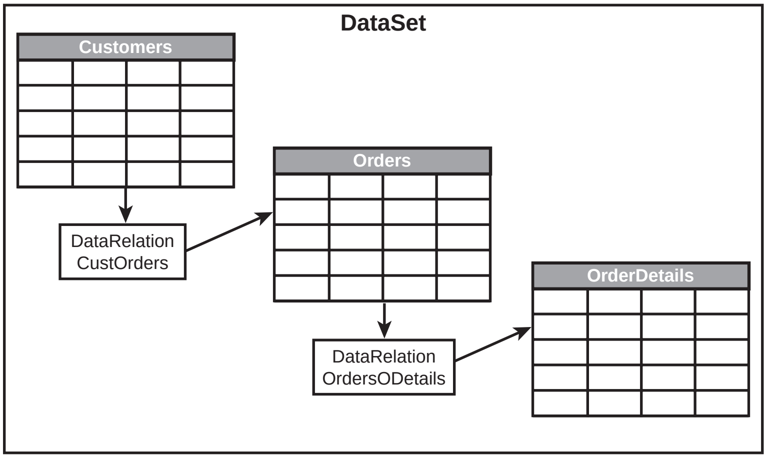


FIGURE 4.2
The structure of the DataSet instance for the nested DataGrid control example.

The first of these relationships is used to create the row set for the data source of the second DataGrid control (id="dgr2") that displays details (such as the delivery address) of each order for the current customer. The second relationship is used to create the row set for the data source of the third DataGrid control (id="dgr3"), which displays the individual lines for each order:

```
DataSource='<%# CType(Container.DataItem, _
          DataRowView).CreateChildView("OrdersODetails") %>' >
```

When you subsequently bind the root DataGrid control to its data source, the nested grids will automatically be populated with the matching sets of child rows.

Populating a DataSet Instance and Adding Relationships

The code in the sample page is responsible for creating the DataSet instance and adding the relationships between the tables to it. Listing 4.2 declares a page-level variable to hold the DataSet instance and then calls a separate routine named FillDataSet in the Page_Load event handler to fill it with the data and relationships required. When you have the DataSet instance, you bind it to the root DataGrid, specify which table it should draw its data from, and call the DataBind method to initiate the process of binding all three DataGrid objects.

LISTING 4.2 The Page-Level Variable and the Code in the Page_Load Event Handler

```
' variable to hold reference to DataSet across routines
Dim oDataSet As DataSet

Sub Page_Load()

  'fill the data set with some rows from database
  FillDataSet("c%")

  ' bind the data to the grid for display
  dgr1.DataSource = oDataSet
  dgr1.DataMember = "Customers"
  dgr1.DataBind()

End Sub
```

Listing 4.3 shows the `FillDataSet` routine that is used to populate the `DataSet` instance. This routine receives a `String` object that contains the full or partial match for the customer ID whose orders you want to list. (In Listing 4.2, it is set to `"c%"` to extract order details for all customers whose ID starts with *c*.) Using this ID, you can build the SQL statements you require to extract the appropriate sets of rows from the `Customers`, `Orders`, and `Order Details` tables in the database. You have to join the `Products` table in the third SQL statement to get the name of the product because the `Order Details` table only contains a foreign key to the rows in this table—not the product name.

LISTING 4.3 The Code to Populate the DataSet Instance

```
Sub FillDataSet(sCustID As String)

  ' get DataSet with rows from Northwind tables

  Dim sCustSql As String _
    = "SELECT CustomerID, CompanyName, City, Country " _
    & "FROM Customers WHERE CustomerID LIKE '" & sCustID & "'"
  Dim sOrdersSql As String _
    = "SELECT CustomerID, OrderID, OrderDate FROM Orders " _
    & "WHERE CustomerID LIKE '" & sCustID & "'"
  Dim sDetailsSql As String _
    = "SELECT [Order Details].OrderID, Products.ProductID, " _
    & "Products.ProductName, [Order Details].Quantity, " _
    & "[Order Details].UnitPrice " _
    & "FROM [Order Details] JOIN Products " _
    & "ON [Order Details].ProductID = Products.ProductID " _
    & "WHERE [Order Details].OrderID IN " _
    & " (SELECT OrderID FROM Orders " _
```

LISTING 4.3 Continued

```
    & "  WHERE CustomerID LIKE '" & sCustID & "')"

  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindOleDbConnectString")
  Dim oConnect As New OleDbConnection(sConnect)
  oDataSet = New DataSet()

  Try

    ' fill DataSet with three tables
    Dim oDA As New OleDbDataAdapter(sCustSQL, oConnect)
    oConnect.Open()
    oDA.Fill(oDataSet, "Customers")
    oDA.SelectCommand.CommandText = sOrdersSql
    oDA.Fill(oDataSet, "Orders")
    oDA.SelectCommand.CommandText = sDetailsSql
    oDA.Fill(oDataSet, "OrderDetails")
    oConnect.Close()

    ' create relations between the tables
    Dim oRel As New DataRelation("CustOrders", _
      oDataSet.Tables("Customers").Columns("CustomerID"), _
      oDataSet.Tables("Orders").Columns("CustomerID"))
    oDataSet.Relations.Add(oRel)
    oRel = New DataRelation("OrdersODetails", _
      oDataSet.Tables("Orders").Columns("OrderID"), _
      oDataSet.Tables("OrderDetails").Columns("OrderID"))
    oDataSet.Relations.Add(oRel)

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text = oErr.Message

  End Try

End Sub
```

> **Using Stored Procedures**
>
> You could use stored procedures to extract the rows, of course, but the aim of this example is to demonstrate binding techniques for the server controls, so you use SQL statements to avoid unnecessary complexity.

Next, the connection string is extracted from `web.config`, and you can create a `Connection` instance and a new empty `DataSet` instance. Then, within the `Try...Catch` construct, you create a `DataAdapter` instance, open the connection, and fill the three tables—changing the `CommandText` property of `DataAdapter` to the appropriate SQL statement as you go.

Creating and Adding the `DataRelation` Instances

When the tables are filled, you can create and add the relationships you need between them. You create a new `DataRelation` object by specifying the name you want to assign to it, the column in the parent table that contains the key to match to the child table, and the column in the child table that contains this value as a foreign key. Figure 4.3 shows the relationships between the Northwind database tables that are used in this example, as well as the primary key and foreign keys in each table.

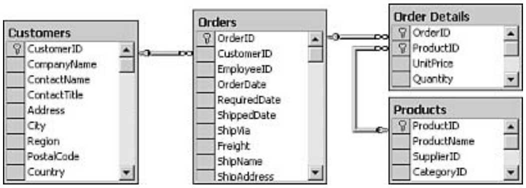


FIGURE 4.3 The relationships between the tables used in this example.

In Listing 4.3 you can see that to create the relationship between the `Customers` and `Orders` tables in the `DataSet` instance, you use the following:

```
Dim oRel As New DataRelation("CustOrders", _
  oDataSet.Tables("Customers").Columns("CustomerID"), _
  oDataSet.Tables("Orders").Columns("CustomerID"))
```

Then, to add this relationship to the `DataSet` instance, you use the following:

```
oDataSet.Relations.Add(oRel)
```

To create the relationship between the `Orders` and `Order Details` tables and add that relationship to the `DataSet` instance, you use the following:

```
oRel = New DataRelation("OrdersODetails", _
  oDataSet.Tables("Orders").Columns("OrderID"), _
  oDataSet.Tables("OrderDetails").Columns("OrderID"))
  oDataSet.Relations.Add(oRel)
```

Other than closing the connection if it's open and displaying a message if an error occurs, this is all the code you need. When the page is opened, the `DataSet` instance is filled with data, and the three relationships are added. Then the `DataBind` method causes the three `DataGrid` controls to be populated. The output you want (refer to Figure 4.1) is then generated automatically.

Filling Nested DataGrid Controls with a DataSet Instance

Instead of using declarative binding, as demonstrated in the previous example, you might want to exert more control over the binding of child rows to their respective DataGrid controls. The example in this section uses the same DataSet instance as the previous example, but in this case, you'll bind the nested DataGrid controls dynamically, using code, instead of defining the bindings declaratively.

This approach allows you to access the data row and examine the values, modify them as required, and even decide whether to bind the nested DataGrid control at runtime. You could, for example, test whether a product was in stock before displaying the details or omit discontinued products when generating sales forecasts.

Figure 4.4 shows the output for this example, and you can see that there are subtle differences from the preceding example. This example omits the details of orders that have not yet been shipped. (In this example, the orders numbered 10782 and 10937 are visible in Figure 4.1.) This example also highlights the names of products that have unit prices greater than $10.00, using bold italic text.

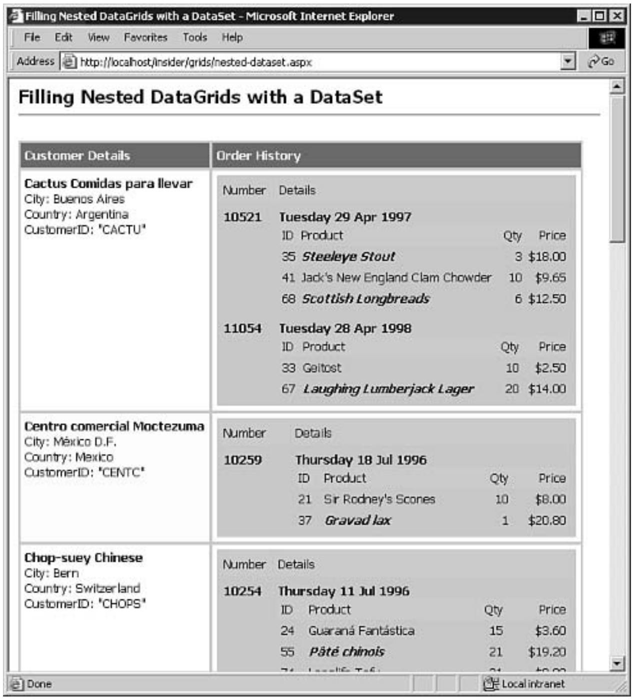


FIGURE 4.4
A page that demonstrates nested data binding to a DataSet instance.

The Changes to This Example when Declaring the DataGrid Controls

When you declare the DataGrid controls in this example, you no longer include the DataSource attributes for the two nested grids (dgr1 and dgr2), but you do add two more: the DataKeyField and OnItemDataBound attributes.

The `DataKeyField` attribute specifies the name of the column in the source row set that contains the primary key for each row. You can easily extract this value for any row by referring to the `DataKeys(`*`row-index`*`)` property of the `DataGrid` control.

The `OnItemDataBound` attribute specifies the name of an event handler that the `DataGrid` control will execute each time it binds to the source data for a row. In this event handler, you can access the ASP.NET server controls in the current row and the row in the source row set that is providing the data for the row.

The opening tag of the root `DataGrid` control (`id="dgr1"`) looks like this:

```
<asp:DataGrid id="dgr1" runat="server"
     ...
     AutoGenerateColumns="False"
     DataKeyField="CustomerID"
     OnItemDataBound="BindOrdersGrid">
```

The opening tag of the first nested `DataGrid` control (`id="dgr2"`) looks like this:

```
<asp:DataGrid id="dgr2" runat="server"
     ...
     AutoGenerateColumns="False"
     DataKeyField="OrderID"
     OnItemDataBound="BindOrderItemsGrid">
```

The Changes to This Example when Populating the Data Set

The only change to the code used to populate the `DataSet` instance and add the relationships to it occurs because, this time, you want to be able to access the value of the `ShippedDate` column in the `Orders` table for each row; this is how you detect whether the order has shipped. All you do is add the `ShippedDate` column to the SQL statement that extracts the rows from the `Orders` table:

```
Dim sOrdersSql As String _
  = "SELECT CustomerID, OrderID, OrderDate, ShippedDate " _
  & "FROM Orders WHERE CustomerID LIKE '" & sCustID & "'"
```

Handling the `ItemDataBound` Events

In this example, you've removed the `DataSource` attributes from the two nested `DataGrid` controls, which means that they will not display anything when you view the page. All you'll see is the list of customers, generated when the root `DataGrid` is bound to the `Customers` table in the `DataSet` instance by code in the `Page_Load` event handler. However, both this root `DataGrid` control and the first of the nested `DataGrid` controls will execute the custom routines when the `ItemDataBound` event occurs.

The `ItemDataBound` Event Handler for the `Customers` Table `DataGrid` Control

The root `DataGrid` control, which displays data from the `Customers` table, will execute the routine named `BindOrdersGrid` for each row it contains. The task here is to create a row set containing just the appropriate matching child rows from the Orders table and then bind that row set to the nested DataGrid control within each Customers row. Along the way, after you've created the child row set, you can play with it by changing the values and the output generated by the DataGrid control.

The BindOrdersGrid routine is shown in Listing 4.4. In it, you first test what type of item the event is occurring for—it could be a row containing data, a header row, a footer row, or a separator row. (All these types of row can be declared using templates or attributes of the DataGrid control, and the ItemDataBound event occurs for them all when present.) Also, notice that the code tests for an AlternatingItemRow instance. Even if you only define an <ItemTemplate> element or use a BoundColumn control, the event is raised alternately as an Item row type and an AlternatingItem row type.

LISTING 4.4 The BindOrdersGrid Event Handler

```
Sub BindOrdersGrid(sender As Object, e As DataGridItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused the event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  ' only process it if it's an Item or AlternatingItem event
  If oType = ListItemType.Item _
  Or oType = ListItemType.AlternatingItem Then

    ' get a reference to the DataGrid control in this row
    Dim oGrid As DataGrid = CType(e.Item.FindControl _
                              ("dgr2"), DataGrid)

    ' get value of CustomerID for this row from DataKeys collection
    Dim sKey As String = dgr1.DataKeys(e.Item.ItemIndex)

    ' get a DataView containing just the current row in
    ' the Customers table within the DataSet
    Dim oView, oChildView As DataView
    oView = oDataSet.Tables("Customers").DefaultView
    oView.RowFilter = "CustomerID = '" & sKey & "'"
    oChildView = oView(0).CreateChildView( _
                 oDataSet.Relations("CustOrders"))

    ' find rows that have not yet shipped and delete them
    ' have to go backwards through row collection to avoid
    ' errors as indexes of rows change when one is deleted
    For iIndex As Integer = (oChildView.Count - 1) To 0 Step -1
      If oChildView(iIndex)("ShippedDate").ToString() = "" Then
        oChildView(iIndex).Delete()
      End If
```

LISTING 4.4 Continued

```
    Next

    ' bind nested "orders" DataGrid to child DataView
    oGrid.DataSource = oChildView
    oGrid.DataBind()

  End If

End Sub
```

Testing the Row Type in `ItemDataBound` Event Handlers

A common mistake when handling the `ItemDataBound` event and the `ItemCreated` event is to fail to properly establish the type of row that each event is being raised for before trying to access the contents. For example, if a row contains a `Label` control when in "normal" mode and a `TextBox` control when in "edit" mode, you must determine the row type before trying to access the `Label` or `TextBox` control. If the row type is `ListItemType.Item` or `ListItemType.AlternatingItem`, you can only access the `Label` control. If it is `ListItemType.EditItem`, you can only access the `TextBox` control. The same kind of logic applies to a row that is in "selected" mode, in which case the row type is `ListItemType.SelectedItem`.

The next step is to get a reference to the `DataGrid` control within the current row. The event handler is executed once for each row in the root `DataGrid` control. It receives a `DataGridItemEventArgs` instance that contains more details of the event and a reference to the current row in the `DataGrid` control as a `DataGridItem` object. This object has a whole range of properties that can be used to set the style of the row, such as the background and foreground colors, borders, font, text alignment, and so on. However, the properties and methods you're usually most interested in when accessing or manipulating the contents of a row are those shown in Table 4.1.

TABLE 4.1
Commonly Used Properties and Methods of the `DataGridItem` Object

Property or Method	Description
`Cells`	Gets a collection of the table cells in the current row as `TableCell` objects.
`Attributes`	Enables attributes to be added to or removed from the HTML elements that are generated for the current row.
`Controls`	Returns a collection of all the child controls for the current row.
`DataItem`	Returns a reference to the source data row as a `DataRowView` object.
`DataSetIndex`	Returns the index of the current row within the bound data source.
`EnableViewState`	Specifies whether the controls in the current row will persist their viewstates within the page.
`ItemIndex`	Returns the index of the current row within the `Items` collection of the `DataGrid` control.
`ItemType`	Returns a value from the `ListItemType` enumeration that indicates the current row type.
`FindControl("`*`id`*`")`	Returns a reference to a control within the row, given its ID, or `Nothing` if the control is not found.
`HasControls()`	Returns `True` if the current row contains any server controls, or `False` if not.

You can use the FindControl method to locate the DataGrid control we're looking for in the current row. You have to cast the result to the correct type on return because the FindControl method returns the reference as a generic Object type:

```
Dim oGrid As DataGrid = CType(e.Item.FindControl("dgr2"), DataGrid)
```

Then you generate the set of child rows that match the current row by creating a filtered DataView instance on the Customers table, which will only contain the row for the current customer. You do this by extracting the ID of the current customer from the DataKeys collection of the DataGrid control, using the ItemIndex property of the DataGridItem instance passed to the routine (refer to Table 4.1).

Then, to limit the rows that are displayed in the DataGrid control, you set the RowFilter property of the default DataView instance of the Customers table, as shown in this section of the code:

```
Dim sKey As String = dgr1.DataKeys(e.Item.ItemIndex)
Dim oView, oChildView As DataView
oView = oDataSet.Tables("Customers").DefaultView
oView.RowFilter = "CustomerID = '" & sKey & "'"
```

Now you can use the CreateChildView method of the first (and only) row in the DataView instance to create the set of related child rows from the Orders table. You do this the same way as in the previous declarative binding example, specifying the name of the DataRelation instance that links the two tables in the DataSet instance:

```
oChildView = oView(0).CreateChildView( _
             oDataSet.Relations("CustOrders"))
```

At this point, you can perform any actions you want to carry out on the source data or on the DataGrid row and its contents. In this example, you want to hide any rows that have not yet shipped. You can do this by simply deleting them from the child DataView instance. However, because the index of the remaining rows changes when a row is deleted, you have to iterate through the rows in reverse order:

```
For iIndex As Integer = (oChildView.Count - 1) To 0 Step -1
  If oChildView(iIndex)("ShippedDate").ToString() = "" Then
    oChildView(iIndex).Delete()
  End If
Next
```

Then, when you're happy with the contents of the DataView instance, you can bind it to the nested DataGrid control to which you're holding a reference in the oGrid variable:

```
oGrid.DataSource = oChildView
oGrid.DataBind()
```

This causes the DataGrid control showing the orders for the current customer to generate its contents (a list of orders for this customer) for display. However, remember that you also declared an `ItemDataBound` event handler for this `DataGrid` control—and in it you'll perform much the same process you've just seen to populate the third `DataGrid` control, which contains the list of order lines.

The `ItemDataBound` Event Handler for the `Orders` Table `DataGrid` Control

The second `DataGrid` control, which displays data from the `Orders` table, will execute the routine named `BindOrderItemsGrid` for each row as it is bound to its data source. Listing 4.5 shows this routine in full. Much of this listing is similar to the `BindOrdersGrid` routine in Listing 4.4. The differences are summarized individually in this section.

LISTING 4.5 The `BindOrderItemsGrid` Event Handler

```
Sub BindOrderItemsGrid(sender As Object, e As DataGridItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused the event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  ' only process it if it's an Item or AlternatingItem event
  If oType = ListItemType.Item _
  Or oType = ListItemType.AlternatingItem Then

    ' get the value of the CustomerID column
    ' argument sender is a reference to the containing DataGrid
    Dim iKey As Integer = sender.DataKeys(e.Item.ItemIndex)

    ' get a reference to the DataGrid control in this row
    Dim oGrid As DataGrid = CType(e.Item.FindControl("dgr3"), DataGrid)

    ' get a DataView containing just the current row in
    ' the Orders table within the DataSet
    Dim oView, oChildView As DataView
    oView = oDataSet.Tables("Orders").DefaultView
    oView.RowFilter = "OrderID = " & iKey
    oChildView = oView(0).CreateChildView( _
                 oDataSet.Relations("OrdersODetails"))

    ' find rows where unit price is greater
    ' than $10.00 and highlight product name
    For iIndex As Integer = 0 To oChildView.Count - 1
      If oChildView(iIndex)("UnitPrice") > 10 Then
        oChildView(iIndex)("ProductName") = "<b><i>" _
                 & oChildView(iIndex)("ProductName") & "</i></b>"
      End If
    Next

    ' bind nested "order details" DataGrid to child DataView
```

LISTING 4.5 Continued

```
    oGrid.DataSource = oChildView
    oGrid.DataBind()

  End If

End Sub
```

After checking the type of item that the event was raised for, the next task is to get a reference to the child DataGrid control that you want to populate with the lists of order lines from the OrderDetails table in the DataSet instance. In the BindOrdersGrid routine, you accessed the DataKeys collection of the current DataGrid control (the one that raised the ItemDataBound event) simply by referring to the DataGrid control with its ID. This works because there is only one instance of the root DataGrid control.

However, the DataGrid control for which you're handling the ItemDataBound event this time is one of multiple instances—there is an instance for each order for each customer. Therefore, you can't just use the ID of the grid (dgr2) to reference the DataKeys collection. The actual ID of each grid will be a combination of the parent grid control ID, any intermediate container control IDs, and the ID of this DataGrid control—in other words, something like "dgr1__ctl2_dgr2".

However, remember that event handlers pass a reference to the control that raised the event as the first (*sender*) parameter. You can use this to get a reference to the DataKeys collection, and from it you can get the OrderID value of the current order. Then you can get a reference to the child grid control in this row (the one that will display the order lines), using the FindControl method of the current DataGridItem instance as before.

The next section of code in Listing 4.5 creates the child DataView instance you want to bind to the DataGrid control in this row, using the same techniques as in Listing 4.4. However, before you bind this row set to the DataGrid control, you "massage" it by checking for any items that have a unit price greater than $10.00. For each one you find, you just add some formatting elements to the text value in the ProductName column of that row in the DataView instance, before binding it to the current DataGrid control to display the results.

Declarative Nested Binding to a Custom Function

The third technique for binding related data to nested list controls is actually a combination of the two techniques just described. ASP.NET supports declarative data binding statements that bind to the result of a function, using the following syntax:

```
<%# function-name(parameters) %>
```

You can use this technique to insert the result of a function almost anywhere in an ASP.NET page. You can use it simply to generate output directly. For example, if you have a function that returns the description for a specific paragraph in a document, you can insert the result into the page by using the following:

```
<p>This paragraph describes <%# GetParaDescription(42) %></p>
```

Alternatively, you can bind the function result to a property of a server control. For example, you can set the `Text` property of a `Label` control by using the same function like this:

```
<asp:Label id="mylabel" runat="server"
     Text='<%# GetParaDescription(42) %>' />
```

Your code simply has to call the `DataBind` method of the appropriate container control to force the binding to take place. In the two preceding cases, you'd call the `DataBind` method of the `Page` object itself.

Of course, this approach to declarative binding is just what you used in the first example in this chapter. You specified the `DataSource` property of the nested `DataGrid` controls, using an attribute such as this:

```
DataSource='<%# CType(Container.DataItem, _
                DataRowView).CreateChildView("OrdersODetails") %>'
```

`CreateChildView` is a method of the `DataRowView` class, and it returns a `DataView` instance (a row set) that can be used as the source for a `DataGrid` control. So you shouldn't be surprised to see in the next example that you can use the same approach but specify a custom function that returns a row set and have it used to populate the nested `DataGrid` controls.

The Custom Functions to Return Row Sets

In the previous example, you handled the `ItemDataBound` event of two of the `DataGrid` controls so that you could create and then massage the row sets before using them to populate their respective nested `DataGrid` controls. In this example, you use the same core code to generate the row sets you need, and you modify their contents, as in the previous example. However, this time the two sections of code (which were in the `BindOrdersGrid` and `BindOrderItemsGrid` event handlers) are extracted and converted into functions that return a `DataView` instance.

Listing 4.6 shows the two functions, named `GetOrdersGridRows` and `GetOrderItemsGridRows`. Obviously, this time, because you aren't using the functions to handle events, you don't have access to the `DataGridItemEventArgs` objects that contain details of the event and that are passed to the `ItemDataBound` event handler. However, the only information you actually need to be able to create the appropriate row set is the value of the key for the current row in the `DataGrid` control.

LISTING 4.6 The Custom Functions That Return Row Sets

```
Function GetOrdersGridRows(sRowKey As String) As DataView

  ' get a DataView containing just the current row in
  ' the Customers table within the DataSet
```

LISTING 4.6 Continued

```
  Dim oView, oChildView As DataView
  oView = oDataSet.Tables("Customers").DefaultView
  oView.RowFilter = "CustomerID = '" & sRowKey & "'"
  oChildView = oView(0).CreateChildView( _
               oDataSet.Relations("CustOrders"))

  For iIndex As Integer = (oChildView.Count - 1) To 0 Step -1
    If oChildView(iIndex)("ShippedDate").ToString() = "" Then
      oChildView(iIndex).Delete()
    End If
  Next

  Return oChildView

End Function


Function GetOrderItemsGridRows(iRowKey As Integer) As DataView

  ' get a DataView containing just the current row in
  ' the Orders table within the DataSet
  Dim oView, oChildView As DataView
  oView = oDataSet.Tables("Orders").DefaultView
  oView.RowFilter = "OrderID = " & iRowKey
  oChildView = oView(0).CreateChildView( _
               oDataSet.Relations("OrdersODetails"))

  For iIndex As Integer = 0 To oChildView.Count - 1
    If oChildView(iIndex)("UnitPrice") > 10 Then
      oChildView(iIndex)("ProductName") = "<b><i>" _
                & oChildView(iIndex)("ProductName") & "</i></b>"
    End If
  Next

  Return oChildView

End Function
```

Assuming that you can pass this key as a parameter to your functions, the remaining code (which actually generates the `DataView` instance) is identical to that in Listings 4.4 and 4.5. You reference the single row in the parent table that matches the key supplied as a parameter, and you use the `CreateChildView` method to generate the child row set. Then you remove any rows for orders that have not shipped or highlight the names of products over $10.00, just as before.

Binding DataGrid Controls to Custom Functions

The two functions described in the preceding section replace the two event handlers that are used in the previous example, so you must remove the OnItemDataBound attributes from the two DataGrid controls. The server-side code to populate the DataGrid control, add the DataRelation instances to it, and initiate the binding of the root DataGrid control in the Page_Load event handler is identical to the code in the previous example.

The only other change to the page is the way you declare the DataSource attributes for the two DataGrid controls. The first of the nested DataGrid controls (id="dgr2"), which displays the list of orders for each customer, contains the following DataSource attribute:

```
DataSource='<%# GetOrdersGridRows( _
                Container.DataItem("CustomerID")) %>'
```

The GetOrdersGridRows function takes a parameter that is the ID of the current customer. Normally, in the ItemDataBound event handler, you'd get this value from the DataKeys collection of the root DataGrid control. However, the DataView instance that provides the data for this DataGrid control contains the value of the customer ID in each row. It is included in the SQL statement, and you use it when you generate the value for the Customer Details column. You can therefore refer to it here and use it as the parameter value for the function, in the same way you would to populate a column in the DataGrid control.

The same logic applies to the second nested DataGrid control (id="dgr3"), which displays the list of order lines for each order. In this case, the parameter is the order ID, and again it is in the row set you use to populate the parent DataGrid control. So you can set the DataSource property of the innermost DataGrid control by using the following:

```
DataSource='<%# GetOrderItemsGridRows( _
                Container.DataItem("OrderID")) %>'
```

Figure 4.5 shows the result of this example; you can see that it produces identical output to the previous example. This is to be expected because the code you use to generate the row sets is the same. Only the way that you apply it to binding the grid controls differs.

Filling Nested DataGrid Controls from a DataReader Instance

The fourth and final approach to populating nested list controls doesn't use a DataSet instance as the source of the rows. Instead, it uses a DataReader instance to extract the data for the data store. Or, to be more precise, it uses multiple DataReader instances.

It's generally accepted that the DataReader class provides better performance than the DataSet approach when you're extracting data and using it in an ASP.NET page. Figures published by Microsoft while ASP.NET was under development suggested that there were gains of more than 20%, although ultimately the performance gain depends on how you actually end up using the data.

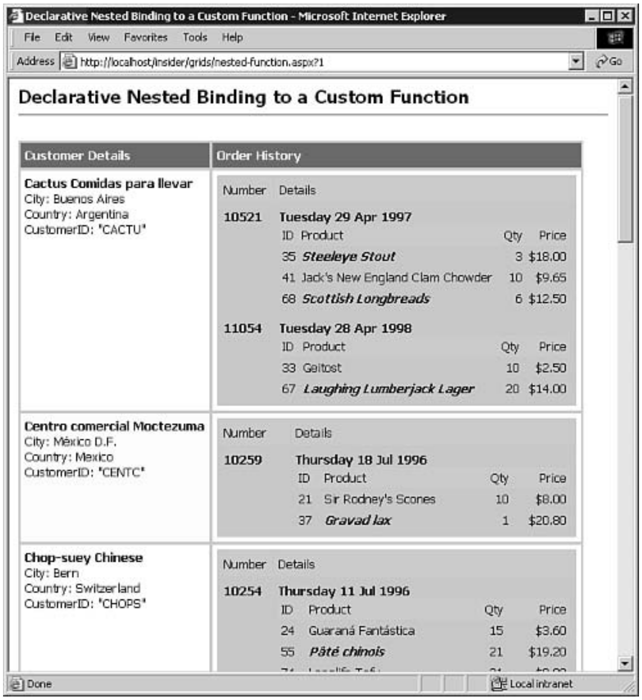


FIGURE 4.5
A sample page that uses declarative binding to custom functions.

The `DataReader` Class Versus the `DataSet` Class

The `DataReader` class is far lighter weight than the `DataSet` class. It's really just a "pipe" that connects the results of a query in the database with the consumer in the ASP.NET page (or other type of application). When you're using ASP.NET server-side data binding, the `DataReader` class is generally the optimal solution, unless you need to cache the data after extracting it or pass it between the tiers of an application.

So how does the `DataReader` class work when you're performing nested data binding? In some ways, it makes the process more complicated. And rudimentary tests show that it doesn't tend to provide any performance increase unless there are only a few rows in the root row set.

The reason for this is that you can't create a hierarchy of tables and the relationships between them with a `DataReader` instance. You can only get one or more unrelated row sets from the data store. This means that each time you need a row set to populate a nested list control, you end up generating a `DataReader` instance, opening the connection, executing the query, and returning the row set.

Okay, so you could reduce the performance hit by reusing the same `DataReader` instance each time (although you'd have to close it and reopen it) and by holding the database connection open until all the row sets have been extracted. But this isn't likely to provide major performance gains because the real hit is the multiple trips to the database that are required.

Still, this technique might prove useful in certain scenarios, and you might decide to adopt it if you have pages with a shallow hierarchy and few rows in the root row sets. This is where any performance gains are most likely to be felt. Figure 4.6 shows the output from the sample page, and you can see that it is identical to the example shown in Figure 4.1. That example used a `DataSet` instance as the data source, but the declarations of the `DataGrid` controls, and the data itself, are the same.

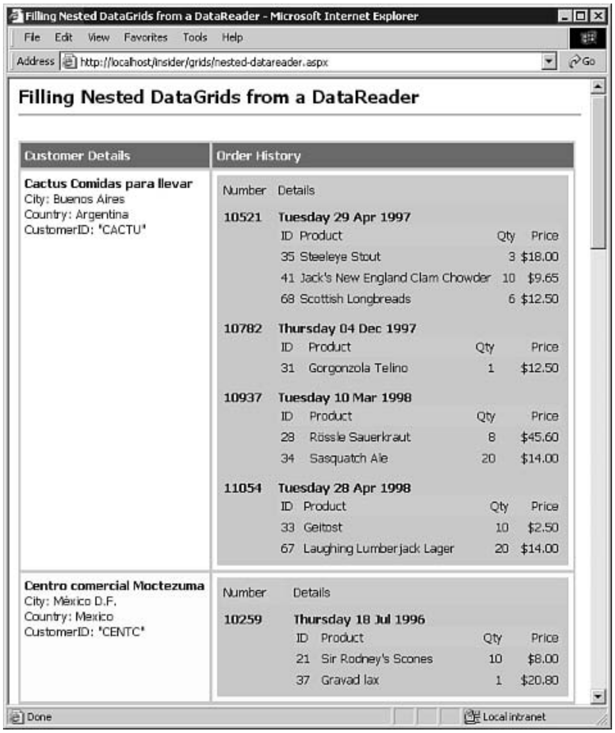


FIGURE 4.6
A demonstration page that uses a `DataReader` instance to extract the data rows.

The Changes to This Example when Declaring the `DataGrid` Controls

The previous examples demonstrate the appearance and disappearance of the `OnItemDataBound` attributes in the `DataGrid` controls. Now they're back again. In this example, you handle the `ItemDataBound` event just as you did when we used a `DataSet` instance as the source for the `DataGrid` controls, in the second example in this chapter (refer to Figure 4.4).

So the root `DataGrid` control contains the attribute `OnItemDataBound="BindOrdersGrid"`, and the nested `DataGrid` control that displays the list of orders for each customer contains the attribute `OnItemDataBound="BindOrderItemsGrid"`. In fact, the declaration of the three grid controls is identical to what

Creating Custom Functions to Return a `DataReader` Instance

Of course, there's no reason you can't create custom functions that return row sets as open `DataReader` instances rather than as `DataView` instances. If you did this, you could avoid handling the `ItemDataBound` event and instead use the same declarative approach as in the preceding example. As you can see, the four examples in this chapter are designed to give you a taste of the possible combinations of techniques. They by no means cover the complete set of permutations.

is used in the example of Figure 4.4—where you bound them to row sets extracted from a `DataSet` instance.

The Changes to the Server-Side Code in This Example

Using a `DataReader` instance instead of a `DataSet` instance requires an almost complete change to the server-side code in the page. Listing 4.7 shows the `Page_Load` event handler, which binds the root `DataGrid` control to its data source, and the three functions that return `DataReader` instances. The first of these is used to generate the row set containing a list of customers that is bound to the root `DataGrid` control in the `Page_Load` event handler.

LISTING 4.7 The `Page_Load` Event Handler and the Routines to Fetch the Row Sets

```
Sub Page_Load()

  ' bind the data to the grid for display
  dgr1.DataSource = GetCustomers()
  dgr1.DataBind()

End Sub


Function GetCustomers() As OleDbDataReader

  Dim sSelect As String _
    = "SELECT CustomerID, CompanyName, City, Country " _
    & "FROM Customers WHERE CustomerID LIKE 'c%'"
  Return GetReader(sSelect)

End Function


Function GetOrders(sKey As String) As OleDbDataReader

  Dim sSelect As String _
    = "SELECT OrderID, OrderDate FROM Orders WHERE CustomerID='" & sKey & "'"
  Return GetReader(sSelect)

End Function


Function GetOrderLines(iKey As Integer) As OleDbDataReader

  Dim sSelect As String _
    = "SELECT Products.ProductID, Products.ProductName, " _
    & "[Order Details].Quantity, [Order Details].UnitPrice " _
    & "FROM [Order Details] JOIN Products " _
```

LISTING 4.7 Continued

```
    & "ON [Order Details].ProductID = Products.ProductID " _
    & "WHERE OrderID=" & iKey.ToString()
  Return GetReader(sSelect)

End Function
```

The three functions shown in Listing 4.7 simply declare a SQL statement and then call the function named `GetReader` shown in Listing 4.8 to create the `DataReader` instance and return it. When creating the row sets for the list of orders or the list of order lines, you need a parameter that specifies the current customer ID or order ID. You can see in Listing 4.7 how these parameters are used to build the SQL statements.

Notice in Listing 4.8 that you specify the value `CommandBehavior.CloseConnection` as a parameter to the `ExecuteReader` method when you create the `DataReader` instance. This ensures that the connection will be closed when the `DataReader` instance is closed or when it goes out of scope.

LISTING 4.8 The Routine to Create a `DataReader` Instance

```
Function GetReader(sSQL As String) As OleDbDataReader

  ' get DataReader for rows from Northwind tables
  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindOleDbConnectString")
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSQL, oConnect)
    Return oCommand.ExecuteReader(CommandBehavior.CloseConnection)

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text = oErr.Message

  End Try

End Function
```

Handling the ItemDataBound Events

As shown in the example in Figure 4.4, the Page_Load event handler initiates the process of displaying the related data by binding the root DataGrid control to the set of customer rows. This raises the ItemDataBound event for each row as it's bound, and in the event handler (BindOrdersGrid), you generate the appropriate set of order rows and bind it to the nested DataGrid control. This in turn causes the ItemDataBound event to be raised for each row in this DataGrid control. In the event handler for this event (BindOrderItemsGrid), you generate the matching set of order detail rows and bind it to the third DataGrid control.

Listing 4.9 shows the two event handlers BindOrdersGrid and BindOrderItemsGrid. As before, you have to check what type of item the event is being raised for, and then you can extract the value of the key from the current row. In the BindOrdersGrid routine, you reference the root DataGrid control, and in the BindOrderItemsGrid routine you use the reference to the DataGrid control that is passed to the event handler as the sender parameter.

Next, you get a reference to the nested DataGrid control in the current row, using the FindControl method of the DataGridItem object that is passed to the event handler. Then you can bind this grid to the result of the appropriate method for generating a DataReader instance.

LISTING 4.9 The Event Handlers for the ItemDataBound Events

```
Sub BindOrdersGrid(sender As Object, e As DataGridItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused the event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  ' only process it if it's an Item or AlternatingItem event
  If oType = ListItemType.Item _
  Or oType = ListItemType.AlternatingItem Then

    ' get value of CustomerID for this row from DataKeys collection
    Dim sKey As String = dgr1.DataKeys(e.Item.ItemIndex)

    ' get a reference to the DataGrid control in this row
    Dim oGrid As DataGrid = CType(e.Item.FindControl("dgr2"), DataGrid)

    ' bind nested "orders" DataGrid to DataReader
    oGrid.DataSource = GetOrders(sKey)
    oGrid.DataBind()

  End If

End Sub


Sub BindOrderItemsGrid(sender As Object, e As DataGridItemEventArgs)
```

LISTING 4.9 Continued

```
 ' see what type of row (header, footer, item, etc.) caused the event
 Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

 ' only process it if it's an Item or AlternatingItem event
 If oType = ListItemType.Item _
 Or oType = ListItemType.AlternatingItem Then

   ' get the value of the CustomerID column
   ' argument sender is a reference to the containing DataGrid
   Dim iKey As Integer = sender.DataKeys(e.Item.ItemIndex)

   ' get a reference to the DataGrid control in this row
   Dim oGrid As DataGrid = CType(e.Item.FindControl("dgr3"), DataGrid)

   ' bind nested "order details" DataGrid to DataReader
   oGrid.DataSource = GetOrderLines(iKey)
   oGrid.DataBind()

 End If

End Sub
```

Other Approaches to Accessing the Data and Performing Nested Data Binding

With the four approaches described in this chapter, you can create the row set to populate the nested `DataGrid` control in whatever way you want. The last example uses simple SQL statements with a `DataReader` instance, but you could equally well use any stored procedure to generate the required results and massage these results as in earlier examples to get exactly the row set you want. Likewise, you could build a row set from an XML document or using custom code to add the rows and columns directly. You could also combine the use of `DataSet` instances and `DataReader` instances, or you could use a `HashTable` instance, an `ArrayList` instance, or whatever data source suits the list controls you are using in the page. And while this chapter's examples use `DataGrid` controls, the same techniques work with various combinations of other list controls—such as `Repeater`, `DataList`, `ListBox`, and `CheckBoxList` controls.

As you can see, using the `DataReader` class is not that different from using the `DataSet` class as far as implementation is concerned. However, remember that you must consider the ramifications of the increased number of trips to the database that the `DataReader` approach requires.

A Master/Detail Display with `DataList` and `DataGrid` Controls

So far, you've only seen pages that display related data and you've only used the `DataGrid` control. To demonstrate some different techniques when using nested list controls, this section shows an example that provides a collapsible master/detail display, using two different list controls; it also allows the child rows to be edited.

Figure 4.7 shows the completed sample page. You can see the same list of customers as in the previous examples in this chapter. For each one there is a drop-down button that, when clicked, opens a list of the orders for that customer and allows them to be edited. At the same time, the button changes to an "up" button that closes the list of orders. Selecting a different customer while one list is open closes that list and opens the selected one, to provide a compact display that reduces bandwidth requirements and provides faster page load times.

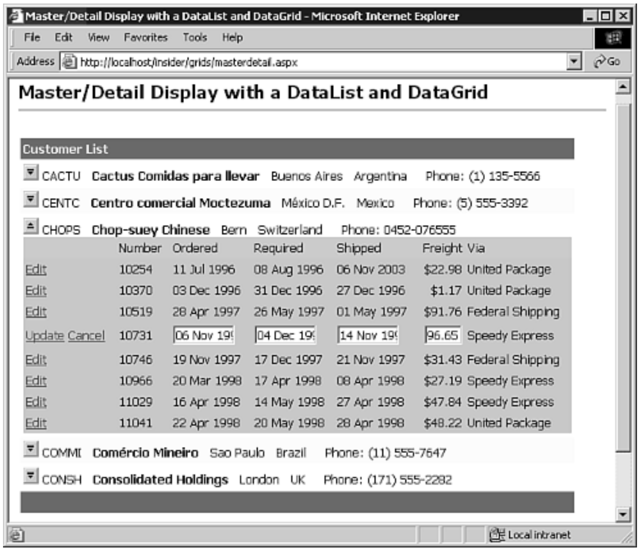


FIGURE 4.7
Creating a collapsible master/detail display for related row sets.

Declaring the DataList and DataGrid Controls

The sample page consists of a `DataList` control that generates the list of customers, to which you apply various formatting and style attributes. This is bound to a row set extracted from the `Customers` table in the Northwind database through a `DataReader` instance.

However, when a row in the `DataList` control is switched to selected mode, that row also displays a `DataGrid` control containing the customer's order details. These rows are extracted from the `Orders` table of the database through another `DataReader` instance.

Finally, when the Edit link in one of the order rows for the selected customer is clicked, that row is placed into edit mode. It then displays the data in the row set that is not read-only in text boxes and provides the Update and Cancel links. Listing 4.10 shows the complete declaration of the `DataList` control and the nested `DataGrid` controls.

LISTING 4.10 The Declaration of the `DataList` and `DataGrid` Controls

```
<asp:DataList id="dtl1" Width="95%" runat="server"
     CellPadding="3" CellSpacing = "2"
     DataKeyField="CustomerID"
     OnItemCommand="DoItemSelect"
```

LISTING 4.10 Continued

```
     OnItemDataBound="BindOrdersGrid" >
  <HeaderStyle Font-Bold="True" ForeColor="#ffffff"
               BackColor="#b50055" />
  <FooterStyle Font-Bold="True" ForeColor="#ffffff"
               BackColor="#b50055" />
  <ItemStyle BackColor="#FFF7E7" VerticalAlign="Top" />
  <AlternatingItemStyle BackColor="#FFFFC0" />

  <HeaderTemplate>
    <b>Customer List</b>
  </HeaderTemplate>

  <ItemTemplate>
    <asp:ImageButton CommandName="Select"
         ImageUrl="~/images/click-down.gif"
         Width="16" Height="17" runat="server"
         AlternateText="Click to view orders" />
    <%# Container.DataItem("CustomerID") %>  
    <b><%# Container.DataItem("CompanyName") %></b>  
    <%# Container.DataItem("City") %>  
    <%# Container.DataItem("Country") %>    
    Phone: <%# Container.DataItem("Phone") %>  
  </ItemTemplate>

  <SelectedItemTemplate>
    <asp:ImageButton CommandName="UnSelect"
         ImageUrl="~/images/click-up.gif"
         Width="16" Height="17" runat="server"
         AlternateText="Click to hide orders" />
    <%# Container.DataItem("CustomerID") %>  
    <b><%# Container.DataItem("CompanyName") %></b>  
    <%# Container.DataItem("City") %>  
    <%# Container.DataItem("Country") %>    
    Phone: <%# Container.DataItem("Phone") %>  

    <asp:DataGrid id="dgr1" runat="server"
         BorderStyle="None" BorderWidth="0" BackColor="#DEBA84"
         CellPadding="3" CellSpacing="0" Width="100%"
         DataKeyField="OrderID"
         OnEditCommand="DoItemEdit"
         OnUpdateCommand="DoItemUpdate"
         OnCancelCommand="DoItemCancel"
         AutoGenerateColumns="False" >
      <HeaderStyle BackColor="#c0c0c0" />
```

LISTING 4.10 Continued

```
<Columns>
  <asp:EditCommandColumn EditText="Edit"
       CancelText="Cancel" UpdateText="Update" />
  <asp:BoundColumn DataField="OrderID" HeaderText="Number"
                   ReadOnly="True" />
  <asp:TemplateColumn HeaderText="Ordered">
    <ItemTemplate>
      <%# DataBinder.Eval(Container.DataItem, "OrderDate", _
                    "{0:dd MMM yyyy}") %>
    </ItemTemplate>
    <EditItemTemplate>
      <asp:TextBox Columns="8" id="txtOrderDate"
           runat="server"
           Text='<%# DataBinder.Eval(Container.DataItem, _
                     "OrderDate", "{0:dd MMM yyyy}") %>' />
    </EditItemTemplate>
  </asp:TemplateColumn>
  <asp:TemplateColumn HeaderText="Required">
    <ItemTemplate>
      <%# DataBinder.Eval(Container.DataItem, "RequiredDate", _
                    "{0:dd MMM yyyy}") %>
    </ItemTemplate>
    <EditItemTemplate>
      <asp:TextBox Columns="8" id="txtRequiredDate"
           runat="server"
           Text='<%# DataBinder.Eval(Container.DataItem, _
                     "RequiredDate", "{0:dd MMM yyyy}") %>' />
    </EditItemTemplate>
  </asp:TemplateColumn>
  <asp:TemplateColumn HeaderText="Shipped">
    <ItemTemplate>
      <%# DataBinder.Eval(Container.DataItem, "ShippedDate", _
                    "{0:dd MMM yyyy}") %>
    </ItemTemplate>
    <EditItemTemplate>
      <asp:TextBox Columns="8" id="txtShippedDate"
           runat="server"
           Text='<%# DataBinder.Eval(Container.DataItem, _
                     "ShippedDate", "{0:dd MMM yyyy}") %>' />
    </EditItemTemplate>
  </asp:TemplateColumn>
  <asp:TemplateColumn HeaderText="Freight"
       HeaderStyle-HorizontalAlign="Right"
       ItemStyle-HorizontalAlign="Right">
```

LISTING 4.10 Continued

```
          <ItemTemplate>
            <%# DataBinder.Eval(Container.DataItem, _
                                "Freight", "${0:f2}") %>
          </ItemTemplate>
          <EditItemTemplate>
            <asp:TextBox Columns="3" id="txtFreight" runat="server"
                 Text='<%# Container.DataItem("Freight") %>' />
          </EditItemTemplate>
        </asp:TemplateColumn>
        <asp:BoundColumn DataField="ShipperName"
             HeaderText="Via" ReadOnly="True"/>
      </Columns>
    </asp:DataGrid>

  </SelectedItemTemplate>

  <FooterTemplate>
     
  </FooterTemplate>

</asp:DataList>
```

The Important Points of the `DataList` Control Declaration

The `DataList` control displays the list of customers, and you add to it three attributes that control its behavior in terms of viewing the order list for each customer. You set the `DataKeyField` attribute to the `CustomerID` column in the source row set so that you can easily get the ID of the customer for the current row:

```
DataKeyField="CustomerID"
```

You also specify the names of two event handlers. The routine named `DoItemSelect` will be executed when any control within the `DataList` control causes a postback, and the routine named `BindOrdersGrid` will be executed each time a row in the `DataList` control is bound to its source data:

```
OnItemCommand="DoItemSelect"
OnItemDataBound="BindOrdersGrid"
```

The `DataList` control declaration uses a header and a footer row to achieve the appearance of the dark bands above and below the list, with the header containing just the plain text “Customer List” and the footer containing a nonbreaking space character (` `) to preserve the row height.

In the `<ItemTemplate>` section, you use an `ImageButton` control to generate the drop-down button. The declaration of the `ImageButton` control sets `CommandName` to `"Select"`; this value is used to detect whether the `ImageButton` button was clicked when the `ItemCommand` event was raised. You also specify the image file for the button (in the `images` subfolder of the application root), the size, and the alternate text that will provide the pop-up ToolTip:

```
<asp:ImageButton CommandName="Select"
     ImageUrl="~/images/click-down.gif"
     Width="16" Height="17" runat="server"
     AlternateText="Click to view orders" />
```

The remainder of the `<ItemTemplate>` content is made up of the usual `Container.DataItem` `("`*`column-name`*`")` data binding statements that display values from the customer row.

The `<SelectedItemTemplate>` section of the `DataList` control declaration comes next. This contains the content that will only be displayed for the single row that is in selected mode when the `DataList` control is bound to its data source. (If no row is selected, this content will not be displayed.) In this template, you provide another `ImageButton` control that allows the user to close the list. You use a different `CommandName` setting this time (`"UnSelect"`), and you use a different image and alternate text (see Figure 4.8):

```
<asp:ImageButton CommandName="UnSelect"
     ImageUrl="~/images/click-up.gif"
     Width="16" Height="17" runat="server"
     AlternateText="Click to hide orders" />
```

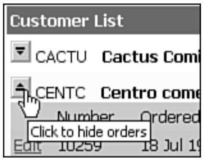


FIGURE 4.8 The buttons to open and close the lists of orders.

Then, after the same set of `Container.DataItem("`*`column-name`*`")` data binding statements as in the `<ItemTemplate>` section (because you want to display the customer details in both modes) comes the declaration of the nested `DataGrid` control.

The Important Points of the `DataGrid` Control Declaration

The `DataGrid` control that displays the order details for the selected customer is placed in the `<SelectedItemTemplate>` element of the `DataList` control, so it will be generated and displayed only for the row (if any) that is currently in selected mode.

In the opening tag, you add the attributes that wire up event handlers for the three events you want to handle: the `EditCommand` event that occurs when an Edit link is clicked, the `UpdateCommand` event that occurs when an Update link is clicked, and the `CancelCommand` event that occurs when a Cancel link is clicked. You also specify the OrderID column from the source row set as the DataKeyField value and turn off autogeneration of columns in the DataGrid control:

```
DataKeyField="OrderID"
OnEditCommand="DoItemEdit"
OnUpdateCommand="DoItemUpdate"
OnCancelCommand="DoItemCancel"
AutoGenerateColumns="False"
```

To create the Edit, Update, and Cancel links in each row, you declare the first column within the <Columns> element of the DataGrid control as an <EditCommandColumn> element. In it, you can set the text that will be displayed for the three links:

```
<asp:EditCommandColumn EditText="Edit"
     CancelText="Cancel" UpdateText="Update" />
```

The rest of the columns for the DataGrid control are declared either as read-only BoundColumn controls like this:

```
<asp:BoundColumn DataField="column-name"
     HeaderText="column-heading" ReadOnly="True" />
```

or as <TemplateColumn> elements that display the value as text when in normal mode or in a TextBox control when in edit mode:

```
<asp:TemplateColumn HeaderText="Ordered">
  <ItemTemplate>
    <%# DataBinder.Eval(Container.DataItem, "OrderDate", _
                        "{0:dd MMM yyyy}") %>
  </ItemTemplate>
  <EditItemTemplate>
    <asp:TextBox Columns="8" id="txtOrderDate"
         runat="server"
         Text='<%# DataBinder.Eval(Container.DataItem, _
                   "OrderDate", "{0:dd MMM yyyy}") %>' />
  </EditItemTemplate>
</asp:TemplateColumn>
```

Populating the DataList Control

You'll recognize much of the code used to populate the DataList control and the nested DataGrid controls because it is very similar to the code in the previous example, where you populate nested DataGrid controls using a DataReader instance. However, one major change in this example is that you are supporting postbacks, to allow the user to show or hide order details and edit them.

The first consequence of this, taking into account the fact that you have enabled viewstate for this page, is that you must be sure to populate the DataList control only when the page first loads and not following a postback.

Listing 4.11 shows the Page_Load event handler for this example, and it contains the functions that create the DataReader instance required to provide the data for the DataList and DataGrid controls. This time, you only need two row sets—the lists of customers and orders—and these are provided by the two functions named GetCustomers and GetOrders. Each one uses the same GetReader function as in the previous example to generate the DataReader instance and return it.

Using Viewstate with List Controls

Not enabling viewstate is a common error newcomers make when using data binding and postbacks with the list controls in ASP.NET. If viewstate is not enabled, the list control will not maintain its state; there will be no values in it after a postback. However, if you repopulate it in the Page_Load event after every postback, the list control may not behave properly. For example, it may not display the selected row or raise events on the server when controls in the grid (such as the Edit links) are activated. The solution is to enable viewstate and only populate the list control in the Page_Load event handler the first time the page is loaded. Afterward, you repopulate the list control only when you change a property such as SelectedIndex or EditIndex, in order to display the rows in the appropriate modes. And you only do so in the event handler that handles the mode change, as you'll see in this example.

LISTING 4.11 The Page_Load Event Handler and Functions That Generate the Row Sets from the Database

```
Sub Page_Load()

  If Not Page.IsPostback Then
    dtl1.DataSource = GetCustomers()
    dtl1.DataBind()
  End If

End Sub


Function GetCustomers() As OleDbDataReader

  Dim sSelect As String _
    = "SELECT CustomerID, CompanyName, City, Country, Phone " _
    & "FROM Customers WHERE CustomerID LIKE 'c%'"
  Return GetReader(sSelect)

End Function
```

LISTING 4.11 Continued

```
Function GetOrders(sKey As String) As OleDbDataReader

  Dim sSelect As String _
    = "SELECT Orders.OrderID, Orders.OrderDate, " _
    & "Orders.RequiredDate, Orders.ShippedDate, Orders.Freight, " _
    & "Shippers.CompanyName As ShipperName " _
    & "FROM Orders JOIN Shippers " _
    & "ON Orders.ShipVia = Shippers.ShipperID " _
    & "WHERE CustomerID='" & sKey & "'"
  Return GetReader(sSelect)

End Function


Function GetReader(sSQL As String) As OleDbDataReader

  ' get DataReader for rows from Northwind tables
  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindOleDbConnectString")
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSQL, oConnect)
    Return oCommand.ExecuteReader(CommandBehavior.CloseConnection)

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text = oErr.Message & "<p />"

  End Try

End Function
```

Populating the DataGrid Control

As each row in the DataList control is bound to its source data, the ItemDataBound event is raised. This causes the BindOrdersGrid event handler that you specified for the OnItemDataBound attribute of the DataList control to execute. Listing 4.12 shows the BindOrdersGrid event handler, and you can see that the first task is (as usual) to examine the row type.

However, in this case, the nested DataGrid control will exist only if the current row in the DataList control is in selected mode, so you check to see whether the row type is ListItemType.SelectedItem. If it is, you get the customer ID from the DataKeys collection, get a reference to the nested DataGrid control in this row, and then bind the DataGrid control to the result of the GetOrders function shown in Listing 4.11. The customer ID is passed to the GetOrders function so that it returns only the order rows for the current customer.

LISTING 4.12 The BindOrdersGrid Event Handler for the ItemDataBound Event

```
Sub BindOrdersGrid(sender As Object, e As DataListItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused the event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  ' only process it if it's the Selected row
  If oType = ListItemType.SelectedItem Then

    ' get value of CustomerID for this row from DataKeys collection
    Dim sKey As String = dtl1.DataKeys(e.Item.ItemIndex)

    ' get a reference to the DataGrid control in this row
    Dim oGrid As DataGrid = CType(e.Item.FindControl("dgr1"), DataGrid)

    ' bind nested "orders" DataGrid to DataReader
    oGrid.DataSource = GetOrders(sKey)
    oGrid.DataBind()

  End If

End Sub
```

Selecting a Row in the DataList Control

You've seen how the nested DataGrid control is populated for the row that is in selected mode. To put the row into this mode, you handle the ItemCommand event of the DataList control. Recall that you included the attribute OnItemCommand="DoItemSelect" in the declaration of the DataList control, so any postback that is initiated by a control within the DataList control will raise the ItemCommand event and execute the DoItemSelect event handler routine.

Listing 4.13 shows the `DoItemSelect` event handler. The first step is to determine which control caused the postback, and you do this by examining the `CommandName` property of the control referenced by the sender argument passed to the event handler. You set this property on the two `ImageButton` controls that display the up and down images in the first column of the `DataList` control.

LISTING 4.13 The Event Handler for the `ItemCommand` Event of the `DataList` Control

```
Sub DoItemSelect(sender As Object, e As DataListCommandEventArgs)

  ' see if it was the Select button that was clicked
  If e.CommandName = "Select" Then

    ' set the SelectedIndex property of the list to this item's index
    dtl1.SelectedIndex = e.Item.ItemIndex
    dtl1.DataSource = GetCustomers()
    dtl1.DataBind()

  End If

  ' see if it was the Un-Select button that was clicked
  If e.CommandName = "UnSelect" Then

    ' set the SelectedIndex property of the list to -1
    dtl1.SelectedIndex = -1
    dtl1.DataSource = GetCustomers()
    dtl1.DataBind()

  End If

End Sub
```

If the down image was clicked (`CommandName="Select"`), you want to put that row into selected mode by setting the `SelectedIndex` property of the `DataList` control to the index of the row. You get the index of the current row from the `ItemIndex` property of the current `DataListItem` instance, set the `SelectedIndex` property, and then repopulate the `DataList` control. The control will automatically display the current row in selected mode by using the contents of the `<SelectedItemTemplate>` element instead of the `<ItemTemplate>` element.

Alternatively, if the `CommandName` property of the control that caused the postback is set to `"UnSelect"`, you know that the user clicked the up button in this row. In this case, you just set the `SelectedIndex` property to `-1` and repopulate the `DataList` control to display all the rows in normal mode.

Editing a Row in the DataGrid Control

If a row in the DataList control is in selected mode, the DataGrid control that displays the orders for the selected customer is visible. The first column of this DataGrid control contains the three links, Edit, Update, and Cancel, depending on whether that DataGrid control row is currently in edit mode. So you have to handle three events that can be raised by the DataGrid control. You specified the event handlers as attributes when you declared the DataGrid control:

```
OnEditCommand="DoItemEdit"
OnUpdateCommand="DoItemUpdate"
OnCancelCommand="DoItemCancel"
```

> **Accessing the Controls in a Row in the DataList Control**
>
> **Each row in a DataList control is represented by a DataListItem instance in the DataListCommandEventArgs object that is passed to the ItemDataBound and ItemCreated event handlers. The DataListItem object is very similar to the DataGridItem object discussed earlier in this chapter. It has the same commonly used members shown in Table 4.1 for the DataGridItem object, with the exception of the DataSetIndex property and the Cells collection (because the individual values in a DataList control are not output as HTML table cells). Likewise, the individual rows in a Repeater control are represented by the RepeaterItem object, which provides a slightly more restricted set of properties.**

The event handlers for the EditCommand event, named DoItemEdit, and the CancelCommand event, named DoItemCancel, are shown in Listing 4.14. The one issue you have to contend with is that the DataGrid control is nested within one of the rows of the parent DataList control. So to get a reference to it, you can search for it within the Controls collection of the row in the DataList control that is currently selected.

LISTING 4.14 The Event Handlers for Switching Into and Out of Edit Mode

```
Function GetDataGridRef() As DataGrid

  ' get a reference to the DataGrid in the selected DataList row
  Dim oRow As DataListItem = dtl1.Items(dtl1.SelectedIndex)
  Return CType(oRow.FindControl("dgr1"), DataGrid)

End Function


Sub DoItemEdit(sender As Object, e As DataGridCommandEventArgs)

  ' get a reference to the DataGrid control in this row
  Dim oGrid As DataGrid = GetDataGridRef()

  ' set the EditItemIndex of the grid to this item's index
  oGrid.EditItemIndex = e.Item.ItemIndex

  ' bind grid to display row in new mode
  ' get CustomerID from the DataKeys collection of the DataList
```

LISTING 4.14 Continued

```
  oGrid.DataSource = GetOrders(dtl1.DataKeys(dtl1.SelectedIndex))
  oGrid.DataBind()

End Sub


Sub DoItemCancel(sender As Object, e As DataGridCommandEventArgs)

  ' get a reference to the DataGrid control in this row
  Dim oGrid As DataGrid = GetDataGridRef()

  ' set EditItemIndex of grid to -1 to switch out of Edit mode
  oGrid.EditItemIndex = -1

  ' bind grid to display row in new mode
  ' get CustomerID from the DataKeys collection of the DataList
  oGrid.DataSource = GetOrders(dtl1.DataKeys(dtl1.SelectedIndex))
  oGrid.DataBind()

End Sub
```

The function named GetDataGridRef shown at the start of Listing 4.14 does this by first getting a reference to the DataListItem object that represents the selected row in the DataList control, using the current SelectedIndex property of the DataList control to locate it. You know that one row must be selected; otherwise, the DataGrid control would not be visible and the user could not have clicked the Edit link or the Cancel link. Then you can use the FindControl method exposed by the selected DataListItem object to locate the DataGrid control.

Using the Sender Argument As a Reference to the Source Control

You may have realized that there is a simpler approach to getting a reference to the nested DataGrid control than is used in this example. In fact, you saw the alternative technique in previous examples in this chapter. You can use the sender argument passed to the event handler instead; this argument is, of course, a reference to the control that raised the event. However, the function provided in this example is intended to demonstrate another way that you can achieve the same result, and it may come in handy in other situations.

Then, in the DoItemEdit routine, you can use the GetDataGridRef function to get a reference to the DataGrid control and set EditItemIndex to the index of the row containing the Edit link that was clicked. To display the grid with this row in edit mode, you repopulate it, using the GetOrders routine shown in Listing 4.11. This requires the ID of the currently selected customer, and you can get that easily enough from the DataList control's DataKeys collection—by specifying the current SelectedIndex value of the DataList control as the row index for the DataKeys collection.

To switch the row out of edit mode when the user clicks the Cancel link, you just get a reference to the DataGrid control (again using the GetDataGridRef function), set EditItemIndex to -1, and repopulate the grid.

The remaining event handler, named DoItemUpdate, is executed when the user clicks the Update link after changing some values in the text boxes within the grid. This is a more complicated routine, although much of the code is concerned with trapping data input errors.

> **Using the UpdateCommand Event**
>
> **Notice that you don't have to worry about what type of row you're dealing with here, as you do when handling the ItemDataBound and ItemCreated events. The UpdateCommand event is only raised for the row that is already in edit mode, so you know that the controls defined in the <EditItemTemplate> section will be present in this row.**

Listing 4.15 shows the complete event handler, and you can see that the first task is to get a reference to the DataGrid control. Then you can get references to each of the TextBox controls in the row by using the FindControl method of the current DataGridItem instance.

LISTING 4.15 The Event Handler for the UpdateCommand Event of the DataGrid Control

```
Sub DoItemUpdate(sender As Object, e As DataGridCommandEventArgs)

  ' get a reference to the DataGrid control in this row
  Dim oGrid As DataGrid = GetDataGridRef()

  ' get a reference to the text boxes
  Dim oOrdered As TextBox _
    = CType(e.Item.FindControl("txtOrderDate"), TextBox)
  Dim oRequired As TextBox _
    = CType(e.Item.FindControl("txtRequiredDate"), TextBox)
  Dim oShipped As TextBox _
    = CType(e.Item.FindControl("txtShippedDate"), TextBox)
  Dim oFreight As TextBox _
    = CType(e.Item.FindControl("txtFreight"), TextBox)

  ' verify that the values are valid
  Dim dOrderDate, dRequDate, dShipDate As DateTime
  Dim cFreight As Decimal
  Try
    dOrderDate = DateTime.Parse(oOrdered.Text)
  Catch
    lblErr.Text = "ERROR: Invalid value entered for Order Date"
    Exit Sub
  End Try
  Try
    dRequDate = DateTime.Parse(oRequired.Text)
  Catch
    lblErr.Text = "ERROR: Invalid value entered for Required Date"
    Exit Sub
```

LISTING 4.15 Continued

```
  End Try
  Try
    dShipDate = DateTime.Parse(oShipped.Text)
  Catch
    lblErr.Text = "ERROR: Invalid value entered for Shipped Date"
    Exit Sub
  End Try
  Try
    cFreight = Decimal.Parse(oFreight.Text)
  Catch
    lblErr.Text = "ERROR: Invalid value entered for Freight Cost"
    Exit Sub
  End Try

  ' create a suitable SQL statement and execute it
  Dim sSQL As String
  sSQL = "UPDATE Orders SET OrderDate='" _
       & dOrderDate.ToString("yyyy-MM-dd") & "', " _
       & "RequiredDate='" _
       & dRequDate.ToString("yyyy-MM-dd") & "', " _
       & "ShippedDate='" _
       & dShipDate.ToString("yyyy-MM-dd") & "', " _
       & "Freight=" & cFreight.ToString() & " " _
       & "WHERE OrderID=" & oGrid.DataKeys(e.Item.ItemIndex)
  ExecuteSQLStatement(sSQL)

  ' set EditItemIndex of grid to -1 to switch out of Edit mode
  oGrid.EditItemIndex = -1

  ' bind grid to display row in new mode
  ' get CustomerID from the DataKeys collection of the DataList
  oGrid.DataSource = GetOrders(dtl1.DataKeys(dtl1.SelectedIndex))
  oGrid.DataBind()

End Sub
```

The code in Listing 4.15 extracts the values from the four `TextBox` controls, using a `Try...Catch` construct to detect invalid values and catch errors. If an invalid data type conversion occurs for the `Parse` method, the `Catch` section of each construct displays the error message in a `Label` control located above the `DataList` control in the page and prevents further processing by exiting from the event handler routine. Figure 4.9 shows the result when an invalid value is detected.

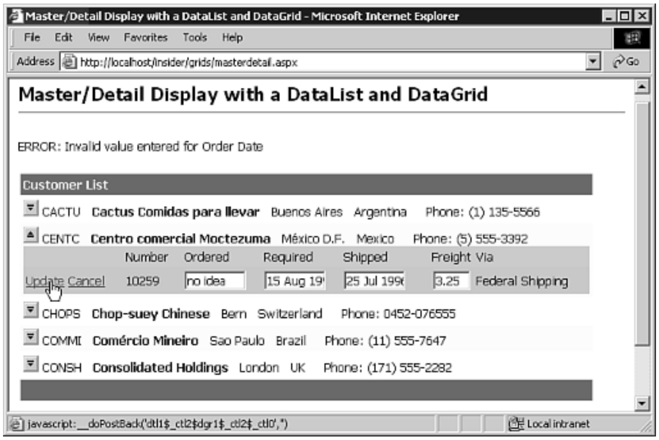


FIGURE 4.9
Catching data entry errors and invalid values in the master/detail sample page.

Next, the routine builds up a SQL statement. It uses the values from the `TextBox` controls, together with the current order ID extracted from the `DataKeys` collection of the current `DataGrid` control. This SQL statement is passed to a separate routine named `ExecuteSQLStatement`, which we'll look at shortly. Of course, you could use a stored procedure to update the database if preferred.

Finally, you switch the current row in the `DataGrid` control out of edit mode and repopulate it to display the updated values.

> **Concurrent Update Checking**
>
> **Notice that you don't perform full concurrent update error checking here. If the data is updated by another user while the page is displayed, the second user's changes will be overwritten. To avoid this, you would have to check the existing value in every column of the row in the database against its original value when the page was first displayed. This is easier to do when the data you use to populate the page is held in a `DataSet` instance. With a `DataReader` instance (as in this example), you would probably decide to store the original values in hidden controls in the row that is in edit mode or use a timestamp or GUID column in the database that indicates whether the row has been changed concurrently.**

Updating the Original Data in the Database

The final section of code in the sample page is the `ExecuteSQLStatement` routine, shown in Listing 4.16. There's nothing new or exciting here: You just create a `Connection` instance and a `Command` instance, open the `Connection` instance, and execute the SQL statement by calling the `ExecuteNonQuery` method. If it doesn't update just one row, or if an error occurs, you display a suitable error message.

LISTING 4.16 The Routine to Push the Updates Back into the Database

```
Sub ExecuteSQLStatement(sSQL)

  ' execute SQL statement against the original data source
  Dim sConnect As String = ConfigurationSettings.AppSettings( _
                           "NorthwindOleDbConnectString")
```

LISTING 4.16 Continued

```
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSQL, oConnect)
    If oCommand.ExecuteNonQuery() <> 1 Then
      lblErr.Text &= "ERROR: Could not update the selected row"
    End If
    oConnect.Close()

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text &= "ERROR: " & oErr.Message & "<p />"

  End Try

End Sub
```

Summary

The topic covered in this chapter is quite narrow, focusing only on the use of nested list controls in ASP.NET pages. However, as you've seen, there are plenty of issues to understand, several interesting problems to solve, and a great many options for how to go about the process.

This chapter describes how to use a `DataSet` instance or a `DataReader` instance and discusses the performance implications. It also shows how you can perform the binding declaratively to a function or by handling the `ItemDataBound` event and generating the row set you need that way. And, as mentioned previously, you can mix and match the techniques and the data sources in almost any combination to achieve the desired end result.

As well as addressing four basic techniques, this chapter looks at the nature of the objects that are available in the event handlers, such as the `DataGridItem` and `DataListItem` objects. It is vital that you understand what they offer and how to get the most from them. When you nest list controls, which event is being raised and how to handle it can quickly become confusing.

One issue that is mentioned a couple times in this chapter and that often causes problems as you develop pages that use complex combinations of list controls is that you must be sure your event handlers test what type of row they are handling. Bear in mind that the `FindControl` method cannot detect errors in your code at compile time because it only searches for controls at runtime and silently returns `null` (`Nothing` in Visual Basic .NET) if it can't find the control it's looking for. The result is a runtime error that can be hard to track down.

This chapter finishes up with a look at how a combination of list controls, in this case a `DataList` control and a `DataGrid` control, can be used to build collapsible master/detail pages with very little effort. And, along the way, this chapter discusses more ways for detecting the source of events and postbacks and managing the edit process inside a list control.

PART II

Reusability

5

Creating Reusable Content

IN THIS CHAPTER

Although the general public's view of computer programmers as a breed apart might be less than complimentary, we are really no different from any other people when it comes to having a hatred of dull, repetitive work. When writing code, experienced programmers are constantly on the lookout for ways to encapsulate chunks that are reusable and save the effort of having to write the same code repeatedly. Subroutines and functions are obvious examples of ways to do this within a single application; components, DLLs, and .NET assemblies provide the same kind of opportunities across different applications.

However, when building Web pages and Web-based interfaces for your applications, it can be difficult to choose the obvious or the most efficient approach for creating reusable content. Traditional techniques have been to read from disk-based template files and to use disk-based include files that rely on the server-side include feature of most Web server systems.

Of course, the use of external code in the form of COM or COM+ components, and in ASP.NET, the use of .NET assemblies, is also prevalent in Web pages. However, the complexity of the plumbing between

COM/COM+ components and the host application has never really been an ideal approach when working with Web pages that have extremely short transitory lifetimes on the server. These components work much better when instantiated within an executable application where they have a longer lifetime.

In ASP.NET, the ideal solution from a component point of view is to use native .NET managed code assemblies. These are, of course, the building blocks of ASP.NET itself, and they provide the classes that implement all the ASP.NET controls we use in our pages. However, the .NET Framework provides several techniques that are extremely useful and efficient and that can provide high levels of reuse for interface declarations and runtime code.

Techniques for Creating Reusable Content

Before delving too deeply into any of the specific techniques for creating reusable content, we'll briefly summarize those that are commonly used within ASP.NET Web applications:

- Server-side include files
- ASP.NET user controls
- Custom master page and templating techniques
- ASP.NET server controls built as .NET assemblies
- Using COM or COM+ components via COM Interop

Server-Side Include Files

Many people shun the use of server-side includes in ASP.NET, preferring to take advantage of one of the newer and flashier techniques that are now available (such as user controls, server controls, and custom templating methods). However, server-side includes are just as useful in ASP.NET as they are in "classic" ASP. They are also more efficient than in ASP because ASP.NET pages are compiled into an assembly the first time they are referenced, and this assembly is then cached and reused automatically until the source changes.

As long as none of the files on which an assembly is dependent change (this applies to things like other assemblies and user controls as well as to server-side include files), the page will not be recompiled. This means that the include process will be required only the first time the ASP.NET page is referenced, and it will not run again until recompilation is required. The content of the include file becomes just a part of the assembly.

> **Using Server-Side Include Files to Insert Code Functions**
>
> Remember that you aren't limited to just using text and HTML in a server-side include file. You can place client-side and server-side code into it and, in fact, you can put in it any content that you can use in an ASP.NET page. This means you can, for example, place just code routines into a server-side include file and then call those functions and subroutines from other code in the main hosting page, or you can even call them directly from control events. However, you can only include files that are located within the same virtual application as the hosting page.

Of course, the same include file is likely to be used in more than one page. Any change to that file will therefore cause all the assemblies that depend on it to be recompiled the next time they are referenced. This makes include files extremely useful for items of text or declarative HTML that are reused on many pages but that change rarely. An example is a page footer containing the Webmaster's contact details and your copyright statement.

Including Dynamic Text Files in an ASP.NET Page

Another area where server-side include files are useful is where you have some dynamically generated text or HTML content that you want to include in a Web page.

One particular example we use ourselves is to remotely monitor the output generated by a custom application that executes on the Web server. It generates a disk-based log file as it runs and allows the name and location of the log file to be specified. We place the log file in a folder that is configured as a virtual Web application root and then insert it into an empty ASP.NET page by using a server-side include statement (see Listing 5.1).

LISTING 5.1 Including a Log File in an ASP.NET Page

```
<%@Page Language="VB" %>
<html>
<body>
<pre>
<!-- #include file="myappruntime.log" -->
</pre>
</body>
</html>
```

Downsides of the Server-Side Include Technique

Although server-side includes are useful, there are at least a couple issues to be aware of with them. The first is one that has long annoyed users of classic ASP. The filename and path of the include file cannot be accessed or changed dynamically as the page executes. This is because the `#include` directive is processed before ASP.NET gets to see the page. You can't decide, for example, which file to include at runtime.

However, you can change the content of the section of the page that is generated from a server-side include file at runtime by including ASP.NET control declarations within the file and setting the properties of these controls at runtime. For example, if the include file contains the code shown in Listing 5.2, you can make the Webmaster's email address visible or hide it by setting the `Visible` property of the `Panel` control at runtime, as shown in Listing 5.3.

LISTING 5.2 Server-Side Include Files Containing ASP.NET Server Controls

```
&copy;2004 Yoursite.com - no content reproduction without permission
<asp:Panel id="WebmasterPanel" runat="server">
  <a href="mailto:webmaster@yoursite.com">webmaster@yoursite.com</a>
</asp:Panel>
```

LISTING 5.3 Setting Properties of Controls in a Server-Side Include File at Runtime

```
<!-- #include file="myfooter.txt" -->
...
<script runat="server">
Sub Page_Load()
  If (some condition) Then
    WebmasterPanel.Visible = True
  Else
    WebmasterPanel.Visible = False
  End If
End Sub
</script>
```

> **When Is an Include File Actually Included?**
>
> **Listings 5.2 and 5.3 prove that the include file is inserted into the page before ASP.NET gets to see it. The code in Listing 5.3 should produce a compile error and report that it can't find the control with ID `WebmasterPanel` because the declaration of this control is not in the page. However, by the time ASP.NET gets to compile the page, the include file has already been inserted into it.**

Designer Support for Server-Side Include Files

The second issue with using server-side include files is that they are rarely supported in the tools that are available to help build pages and sites. This doesn't mean that you can't use them, but it does mean that you're unlikely to get WYSIWYG performance from the tool. However, this may not be important for things like footers or other minor sections of output.

ASP.NET User Controls

The server-side include approach we just discussed is useful and works well with ASP.NET. But there are other ways to build reusable content, and these techniques often overcome the limitations of server-side include files and also offer a better development model as a whole. The simplest, and yet extremely powerful, approach introduced with ASP.NET is the concept of user controls.

Whereas server-side include files are effectively just chunks of content that get inserted into the page before it is processed by ASP.NET, user controls are control objects in their own right. The `System.Web.UI.UserControl` class that is used to implement all user controls is descended from the same base class (`System.Web.UI.Control`) as all the server controls in ASP.NET.

This means that a user control is instantiated by ASP.NET and becomes part of the control tree for the page. It also means that it can implement and expose properties that can be accessed by other controls and by code written within the hosting page. And, because it is part of the control tree, any other server controls that it contains can also be accessed in code within the hosting page, as well as by code within the user control itself (see Figure 5.1).

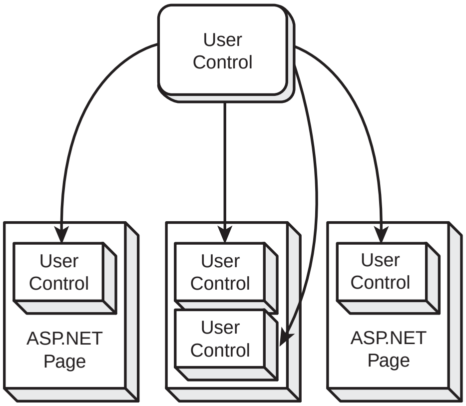


FIGURE 5.1 Reusing user controls in multiple ASP.NET pages.

Registering and Inserting a User Control

A user control is written as a separate file that must have an `.ascx` file extension. It is then registered with any page that needs to use it, via the `Register` directive. The `Register` directive specifies the tag (element) prefix and name that will identify the user control within the page, and this prefix and name are then used to instantiate the user control at the required position within the declarative content of the page, as shown in Listing 5.4.

LISTING 5.4 Registering a User Control and Inserting It into a Page

```
<%@Page Language="VB" %>
<%@Register TagPrefix="ahh" TagName="ComboBox" Src="ascx\combo.ascx" %>
...
<body>
Simple Combo List Box:
<ahh:ComboBox id="cboTest1" IsDropDownCombo="False" runat="server" />
...
</body>
```

You can see in Listing 5.4 how similar the technique for using a user control is to using the standard server controls that are provided with ASP.NET. All the properties of the `System.Web.UI.Control` class are available (for example, `id`, `EnableViewState`, `Visible`) and can be set using attributes or at runtime in your code. The `id` property is set to `"cboTest1"` in Listing 5.4.

You can set the values of properties that are specific to this user control in exactly the same way. For example, Listing 5.4 shows the value of the `IsDropDownCombo` property being set to `False`. And any `Public` methods that the user control exposes can be executed from code in the hosting page, just as with a normal server control. Figure 5.2 shows a page that hosts the `ComboBox` user control you'll develop later in this chapter.

> **Running the `ComboBox` Control Example Online**
>
> **If you want to try out this control, go to the sample pages for this book. You can also run it online on our own server, at `www.daveandal.net/books/6744/combobox/combo.aspx`.**

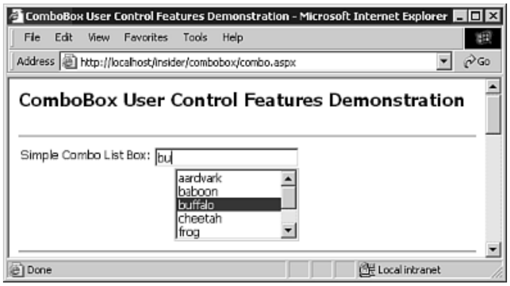


FIGURE 5.2 A `ComboBox` control implemented as a user control.

> **Nesting User Controls**
>
> Note that you can't insert an instance of the same user control into itself. The nested user control would then insert another instance of itself again, ad infinitum, creating a circular reference. The compiler would detect this situation and generate an error. If you need to nest user controls, you must create a hosting instance that references a different file that is identical in content except that it does not contain the reference to the nested control.

The Contents of a User Control

As with server-side includes, you can place almost any content in a user control. It can be just declarative HTML or client-side code and text, or it can include ASP.NET server controls, server-side code, and even other user controls.

Oftentimes you need to insert the same user control more than once into a page, in the same way that you use server controls. Of course, this isn't obligatory, but it does mean that you need to bear in mind some obvious limitations to the content user controls include if you are to use them more than once. There are two things you should generally not include in a user control:

- The opening and closing `<html>`, `<title>`, or `<body>` elements. These should be placed in the hosting page so that they occur only once.
- Server-side form controls (for example, `<form>` elements that contain the `runat="server"` attribute). There can be only one server-side form on an ASP.NET page (except when you're using the `MobilePage` class to create pages suited to mobile devices).

A common scenario is to use a user control that generates no user interface (no visible output) but exposes code functions or subroutines that you want to be able to reuse in several pages. As long as these routines are marked as Public, they will be available to code running in the hosting page—which can reference them through the ID that is assigned to the user control. Listing 5.5 shows how you can access a method of a user control (which in this case just returns a value) and how you can set and read property values. Later in this chapter, you'll see in more detail how properties and methods are declared within a user control.

LISTING 5.5 Accessing Properties or Methods of a User Control

```
' call the ShowMembers method and get back a String
Dim sSyntax As String = cboTest1.ShowMembers()
' set the width and number of rows of the control
cboTest1.Width = 200
```

LISTING 5.5 Continued

```
cboTest1.Rows = 10
' read the current text value of the control
Dim sValue As String = cboTest1.Text
```

User Controls and Output Caching

One extremely good reason for taking advantage of user controls (and, in fact, perhaps one of the prime reasons for their existence) is that they can be configured differently from the hosting page as far as the page-level directives are concerned. In an ASP.NET page, you can add a range of attributes to the `Page` directive and use other directives, such as `OutputCache`, to specify how the page should behave. This includes things like whether debugging and tracing are enabled, whether viewstate is supported, and how output caching should be carried out for the page.

The simplest output cache declaration specifies the number of seconds for which the output generated by ASP.NET for the page should be cached and reused, and it specifies which parameters sent to the page can differ to force a new copy to be generated. When you use an asterisk (*) for the `VaryByParams` attribute, a different copy of the page will be cached for each varying value sent in the `Request` collections (`Form` and `QueryString`):

```
<%@OutputCache Duration="300" VaryByParam="*" %>
```

Output caching provides a huge performance benefit when the content generated by the page is the same for most clients or when there are only a limited number of different versions of the page (in other words, when the values sent in the `Form` and `QueryString` collections fall into a reasonably small subset). When there are many different cached versions, the process tends to be self-defeating.

> **Managing Caching Individually for User Controls**
>
> **User controls allow you to divide a page into sections and manage output caching individually for each section. This means that you can cache the output for sections that change rarely (or for which there are few different versions) for longer periods, while caching other sections for shorter periods or not at all.**

The `OutputCache` directive can be declared in a user control, just as it can in a normal ASP.NET page, but it affects only the output generated by the user control. There is also one extra feature supported by the `OutputCache` directive when used in a user control: the `Shared` attribute.

User controls are designed to be instantiated within more than one ASP.NET page, and yet it's reasonable to suppose that the output they generate could be the same in many cases (regardless of the page that uses them). When the `OutputCache` directive in a user control includes the attribute `Shared="True"`, the same cached output is used for all the pages that host this user control. This saves memory and processing when the output required is the same for all the pages that use the control.

The Downsides of User Controls

Although user controls provide a great development environment for reusable content, they also have a couple of downsides that you must consider. The first and most obvious of these is that they are specific to an ASP.NET application. Unlike the standard ASP.NET server controls, which can be used in any ASP.NET application on a server, user controls can only be instantiated in pages that reside in the same Web application (the root folder of the virtual application, as defined in Internet Services Manager, or a subfolder of this application that is not also defined as a virtual application).

In most cases, this is not a real problem. User controls tend to be specific to an application. For example, if you implement a footer section for all your pages as a user control, it probably makes sense for it to be used only within that application. However, some user controls (such as the `ComboBox` control shown earlier in this chapter) may be useful in many different applications. In this case, you will have to maintain multiple copies of the same user control—one for each application that requires it.

Furthermore, many people still tend to see user controls as being the "poor man's solution" for building controls, as in the `ComboBox` example earlier in this chapter. There are good reasons for this: One is that you can't expose events from a user control in the same way you can from a server control that is defined as a class and compiled into an assembly. We'll look at this topic in Chapter 8, "Building Adaptive Controls."

Finally, of course, you can't hide your code in a user control in quite the same way as you can by compiling a server control into an assembly. Like an ASP.NET page, the source of a user control is just a text file that must be present in the Web site folders. It's unlikely that you could build up your own software megacorporation just by selling user controls.

Custom Master Page and Templating Techniques

One common use of both server-side include files and user controls is to insert some common section of content into a page, perhaps to create the page header, the footer, or a navigation menu. There is, however, a technique that effectively tackles this issue from the opposite direction: You can create a *master page* or *template* for the site and base all the pages on this master page or template. All the content in the master page or template then appears on every page, and each individual page only has to implement the content sections that are specific to that page.

The master page approach tends to encompass the concept of the individual pages being dynamically generated each time from the master page, with the individual content sections being inserted into it (see Figure 5.3). However, bear in mind that ASP.NET pages are compiled on first hit and then cached, so the process happens only the first time the page is referenced and when the source of the page (the master page itself, or the individual content sections) changes.

A template, on the other hand, usually conjures up a vision of a single page from which the individual content pages are generated in their entirety—rather like some kind of merge process (see Figure 5.4). In fact, using master pages and using templates are generically very similar, and both produce compiled pages that are cached for use in subsequent requests.

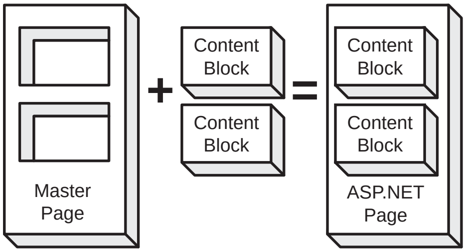


FIGURE 5.3 Generating ASP.NET pages from a master page.

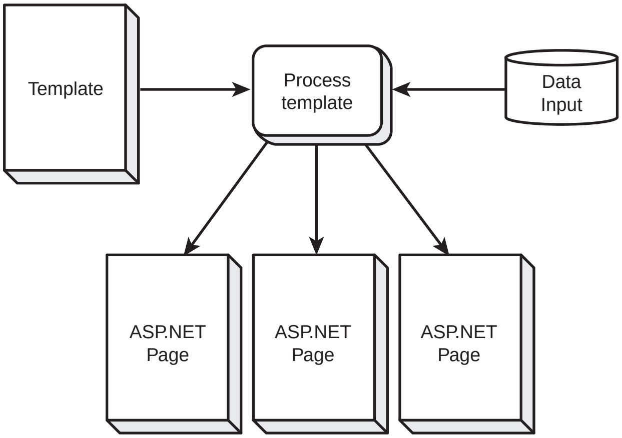


FIGURE 5.4 Generating ASP.NET pages from a template.

Chapter 9, "Page Templates," looks at master pages and page templates; you'll see more discussion there of the different techniques you can use and the various ways you can code pages to provide the most efficient and extensible solutions.

ASP.NET Server Controls Built As .NET Assemblies

The next step up the ladder of complexity versus flexibility is to create reusable content as a native .NET server control. The controls you create using this technique are functionally equivalent, in terms of performance and usability, to the standard server controls provided with ASP.NET. The controls provided in the box with ASP.NET are written in C#, and they're compiled into assemblies. The ASP.NET Web Forms controls (those prefixed with `asp:`) are all implemented within the assembly named `System.Web.dll`, which is stored in your `%windir%\Microsoft.NET\Framework\[`*`version`*`]\` folder.

Subsequent chapters show how easy it is to create your own server controls and then use them in Web pages just as you would the standard ASP.NET controls. Figure 5.5 shows the `SpinBox` control that is created in Chapter 8, with three instances inserted into the page and various styles applied to them.

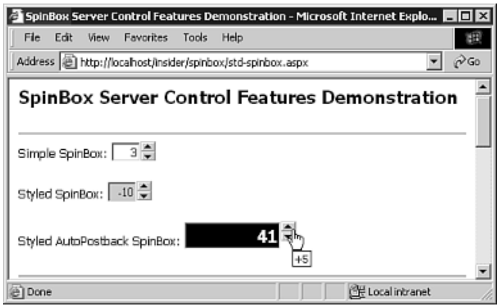


FIGURE 5.5 A `SpinBox` control implemented as a .NET server control.

Server controls provide a few important advantages over user controls and most other reusable content methods. They encapsulate the code and logic, making it harder for others to steal any intellectual property they contain. Although server controls can still be disassembled to view the Microsoft Intermediate Language (MSIL) code they contain, most users are unlikely to be able to see how they work. You can also use obfuscation techniques (as built into Visual Studio) to make it much more difficult for even experienced users to discover the working of a control.

Second, user controls can expose events that you can handle in the hosting page, exactly as the standard ASP.NET Web Forms controls do. For example, the `SpinBox` control exposes an event named `ValueChanged`, which can be handled by assigning an event handler to the `OnValueChanged` attribute of the control, as shown in Listing 5.6.

LISTING 5.6 Handling the `ValueChanged` Event of the `SpinBox` Control

```
<ahh:StandardSpinBox id="spnTest1" runat="server"
                     OnValueChanged="SpinValueChanged" />
...
...
Sub SpinValueChanged(sender As Object, e As EventArgs)
  ' display message when value of control has changed
  lblResult.Text &= "Detected ValueChanged event for control " _
                 & sender.ID & ". New value is " _
                 & sender.Value.ToString()
End Sub
```

Third, server controls can be installed into the global application cache (GAC) so that they are available to all applications on the machine and not restricted to a single application, as are user controls and server-side include files. The following section looks at this particular topic in more detail.

Local and Machinewide Assembly Installation

In many cases, when you build custom controls as assemblies, you'll probably want to use them only within the ASP.NET application for which they were designed. As long as the assembly resides in the `bin` folder of the application, it will be available to any ASP.NET page (or Web service or other resource) that references it. All you need to do is add to the page an appropriate

`Register` directive that specifies the tag prefix for elements that will declare instances of the control, the namespace in the assembly within which the control is declared, and the assembly filename, without the `.dll` extension:

```
<%@ Register TagPrefix="ahh" Namespace="Stonebroom"
             Assembly="std-spinbox" %>
```

You can then add an instance of the control to the page, using the following:

```
<ahh:SpinBox id="spnTest" runat="server" />
```

However, as just mentioned, you can make a control or an assembly available machinewide by installing it in the GAC. For a control to be available to all the applications on the machine, three major requirements must be met:

- There must be a way for a control to be uniquely identifiable among all other controls, aside from its name. Because the assemblies that implement controls can be installed anywhere on the machine, the filename of the assembly is not sufficient to uniquely identify it.
- There must be a way to specify the version of the control so that new versions can be installed for applications that require them, while the existing version can remain in use for other applications.
- The .NET Framework requires that assemblies must be digitally signed using public key encryption techniques to protect the assemblies from malicious interference with the code.

You can meet all three of these requirements by applying a strong name to an assembly. You create a strong name by using a utility named `sn.exe` to generate a public encryption key pair, and then you add attributes to the assembly before it is compiled to attach this key pair to the assembly and specify the version, the culture, and optionally other information.

After the assembly has been compiled, you can add it to the GAC by using the `gacutil.exe` utility, the .NET Framework Configuration Wizard, or Windows Installer. Finally, ASP.NET pages that use the control must include a `Register` directive that specifies the assembly name, version, culture, and public key. For example, this is how you would register the version of the `SpinBox` control that is inserted into the GAC (and which has the name `GACSpinBox`):

> **A Note About the Assembly Attribute**
>
> **Important: The text string specified for the `Assembly` attribute of the `Register` directive must all be on one line and not broken as it is here due to the limitation of the page width.**

```
<%@Register TagPrefix="ahh" Namespace="Stonebroom"
            Assembly="GACSpinBox,Version=1.0.0.0,Culture=neutral,
            ➥ PublicKeyToken=92b16615bf088252" %>
```

In Chapter 8 you'll build the `SpinBox` server control you've seen in this chapter. At that point, you'll walk through the process, step-by-step, of making a server control globally available across applications.

The Downside of ASP.NET Server Controls

The only real limitation with building server controls is that you really have to know at least the basics of how your chosen language supports and implements features such as inheritance. You also need to understand the event sequence and the life cycle of controls. However, to quote that oft-used saying, "it's not rocket science." You can quickly pick up the knowledge you require.

Using COM or COM+ Components via COM Interop

Using components is a great way to provide encapsulated and reusable content, as you've seen in the preceding sections of this chapter. So far this chapter has talked about various types of components (using the word in the broadest sense) that are all fully compatible with ASP.NET. However, you may have COM or COM+ components that you are already using in a classic ASP application, or you might want to use COM components that are part of Windows or an application you have already installed in an ASP.NET application.

To use COM or COM+ components within the .NET Framework, you can create a wrapper that exposes the interface in a format that allows managed code to access it. You effectively create a .NET manifest that describes the component and that acts as a connector between the component and the .NET runtime environment. Each property, method, and event is mapped through the wrapper, and you can then use the component in the same way you would use a fully managed code (.NET) assembly.

The overall process is referred to as *COM Interop*, and it provides a path to move to .NET without having to rewrite all the business logic and custom components required in an existing or new application immediately, although you should consider this to be a temporary measure and aim to build native components as part of the process when and where possible.

> **Performance Issues with COM Interop**
>
> **Using wrapped COM components affects the performance of your pages. The extra marshaling of values across the managed/unmanaged boundary with each property setting and method call is less efficient than with a native managed code component. The actual performance degradation generally depends on the number of calls you have to make when using the component; for example, a component that requires you to set a dozen property values and then call a method is likely to degrade performance more than one that lets you make a single method call with a dozen parameters. The actual marshaled size of the parameters or values you pass to properties and methods also has some effect on the performance.**

Creating a .NET Wrapper for a COM or COM+ Component

If you are building an application by using Visual Studio .NET, you can create a type library wrapper by simply adding to your project a reference to the component. You right-click the References entry in the Solution Explorer window and select Add Reference. In the Add Reference dialog that appears, you go to the COM tab and select the component or library you want to use.

Alternatively, you can use the Type Library Import utility provided with the .NET Framework. The utility `tlbimp.exe` is installed by default in the `Program Files\Microsoft.NET\SDK\[version]\Bin` folder. To use it, you specify the COM component DLL name and add any options you want to control specific features of the wrapper that is created. You can find a full list of these options in the locally installed .NET SDK at `ms-help://MS.NETFrameworkSDKv1.1/cptools/html/cpgrftypelibraryimportertlbimpexe.htm` or by searching for `tlbimp` in the index.

Using the `tlbimp` Utility

As an example of how to use the Type Library Import utility provided with the .NET Framework, let's look at an example of how to create a wrapper for a fictional custom COM component. The DLL is named `stnxsltr.dll`, and it implements a class named `XslTransform` within the namespace `Stonebroom`. To create the wrapper, you would copy the DLL to a temporary folder and navigate to this folder in a command window. The following command runs the `tlbimp` utility for version 1.1 of the Framework and generates the type library wrapper as a .NET assembly with the `.dll` file extension:

```
"C:\Program Files\Microsoft.NET\SDK\v1.1\Bin\tlbimp" stnxsltr.dll
```

Notice in Figure 5.6 that the name of the new DLL is the name of the namespace declared within the component, not the filename of the original component DLL. This is required to allow ASP.NET to find the type library when it is imported into a page.

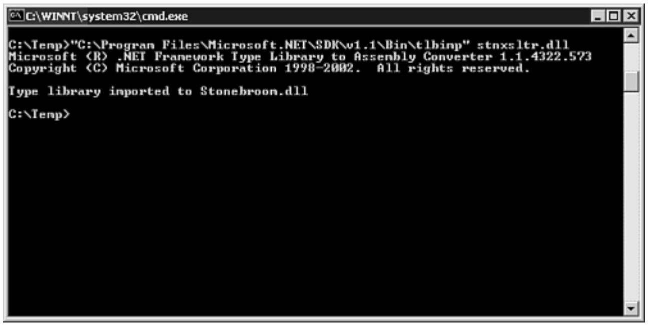


FIGURE 5.6
Executing the `tlbimp` utility to generate a wrapper for a COM component.

Now you would copy the new wrapper DLL into the `bin` folder of an application and use the component in ASP.NET pages just as you would a native .NET component. You'd use an `Import` directive to import the type library wrapper, and then instantiate the component by using the classname. You could use the full *`namespace.classname`* syntax when instantiating the component, but this is not actually required. Because the namespace has been imported, you could instantiate the component by using just the classname (see Listing 5.7).

LISTING 5.7 Using a Custom `XslTransform` COM Component in ASP.NET

```
<%@Import Namespace="Stonebroom" %>
<script runat="server">
Sub Page_Load()
```

LISTING 5.7 Continued

```
  Dim oXml As New XslTransform
  Dim sStatus As String
  Dim sXMLFile As String = "/data/xml/myfile.xml"
  Dim sXSLFile As String = "/data/xsl/myfile.xsl"
  Dim sOutFile As String = "/results/myfile.html"
  Dim blnWorked As Boolean = oXml.TransformXML(sXMLFile, _
                             sXSLFile, sOutFile, sStatus)
  lblResult.Text = sStatus
End Sub
```

Coping with Classname Collisions

The fictional custom component described here has the same classname, `XslTransform`, as a native .NET class within the .NET Framework class library. However, you do not import the `System.Xml.Xsl` namespace (within which the .NET Framework component lives) into the page, so there is no collision of classnames. If there were, you would get a compilation error such as "XslTransform is ambiguous, imported from the namespaces or types System.Xml.Xsl, Stonebroom." In that case, you would use the full *`namespace.classname`* syntax to identify which class you require (for example, `Dim oXml As New Stonebroom.XslTransform`).

ASP Compatibility for Apartment-Threaded COM Components

When you're using COM or COM+ components, one issue to be aware of is that the threading model used in ASP.NET is not directly compatible with components that are single threaded or apartment threaded. Single-threaded components are not suitable for use in ASP or ASP.NET anyway, so this factor should not be an issue.

However, components built with Visual Basic 5 and 6 are usually apartment threaded (via the single-threaded apartment [STA] model) and work fine with only minor performance degradation in classic ASP. Until the arrival of the .NET Framework, which makes creating components in any managed code language easy, Visual Basic was quite a popular environment for building business components and server controls.

To overcome any issues with running apartment-threaded components in ASP.NET, you should always add the attribute `ASPCompat="True"` to the `Page` directive. This forces ASP.NET to adopt a threading model that matches the requirements of Visual Basic apartment-threaded components. It also allows components to access the intrinsic ASP objects, such as `ObjectContext`, and the `OnStartPage` method. There is some performance degradation, but it is not usually significant except in highly stressed Web applications and Web sites.

However, if you add the `ASPCompat="True"` attribute to a page that creates instances of apartment-threaded components before the request is scheduled, you will encounter much more significant performance degradation. You should always create instances of any apartment-threaded components you need in a `Page` event such as `Page_Load` or `Page_Init`.

Building a ComboBox User Control

In the rest of this chapter, you'll see how to build a user control that implements some useful features you can make available in the pages of a Web application. You'll build the ComboBox control you saw earlier in this chapter and then see how it can be used in an ASP.NET page just as a native .NET Web Forms control would be used.

The combo box style of control is one of the most significant omissions from the standard set of controls that are implemented in a Web browser. There is no single HTML element you can use to create one, so you have to build up the complete interface to represent the features you want, using separate HTML control elements. The first step is to consider the requirements for the control and the HTML you will have to generate to produce the final effect you want in a Web page.

Design Considerations

It's usual for a combo box control to offer two modes of operation. The simplest is a combination of a text box and a list control, linked together so that a user can type a value in the text box or select a value from the list. Typing in the text box automatically scrolls and selects the first value in the list that matches the text in the text box, whereas selecting from a list places that value into the text box (see Figure 5.7).

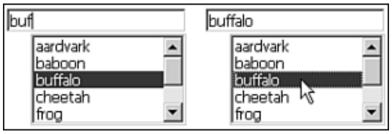


FIGURE 5.7 A standard ComboBox control, showing a user typing in a value and selecting from the list.

The easiest way to create this kind of output in a Web browser is to use a single-cell table to restrain the two controls and include a `<br />` element to force the list to wrap to the next line. By right-aligning the contents of the table cell and controlling the width of the text box and list control with the CSS `width` selector at runtime, you get the appearance you want (see Listing 5.8).

LISTING 5.8 The HTML Required to Implement the Simple ComboBox Control Shown in Figure 5.7

```
<table border="0" cellpadding="0" cellspacing="0">
<tr><td align="right">
<asp:TextBox id="textbox" runat="server" /><br />
<asp:ListBox id="listbox" runat="server" />
</td></tr>
</table>
```

The HTML for a Drop-Down Combo Box

In the second mode of operation of a combo box control, the list is normally hidden and appears only when the user wants to select from it rather than type a value in the text box. The actual behavior of this kind of control varies to some extent, but you might want to implement it so that the user clicks the down button at the right end of the text box to show (drop down) the list. As with the simple combo box shown in the preceding section, selecting a value in the list places that value in the text box.

With this type of combo box, when the list is open, the down button changes to an up button that can be used to close the list again. As the user types in the text box, the first matching item in the list is selected. If the user selects a value in the list, it automatically closes, and that value is copied to the text box (see Figure 5.8).

FIGURE 5.8 The drop-down ComboBox control, showing a user typing in a value and opening the list, as well as the result of selecting from the list.

Obviously, the HTML required to implement this version of the control is more complex than that for the other version of the control; also, it depends on the browser's support for advanced features, such as CSS2. In particular, you need to be able to show and hide the list control, change the image that is displayed for the down and up buttons, and position the elements within some kind of container.

To create this type of control, you can use a `<div>` element as the container and CSS absolute positioning to fix the elements in the correct position. You can also use the CSS `display` selector to show and hide the list control. As with the simple `ComboBox` control, the handling of user interaction is carried out through client-side JavaScript to provide the best possible user experience, rather than posting back to the server with each user interaction.

The HTML used to create the drop-down control, shown in Listing 5.9, declares the enclosing `<div>` element with the `position:relative` style selector so that it acts as a positioning container. Within it are the declarations of the ASP.NET Web Forms controls that will implement the text box, the image button (an `<input type="image">` element), and the list box.

LISTING 5.9 The HTML Required to Implement the Drop-Down `ComboBox` Control Shown in Figure 5.8

```
<div id="dropdiv" Style="position:relative" HorizontalAlign="Right" runat="server">
<asp:TextBox Style="vertical-align:middle" id="textbox2" runat="server"
/><asp:ImageButton id="dropbtn" BorderWidth="0" Width="16" Height="20"
                   Style="vertical-align:middle"
                   ImageUrl="~/images/click-down.gif" runat="server" /><br />
<asp:ListBox Style="display:none;position:absolute;left:20;top:25"
             id="dropbox" runat="server" />
<asp:Image id="imageup" ImageUrl="~/images/click-up.gif"
```

LISTING 5.9 Continued

```
            Style="display:none" runat="server" />
<asp:Image id="imagedown" ImageUrl="~/images/click-down.gif"
            Style="display:none" runat="server" />
</div>
```

The list box is absolutely positioned under the text box, shifted 20 pixels to the right. You can adjust the width of the text box and list control dynamically at runtime, using the `width` style selector, after you find out how wide it needs to be from the settings made by code or declarations in the hosting page.

There are also two `Image` controls, which contain the up and down images used by the image button. They are both declared with the `display:none` style selector so that they are not visible in the page. Note that you use the tilde (~) placeholder in the `ImageUrl` attributes to specify that the images reside in a folder named `images` under the current application root.

The ComboBox User Control Interface

You need to consider what kind of interface you should expose from the user control to allow the hosting page to modify the behavior and appearance of the control—either in code or through attributes added to the control declaration. Let's say you settle on exposing the properties and the single method shown in Table 5.1.

TABLE 5.1
The Interface for the ComboBox User Control

Property or Method	Description
`IsDropDownCombo`	This property is a `Boolean` value, with a default of `False`. When it is `True`, a drop-down combo box is created. When it is `False`, a simple combo box is created.
`CssClass`	This property is a `String` value. It specifies the classname of the CSS style class to apply to the text box and list.
`DataSource`	This property is of type `Object`. It is a reference to a collection, `DataReader` instance, `DataTable` instance, or `HashTable` instance that contains the data to use with server-side data binding to fill the list.
`DataTextField`	This property is a `String` value. It is the name of the column or item in the data source that will be used to create the visible list of items.
`DataTextFormatString`	This property is a `String` value. It is a standard .NET-style format string that will be applied to the values in the `DataTextField` property when the list box is being filled.
`Items`	This property is a read-only `ListItemCollection` instance. It is a reference to the collection of `ListItem` objects that make up the list of values in the control.
`Rows`	This property is an `Integer` value with a default of 5. It specifies the number of rows to display in the list.
`SelectedIndex`	This property is an `Integer` value. It sets or returns the index of the item in the list that is currently selected. It returns `-1` if no item is selected.
`SelectedItem`	This property is a read-only `ListItem` instance. It returns a reference to the `ListItem` object that is currently selected, or `Nothing` (`null` in C#) if no item is selected.

TABLE 5.1
Continued

Property or Method	Description
`SelectedValue`	This property is a `String` value. It sets or returns the text in the text box and selects the matching value in the list, if present. It returns an empty string if no item is selected and the text box is empty.
`Width`	This property is an `Integer` value, with a default of `150`. It sets or returns the current width of the text box, in pixels.
`ShowMembers()`	This method returns a formatted string that contains a summary of the properties exposed by the control, ready to be inserted into an HTML page.

Many of these properties map directly to properties of the controls contained within the user control. However, others are more complex to implement because they affect more than one of the contained controls. Notice that you can expose an interface that allows server-side data binding to be used to populate the list. Using data binding is a common and popular way to fill list controls, and the combo box in this example really has to support it to be useful in many applications.

The other issue is that you want to hide the constituent controls so that users are not tempted to read or set values in those controls directly. Instead, users must access them by using the properties you expose and leave it to the code within the user control to figure out how to read or apply the values in the appropriate way.

The property names in this example are going to be familiar to users of the standard Web Forms controls, making for a much more intuitive user experience with the `ComboBox` control. However, one unconventional feature is the `ShowMembers` method, which simply generates a listing of the properties of the control. For users who are not developing in an environment that can display the properties of controls (such as Visual Studio or WebMatrix), this can be useful. Figure 5.9 shows the string that is returned from the method, as it is displayed in a Web page.

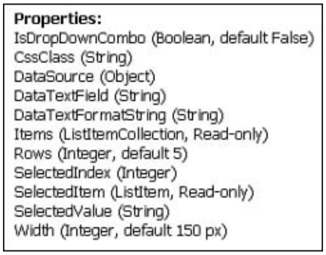

Properties:
IsDropDownCombo (Boolean, default False)
CssClass (String)
DataSource (Object)
DataTextField (String)
DataTextFormatString (String)
Items (ListItemCollection, Read-only)
Rows (Integer, default 5)
SelectedIndex (Integer)
SelectedItem (ListItem, Read-only)
SelectedValue (String)
Width (Integer, default 150 px)

FIGURE 5.9 The output of the `ShowMembers` method of the `ComboBox` control, as displayed in the sample Web page.

Exposing Style Properties

For this example, you do not expose much in the way of style properties. The `CssClass` property (as exposed by most ASP.NET Web Forms controls) can be used to change the appearance of the text box and list. One common technique is to expose the contained controls from a user control, allowing the hosting page to set all the standard properties of these controls, including all the style properties, such as `BackColor`, `BorderWidth`, and `Font`. However, that is not appropriate for this control.

You also need to maintain strict control over at least the width style selector that is applied to the text box and list, in order to maintain the position you want for these elements in the control. By applying the value of the CssClass property to the text box and list first and then setting the width selector afterward, you can override any setting that might upset the layout. You could extend this approach to other style selectors as well. Another approach would be to expose just, say, the Font property of the text box and list. However, any settings that the user requires can be made by applying the relevant value to the CssClass property of the control.

The Structure and Implementation of the ComboBox User Control

To give you a feel for the structure and implementation of the ComboBox user control, Listing 5.10 and, later in this chapter, Listing 5.11 show an outline of the content with the actual code removed for clarity. The following sections of this chapter fill in the gaps.

LISTING 5.10 An Outline of the Server-Side Script Section of the ComboBox User Control

```
<%@Control Language="VB" %>

<script runat="server">
<%-------------- Private Internal Variables ------------------%>
Private _width As Integer = 150
Private _rows As Integer = 5

<%------------------ Public Method --------------------------%>
Public Function ShowMembers() As String
...
End Function

<%-------------- Public Property Variables ------------------%>
Public IsDropDownCombo As Boolean = False
Public CssClass As String
Public DataSource As Object
Public DataTextField As String
Public DataTextFormatString As String

<%------------ Property Accessor Declarations ---------------%>
Public Property Width As Integer
...
End Property

Public Property Rows As Integer
...
End Property

Public Property SelectedValue As String
...
```

LISTING 5.10 Continued

```
End Property

Public ReadOnly Property Items As ListItemCollection
...
End Property

Public ReadOnly Property SelectedItem As ListItem
...
End Property

Public Property SelectedIndex As Integer
...
End Property

<%---------------------------------------------------------%>
...
... code to set the properties of the constituent controls
... and create the remaining output that is required
...
</script>
```

> **Setting Property Values Through Attributes**
>
> Note that when you're setting a property value through attributes, regardless of the data type, the value must be enclosed in single or double quotes. This is exactly the same way the standard .NET server controls work. The value is converted to the correct data type automatically when the control is compiled and instantiated by ASP.NET.

Following the opening `Control` directive, which tells ASP.NET that this is a user control, is the server-side script section. It declares two `Private` variables that you use within the control to maintain values for the properties that are exposed. This is followed by the `Public` declaration of the `ShowMembers` method and the declaration of five `Public` variables. You set default values for some of the `Private` and `Public` variables, and these will be used if the page does not provide specific values at runtime.

Exposing Properties As Public Variables

One of the easiest ways to expose properties from a user control is through `Public` variables. The values are, of course, accessible from within the user control because they are just ordinary variables. However, the user of the control can read or set these directly, by referencing them through the `id` property of the user control when declared in the hosting page. For example, if the `ComboBox` control is declared as follows:

```
<ahh:ComboBox id="MyCombo" runat="server" />
```

the user can set the `IsDropDownCombo` property with this:

```
MyCombo.IsDropDownCombo = True
```

The user can also set the property declaratively in the usual .NET Web Forms control way:

```
<ahh:ComboBox id="MyCombo" IsDropDownCombo="True" runat="server" />
```

Exposing Properties by Using Accessor Routines

The six remaining properties of the `ComboBox` user control are declared using `Public` accessor routines. An *accessor routine* allows the declaration of a property as read-only, write-only, or read/write, and it allows you to execute code when the value is set or read (whether it is set in code in the hosting page or through an attribute in the declaration of the control). You'll see these property accessor routines soon, after you look at the remainder of the user control structure.

Outputting the Appropriate HTML

Listing 5.11 shows the remaining content of the user control. You know that you will have to generate two different chunks of HTML, depending on whether you are creating a simple combo box or the drop-down variety. To do this, you declare both versions, enclosing each one in an ASP.NET `PlaceHolder` control, with its `Visible` property set to `False`. At runtime, all you need to do is change the `Visible` property to `True` for the relevant `PlaceHolder` control, and the correct section of HTML will be output.

Listing 5.11 The Visible User Interface Section of the ComboBox User Control

```
<%---------------- List-style Combo Box --------------------%>
<asp:PlaceHolder id="pchStandard" visible="false" runat="server">
<table border="0" cellpadding="0" cellspacing="0">
<tr><td align="right">
<asp:TextBox id="textbox" runat="server" /><br />
<asp:ListBox id="listbox" runat="server" />
</td></tr>
</table>
</asp:PlaceHolder>

<%---------------- Drop-down Combo Box ---------------------%>
<asp:PlaceHolder id="pchDropDown" visible="false" runat="server">
<div id="dropdiv" Style="position:relative" HorizontalAlign="Right"
     runat="server">
<asp:TextBox Style="vertical-align:middle" id="textbox2" runat="server"
/><asp:ImageButton id="dropbtn" BorderWidth="0" Width="16" Height="20"
                   Style="vertical-align:middle"
                   ImageUrl="~/images/click-down.gif" runat="server" /><br />
<asp:ListBox Style="display:none;position:absolute;left:20;top:25"
             id="dropbox" runat="server" />
<asp:Image id="imageup" ImageUrl="~/images/click-up.gif"
           Style="display:none" runat="server" />
<asp:Image id="imagedown" ImageUrl="~/images/click-down.gif"
```

LISTING 5.11 Continued

```
           Style="display:none" runat="server" />
</div>
</asp: PlaceHolder>
```

The ShowMembers Method

The declaration of the `ShowMembers` method is almost trivial. In the `Public` function that implements the method, you simply construct a string that contains the required formatted HTML, and you return it as the value of the function (see Listing 5.12).

LISTING 5.12 The Implementation of the `ShowMembers` Method

```
Public Function ShowMembers() As String
  Dim sResult As String = "<b>Combo Box User Control</b>" _
    & "</p><b>Properties:</b><br />" _
    & "IsDropDownCombo (Boolean, default False)<br />" _
    & "CssClass (String)<br />" _
    & "DataSource (Object)<br />" _
    & "DataTextField (String)<br />" _
    & "DataTextFormatString (String)<br />" _
    & "Items (ListItemCollection, Read-only)<br />" _
    & "Rows (Integer, default 5)<br />" _
    & "SelectedIndex (Integer)<br />" _
    & "SelectedItem (ListItem, Read-only)<br />" _
    & "SelectedValue (String)<br />" _
    & "Width (Integer, default 150 px)"
  Return sResult
End Function
```

Code in the hosting page can then display this string to the user. In the sample page, you simply use it to set the `Text` property of an ASP.NET `Label` control declared within the page:

```
lblResult.Text = MyCombo.ShowMembers()
```

Public Property Accessor Declarations

We mentioned the use of property accessor routines earlier in this chapter. This section looks at the implementation in the `ComboBox` user control. The simplest type of property accessor is shown in Listing 5.13. This property accessor exposes a read/write property that returns the value of an internal variable or sets the value of the internal variable to the value provided by code or in the control declaration within the hosting page. The value assigned to the property from the hosting page must be able to be cast (converted) into the correct data type, as defined in the property accessor declaration, or an exception will be raised.

LISTING 5.13 A Simple Property Accessor Routine

```
Public Property property-name As data-type
  Get
    Return internal-variable
  End Get
  Set
    internal-variable = value
  End Set
End Property
```

In the example in Listing 5.13, the new value for the internal variable is obtained using the keyword `value`, which is automatically set to the value assigned to the property. An alternative approach is to specify the name of the variable that will receive the new value when the property is set, as shown in Listing 5.14.

LISTING 5.14 A Property Accessor Routine That Specifies the Variable `TheNewValue`

```
Public Property property-name(TheNewValue) As data-type
  Get
    Return internal-variable
  End Get
  Set
  _ internal-variable = TheNewValue
  End Set
End Property
```

Read-Only and Write-Only Property Accessors

If you need to implement properties as read-only or write-only, you omit the `Get` or `Set` section, as appropriate. However, in Visual Basic .NET you must also add the `ReadOnly` or `WriteOnly` keyword to the property declaration, as shown in Listing 5.15.

LISTING 5.15 Specifying Read-Only and Write-Only Property Accessors

```
Public ReadOnly Property property-name As data-type
  Get
    Return internal-variable
  End Get
End Property

Public WriteOnly Property property-name As data-type
  Set
    internal-variable = value
  End Set
End Property
```

Property Accessors in C#

Listing 5.16 shows how you declare a property accessor in C#. Other than the use of curly braces, the overall approach is identical to that in Visual Basic .NET, with one exception: You don't use the ReadOnly and WriteOnly keywords in C# for read-only and write-only properties.

LISTING 5.16 Specifying Property Accessors in C#

```
public data-type property-name {
  get {
    return internal-variable;
  }
  set {
    internal-variable = value;
  }
}
```

The Property Accessors for the ComboBox User Control

The ComboBox control in this example has a property named Width that you expose via an accessor routine rather than as a Public variable. This is because you want to be able to execute some code when the property is set, which isn't possible if you just expose a variable from within the user control. When the user sets the Width property, you want to accept the value only if it is greater than 20 (it represents the width of the control, in pixels). So in the Get section of the accessor, you copy the value to the internal variable named _width only if it's greater than 20 (see Listing 5.17).

LISTING 5.17 The Property Accessor for the Width Property

```
Public Property Width As Integer
  Get
    Return _width
  End Get
  Set
    If value > 20 Then
      _width = value
      SetWidth()
    End If
  End Set
End Property
```

If the value is accepted, you then have to make sure all the constituent controls that use the value are correctly updated. Listing 5.18 shows how you use the value of the internal variable _width to set the width CSS style selector for the containing <div> element, the text box, and the list. The code in Listing 5.18 checks the value of the IsDropDown property first and then sets the values for the appropriate controls; however, you could just set them all, even though some controls will not actually be output to the client.

LISTING 5.18 The SetWidth Routine That Applies the Width Property

```
Private Sub SetWidth()
  If IsDropDownCombo = True Then
    dropdiv.Style("width") = _width.ToString()
    textbox2.Style("width") = (_width - 17).ToString()
    dropbox.Style("width") = (_width - 20).ToString()
  Else
    textbox.Style("width") = _width.ToString()
    listbox.Style("width") = (_width - 20).ToString()
  End If
End Sub
```

The same principles apply to the Rows property as to the Width property. Rows specifies the number of items that will be visible in the fixed or drop-down list of the ComboBox control. The accessor for this property accepts only values greater than zero, and it then applies the specified value to the appropriate list control (see Listing 5.19). Again, you only set the value of the appropriate control, but you could set both, even though only one will be output to the client.

LISTING 5.19 The Rows Property Accessor and SetRows Routine

```
Public Property Rows As Integer
  Get
    Return _rows
  End Get
  Set
    If value > 0 Then
      _rows = value
      SetRows()
    End If
  End Set
End Property
...
Private Sub SetRows()
  If IsDropDownCombo = True Then
    dropbox.Rows = _rows
  Else
    listbox.Rows = _rows
  End If
End Sub
```

The Items property of the ComboBox control exposes the items in the fixed or drop-down list section of the ComboBox control as a ListItemCollection instance, just like all the other standard Web Forms list controls (ListBox, DropDownList, RadioButtonList, and so on). And, like the standard controls, this property is read-only. You just need to return a reference to the Items property of the appropriate ListBox control within the user control, as shown in Listing 5.20.

LISTING 5.20 The Read-Only Items Property Accessor

```
Public ReadOnly Property Items As ListItemCollection
  Get
    If IsDropDownCombo Then
      Return dropbox.Items
    Else
      Return listbox.Items
    End If
  End Get
End Property
```

The SelectedItem, SelectedIndex, and SelectedValue Properties

The three remaining properties exposed by the ComboBox control—SelectedItem, SelectedIndex, and SelectedValue—provide information about the item that is currently selected. Again, following the model of the standard list controls, you expose a read-only property named SelectedItem that returns a ListItem instance representing the first selected item within the Items collection and a read/write property named SelectedIndex that sets or returns the index of the first selected item. You also provide a read/write property named SelectedValue. This property was added to the ASP.NET Web Forms ListControl base class (from which all the list controls are descended) in version 1.1 of the .NET Framework.

Listing 5.21 shows the implementation of the SelectedItem property. This isn't quite as straightforward as the properties examined so far. If the user has selected an item in the list, it will also be in the text box. However, the user may have typed into the text box a value that is not in the list (and so no item will be selected in the list). So the value in the text box really represents the selected value of the control.

Therefore, depending on which mode the control is in and whether there is a value selected in the appropriate list, you create a new ListItem instance or return a reference to an existing one. Notice that when you are creating a new ListItem control instance for the text box, you set both the Text and Value properties to the value of the text box.

LISTING 5.21 The SelectedItem Property Accessor Routine

```
Public ReadOnly Property SelectedItem As ListItem
  Get
    If IsDropDownCombo Then
      If dropbox.SelectedIndex < 0 Then
        Return New ListItem(textbox2.Text, textbox2.Text)
      Else
```

LISTING 5.21 Continued

```
        Return dropbox.SelectedItem
      End If
    Else
      If listbox.SelectedIndex < 0 Then
        Return New ListItem(textbox.Text, textbox.Text)
      Else
        Return listbox.SelectedItem
      End If
    End If
  End Get
End Property
```

The `SelectedIndex` property is a little more complex than the other properties. It's a read/write property; however, the `Get` section is simple enough—you just return the value of the `SelectedIndex` property for the appropriate list (see Listing 5.22). The complexity in the `Set` section comes from the fact that you first have to ensure that the new value is within the bounds of the list. It can be `-1` to deselect any existing selected value, or it can be between zero and one less than the length of the `ListItemCollection` instance. If the new value is valid, you can set the `SelectedIndex` property of the list control and then copy that value into the text box as well (as would happen if the user selected that value in the browser).

LISTING 5.22 The `SelectedIndex` Property Accessor Routine

```
Public Property SelectedIndex As Integer
  Get
    If IsDropDownCombo Then
      Return dropbox.SelectedIndex
    Else
      Return listbox.SelectedIndex
    End If
  End Get
  Set
    If IsDropDownCombo Then
      If (value >= -1) And (value < dropbox.Items.Count) Then
        dropbox.SelectedIndex = value
        textbox2.Text = dropbox.Items(SelectedIndex).Text
      End If
    Else
      If (value >= -1) And (value < listbox.Items.Count) Then
        listbox.SelectedIndex = value
        textbox.Text = listbox.Items(SelectedIndex).Text
      End If
    End If
  End Set
End Property
```

Finally, the most complex of all the property accessors is `SelectedValue`. As shown in Listing 5.23, you can get the selected value from the appropriate list within the user control easily enough (depending on the mode the `ComboBox` control is in). However, setting the `SelectedValue` property involves first copying the new value to the text box and then searching through the list to see if it contains an entry with this value. If it does, you must select this item as well (or, if the value appears more than once in the list, you must select the first instance). Moreover, you have to do all this with the appropriate text box and list control, depending on the mode that the `ComboBox` control is currently in.

LISTING 5.23 The `SelectedValue` Property Accessor Routine

```
Public Property SelectedValue As String
  Get
    If IsDropDownCombo Then
      Return textbox2.Text
    Else
      Return textbox.Text
    End If
  End Get
  Set
    If IsDropDownCombo Then
      textbox2.Text = value
      dropbox.SelectedIndex = -1
      For Each oItem As ListItem In dropbox.Items
        If value.Length <= oItem.Text.Length Then
          If String.Compare(oItem.Text.Substring(0, value.Length), _
                            value, True) = 0 Then
            oItem.Selected = True
            Exit For
          End If
        End If
      Next
    Else
      textbox.Text = value
      listbox.SelectedIndex = -1
      For Each oItem As ListItem In listbox.Items
        If value.Length <= oItem.Text.Length Then
          If String.Compare(oItem.Text.Substring(0, value.Length), _
                            value, True) = 0 Then
            oItem.Selected = True
            Exit For
          End If
        End If
      Next
    End If
```

LISTING 5.23 Continued

```
  End Set
End Property
```

The Page_Load Event Handler for the ComboBox Control

Now that you've looked in some detail at how to expose properties from a user control, the next stage is to see what happens when the control is instantiated in a hosting page. Although many events occur during the process of loading and executing an ASP.NET page and any user controls it contains, at this point you're most interested in the Page_Load event.

You need to accomplish the following tasks during the Page_Load event of the control. They don't have to be performed in this specific order, though this is the ordering used in the example code:

- Output the client-side script functions that are required to make the control work interactively.
- Set the CSS selectors and CSS class for the constituent controls.
- Attach the client-side event handlers to the constituent controls.
- Set the server-side data-binding properties and bind the list.
- Make sure that the width of the constituent controls and the number of rows in the list control are correctly set to override any conflicting CSS style property settings made in the hosting page.

Factoring the Code in the Property Accessors

You could, of course, create routines that remove some of the repeated code shown in Listing 5.23, but the intention here is to illustrate how setting a property of a *composite control* (that is, a control that contains other controls) can actually involve often quite complex internal processing.

The Ordering of Load and Init Events for a User Control

The Page_Load event for a user control occurs immediately after the Page_Load event for the hosting page. However, this is not the case for all events. The other useful event, Page_Init, occurs for all instances of a user control immediately before the Page_Init event of the hosting page.

Generating the Client-Side Script Section

Listing 5.24 shows the client-side script section that you must create and send to the client to enable the control to operate interactively. The selectList function runs when the user makes a selection in the list. It copies the selected value into the text box and, if the current mode is a drop-down combo box, it closes the list by calling the openList function that is shown at the end of Listing 5.24.

LISTING 5.24 The Client-Side Script Required for the Control

```
<script language='javascript'>
function selectList(sCtrlID, sListID, sTextID) {
  var list = document.getElementById(sCtrlID + sListID);
  var text = document.getElementById(sCtrlID + sTextID);
  text.value = list.options[list.selectedIndex].text;
  if (sListID == 'dropbox') openList(sCtrlID);
}

function scrollList(sCtrlID, sListID, sTextID) {
  var list = document.getElementById(sCtrlID + sListID);
  var text = document.getElementById(sCtrlID + sTextID);
  var search = new String(text.value).toLowerCase();
  list.selectedIndex = -1;
  var items = list.options;
  var option = new String();
  for (i = 0; i < items.length; i++) {
    option = items[i].text.toLowerCase();
    if (option.substring(0, search.length) == search ) {
      list.selectedIndex = i;
      break;
    }
  }
}

function openList(sCtrlID) {
  var list = document.getElementById(sCtrlID + 'dropbox');
  var btnimg = document.getElementById(sCtrlID + 'dropbtn');
  if(list.style.display == 'none') {
    list.style.display = 'block';
    btnimg.src = document.getElementById(sCtrlID + 'imageup').src;
  }
  else {
    list.style.display = 'none';
    btnimg.src = document.getElementById(sCtrlID + 'imagedown').src;
  }
  return false;
}
</script>
```

The `scrollList` function runs after the user presses and releases any key while the text box has the focus. It just has to search the list for the first matching value and select it. Notice that it ignores the letter case of the values by converting both values to lowercase before checking for a match.

The openList function runs when the user clicks the image button at the end of the text box, when the current mode is a drop-down combo box (this control is not generated for a simple combo box). It is also called, as you saw earlier, from the selectList function. The code in the openList function shows or hides the list control by switching the CSS display selector value between "block" and "none", depending on the current value, and it also swaps the src attribute of the image button to show the appropriate up or down button image.

More on Using Client-Side Script Code

Chapter 7, "Design Issues for User Controls," looks in more detail at the techniques used in this client-side code. Chapter 7 talks about client-side scripting in general and how you can integrate it with ASP.NET and your own custom controls. It also discusses browser compatibility issues. In subsequent chapters you'll see how you can build controls that adapt their behavior to different browsers.

Registering Client Script Blocks

The traditional way to generate client-side script sections in a Web page when using ASP is to simply write the code directly within the source of the page. This works fine in ASP.NET, too, because the <script> element does not contain the runat="server" attribute, so ASP.NET ignores it and sends it to the client as literal output.

You'll be generating the script section from within a user control, and user controls are intended to allow multiple copies to be placed in the same hosting page. In this case, you'd end up with multiple copies of the script section as well. To prevent this, you use the features of ASP.NET that are designed to inject items such as client-side script into the output generated by the page.

What About the runat="client" Attribute?

Interestingly, the W3C specifications suggest that you use <script runat="client">, although "client" is the default value for this attribute in the browser if the value is omitted. Unfortunately, ASP.NET doesn't allow you to include this attribute, and if you try to use it, you get the error "The Runat attribute must have the value Server." This is a shame because "client" would make it more obvious what the script section was intended for.

You first create the entire script section in a String variable, and then you register that script block with the hosting page by using the RegisterClientScriptBlock method. The string is injected into the page immediately after the opening server-side <form> tag (and after any hidden controls that ASP.NET requires, such as the one that stores the viewstate). The page also keeps track of registrations based on a string value you provide for the key. The hosting page is referenced through the Page property of the user control:

```
Page.RegisterClientScriptBlock("identifier", script-string)
```

Then, to ensure that you only ever insert one copy of the script, you can use the IsClientScriptBlockRegistered method to check whether a script section with the same identifier has already been registered. You register and insert the script section only if it hasn't been injected:

```
If Not Page.IsClientScriptBlockRegistered("identifier") Then
  Page.RegisterClientScriptBlock("identifier", script-string)
End If
```

The Parameters for the Client-Side Functions

If you are using multiple copies of the same user control in a page, you have to make sure that the client-side script can identify which instance it should be processing. One easy way around this is to use the JavaScript keyword `this`, which returns a reference to the current object or control.

However, the user control in this example contains constituent controls, and these vary depending on the mode of the control. So you have to pass in several values to allow the code to process the correct constituent controls. You can see in the earlier listings that the two main client-side script functions take three parameters: the `id` property of the current user control and the IDs of the `ListBox` and `TextBox` controls within the current user control:

```
function scrollList(sCtrlID, sListID, sTextID) { ...
```

When a user control is inserted into a hosting page, it is usually allocated an `id` value within the declaration:

```
<ahh:ComboBox id="cboTest1" runat="server" />
```

If the user does not specify an `id` value, ASP.NET adds an autogenerated one, such as `_ctl5`. Either way, you can retrieve this `id` value from within the user control through the `UniqueID` property that is exposed by all controls (inherited from `System.Web.UI.Control`). Although the autogenerated value is often the same as the `id` property, it may not be if the control is used within the template of another control—for example, in a data-bound `Repeater` or `DataList` control.

The constituent controls within a user control also have their `id` values massaged by ASP.NET. This is required; otherwise, multiple copies of a user control inserted into a hosting page would generate the same `id` values for their constituent controls. ASP.NET automatically prefixes the constituent controls with the ID of the user control itself plus an underscore. So, for example, the control with the ID value `"textbox2"` would appear in the control hierarchy of the hosting page with the `id` value `"cboTest1_textbox2"`.

> **Discovering the `id` Values of the Controls**
>
> **You can view the source of the page in the browser (by selecting View, Source in Internet Explorer) to see the `id` values that are generated. This is also a good way to debug your pages and find errors, as you get to see what output the user control is actually sending to the client.**

Therefore, in the user control in this example, you can create the ID prefix that will be added to the ID of the constituent controls by referencing the `UniqueID` property of the user control (the current object, as obtained using the keyword `Me` in Visual Basic .NET or `this` in C#):

```
Dim sCID As String = Me.UniqueID & "_"
```

The Code in the `Page_Load` Event Handler

In the `Page_Load` event, you can now generate the identifier for the current control and build the client-side script as a string. You must remember to include a carriage return at the end of each line of the script and use single quotes in the code itself so that each line of code can be wrapped in double quotes. In Visual Basic .NET, you can use the built-in `vbCrlf` constant to output a carriage return. In C#, you just have to include \n at the end of each line and also remember to replace any forward slashes in the code with \\.

An abbreviated section of the code to create the script section is shown in Listing 5.25 (the complete code, as seen in the browser, is shown in Listing 5.24, so there is no point in repeating it all here). You register this script to inject a copy if it doesn't already exist.

LISTING 5.25 The First Part of the Code in the Page_Load Event Handler

```
Sub Page_Load()

  Dim sCID As String = Me.UniqueID & "_"

  Dim sScript As String = vbCrlf _
  & "<script language='javascript'>" & vbCrlf _
  & "function selectList(sCtrlID, sListID, sTextID) {" & vbCrlf _
  & "  var list = document.getElementById(sCtrlID + sListID);" _
  & vbCrlf _
... etc ...
  & "}" & vbCrlf _
  & "<" & "/script>" & vbCrlf

  If Not Page.IsClientScriptBlockRegistered("AHHComboBox") Then
    Page.RegisterClientScriptBlock("AHHComboBox", sScript)
  End If
...
```

The next task in the Page_Load event handler is to set the properties and attributes of the constituent text box, list, and image button controls. Listing 5.26 shows this final section of code, continuing from Listing 5.25. Recall from earlier in this chapter that the two sets of HTML declarations for the two different modes that the ComboBox control can exhibit are enclosed in PlaceHolder controls that have their Visible property set to False. So depending on the mode you're currently in, you make the appropriate section of HTML visible by setting the Visible property of the PlaceHolder control that encloses it to True.

> **Hiding the Closing </script> Tag**
>
> You can hide the closing </script> tag from the compiler by splitting it into two sections in the source code. This is a throwback to a technique used when writing script dynamically into the page, which prevents the browser from raising an error. It isn't actually *required* here, but it does no harm.

LISTING 5.26 The Remaining Code for the Page_Load Event Handler

```
...
  If IsDropDownCombo = True Then
    pchDropDown.Visible = True
    If CssClass <> "" Then
```

LISTING 5.26 Continued

```
      dropbox.CssClass = CssClass.ToString()
      textbox2.CssClass = CssClass.ToString()
    End If
    dropbox.Attributes.Add("onclick", "selectList('" & sCID _
                           & "', 'dropbox', 'textbox2')")
    textbox2.Attributes.Add("onkeyup", "scrollList('" & sCID _
                           & "', 'dropbox', 'textbox2')")
    dropbtn.Attributes.Add("onclick", "return openList('" _
                           & sCID & "')")
    dropbox.DataSource = DataSource
    dropbox.DataTextField = DataTextField
    dropbox.DataTextFormatString = DataTextFormatString
    dropbox.DataBind()
  Else
    pchStandard.Visible = True
    If CssClass <> "" Then
      listbox.CssClass = CssClass
      textbox.CssClass = CssClass
    End If
    listbox.Attributes.Add("onclick", "selectList('" & sCID _
                           & "', 'listbox', 'textbox')")
    textbox.Attributes.Add("onkeyup", "scrollList('" & sCID _
                           & "', 'listbox', 'textbox')")
    listbox.DataSource = DataSource
    listbox.DataTextField = DataTextField
    listbox.DataTextFormatString = DataTextFormatString
    listbox.DataBind()
  End If

  SetWidth()
  SetRows()

End Sub
```

Now you can apply any CSS classname that may have been specified for the `CssClass` property to both the list and text box. Then you add the attributes to the text box and list that attach the client-side script functions. You use the complete ID of this instance of the user control (which you generated earlier) and specify the appropriate text box and list control IDs. If you're creating a drop-down combo box, you also have to connect the `openList` function to the list control.

Next, you set the data binding properties of the list within the user control to the values specified for the matching properties of the user control and call the `DataBind` method. In fact, you could check whether the `DataSource` property has been set first, before setting the properties and calling DataBind. This would probably be marginally more efficient, although the DataBind method does nothing if the DatSource property is empty.

Finally, you call the SetWidth and SetRows routines again to ensure that any conflicting CSS styles are removed from the constituent controls. And that's it; the ComboBox control is complete and ready to go. You'll use it in a couple simple sample pages next to demonstrate setting the properties and using data binding.

Using the ComboBox Control

The first example of using the ComboBox control contains three instances and applies three different styles to them so that you can see the possibilities (see Figure 5.10). You can find this page in the samples that you can download for this book (see www.daveandal.net/books/6744/), or you can just run it online on our server (also see www.daveandal.net/books/6744/). There is a [view source] link at the bottom of the page that you can use to see the source code and the source of the .ascx user control.

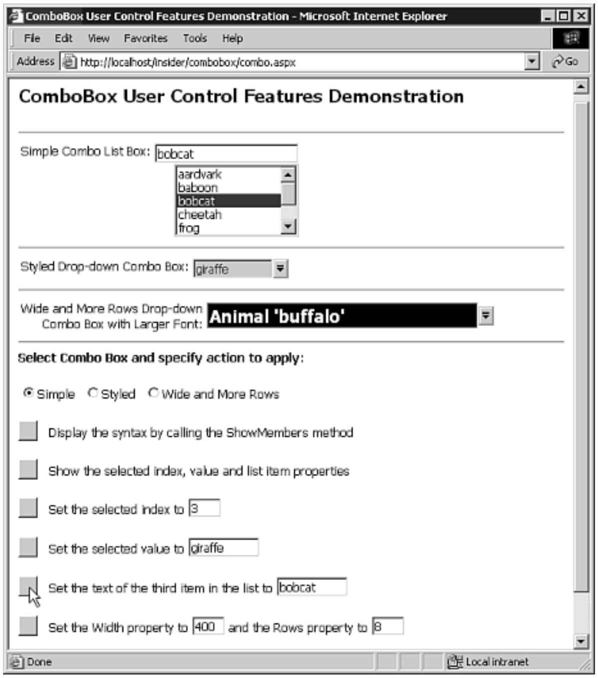


FIGURE 5.10
The ComboBox user control demonstration page in action.

The page contains a Register directive for the ComboBox control:

```
<%@Register TagPrefix="ahh" TagName="ComboBox" Src="ascx\combo.ascx" %>
```

As shown in Listing 5.27, three instances of the ComboBox control are then declared within the <form> section of the page. However, because the constituent controls reside within a <div>

element or a `<table>` element (depending on the mode specified), you have to use another `<table>` element to place a caption next to them. Listing 5.27 shows the attributes you specify for each one to apply the CSS style class (defined elsewhere in the page) and the other properties you set declaratively.

LISTING 5.27 The Declaration of the `ComboBox` Controls in the Sample Page

```
<form runat="server">

  <hr />
  <table border="0"><tr><td align="right" valign="top">
    Simple Combo List Box:</td><td>
    <ahh:ComboBox id="cboTest1" IsDropDownCombo="False" runat="server" />
  </td></tr></table>
  <hr />
  <table border="0"><tr><td align="right" valign="top">
    Styled Drop-down Combo Box:</td><td>
    <ahh:ComboBox id="cboTest2" CssClass="bluegray"
         IsDropDownCombo="True" runat="server" />
  </td></tr></table>
  <hr />
  <table border="0"><tr><td align="right" valign="top">
    Wide and More Rows Drop-down<br />Combo Box with Larger Font:</td><td>
    <ahh:ComboBox id="cboTest3" CssClass="reverse" Width="300"
         Rows="10" IsDropDownCombo="True" runat="server" />
  </td></tr></table>
  <hr />

  <b>Select Combo Box and specify action to apply</b>:<p />
  <asp:RadioButtonList id="optCbo" RepeatLayout="Flow"
       RepeatDirection="Horizontal" RepeatColumns="3" runat="server" >
    <asp:ListItem Value="cboTest1" Text="Simple  " />
    <asp:ListItem Value="cboTest2" Text="Styled  " />
    <asp:ListItem Value="cboTest3" Text="Wide and More Rows" />
  </asp:RadioButtonList>
  <p />
  <asp:Button Text="   " OnClick="ShowMembers" runat="server" />
  Display the syntax by calling the ShowMembers method
  <p />
  ...
  ... other controls here to set properties ...
  ...
  <asp:Label id="lblResult" EnableViewState="False" runat="server" />

</form>
```

The sample page also contains a RadioButtonList control that is used to specify which of the three ComboBox controls you want to apply the property settings to dynamically and a series of controls to specify the action to carry out on the selected ComboBox control. They are not all shown in Listing 5.27 to avoid unnecessary duplication. Notice that the Value properties of the items in the RadioButtonList control are the IDs of the three ComboBox controls.

Populating the ComboBox Controls from an ArrayList Instance

The Page_Load event handler is shown in Listing 5.28. If the current request is not a postback, you set the radio button to the first option and then create an ArrayList instance containing the values to be displayed in the ComboBox control list. By using data binding, you can apply this to all three of the ComboBox controls, just as you would any other list control—but with one exception. The user control in this example automatically calls the DataBind method when it loads (after the current Page_Load event has occurred for the hosting page), so you don't do it here. You can also take advantage of the DataTextFormatString property exposed by the ComboBox control to specify how the values are formatted in the third instance. This gives the effect you see in Figure 5.10 (for example, Animal 'buffalo').

LISTING 5.28 The Page_Load Event Handler in the Sample Page

```
Sub Page_Load()

  If Not Page.IsPostback Then

    ' executed when page is first loaded
    ' select first combobox in radiobutton list
    optCbo.SelectedIndex = 0

    ' create ArrayList to populate comboboxes
    Dim aVals As New ArrayList()
    aVals.Add("aardvark")
    aVals.Add("baboon")
    aVals.Add("buffalo")
    aVals.Add("cheetah")
    aVals.Add("frog")
    aVals.Add("giraffe")
    aVals.Add("lion")
    aVals.Add("lynx")

    ' assign to DataSource of comboboxes
    cboTest1.DataSource = aVals
    cboTest2.DataSource = aVals
    cboTest3.DataSource = aVals

    ' set display format string for third combobox
```

LISTING 5.28 Continued

```
    cboTest3.DataTextFormatString = "Animal '{0}'"

  End If
End Sub
```

Displaying the Members of the `ComboBox` User Control

When the user clicks the Show Members button, the routine named `ShowMembers` in the hosting page is executed. In it, you first have to get a reference to the `ComboBox` control currently selected in the `RadioButtonList` control. Then you call the `ShowMembers` method of this `ComboBox` control to get back a string, and you display that in a `Label` control in the page (see Listing 5.29). To see the result of this, refer to Figure 5.9.

LISTING 5.29 Calling the `ShowMembers` Method

```
Sub ShowMembers(oSender As Object, oArgs As EventArgs)

  ' get a reference to the selected comboxbox control
  Dim oCtrl As Object = Page.FindControl(optCbo.SelectedValue)

  ' call ShowMembers method of combobox control
  lblResult.Text = oCtrl.ShowMembers()

End Sub
```

Displaying Details of the Selected Item

The sample page contains a button that displays details of the item currently selected in the `ComboBox` control. Figure 5.11 shows the output that this generates in the page, and you can see the values for the `SelectedIndex`, `SelectedValue`, and `SelectedItem` properties, plus the items in the list, as obtained by iterating through the `Items` collection.

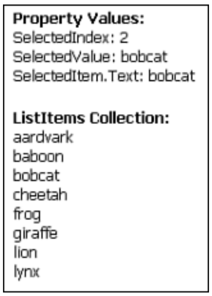


FIGURE 5.11 The output of the `ShowMembers` method of the `ComboBox` control, as displayed in a Web page.

Listing 5.30 shows the code that executes when this button in the hosting page is clicked. After the code gets a reference to the currently selected `ComboBox` control, a `StringBuilder` instance is used to create the string that is displayed in a `Label` control. Again, the process of extracting the values from the `ComboBox` control is exactly the same as you would use with any other Web Forms list control.

LISTING 5.30 The `ShowSelected` Routine That Calls the `ShowMembers` Method

```
Sub ShowSelected(oSender As Object, oArgs As EventArgs)

  ' get a reference to the selected comboxbox control
  Dim oCtrl As Object = Page.FindControl(optCbo.SelectedValue)

  ' use a StringBuilder to hold string for display
  Dim sResult As New StringBuilder("<b>Property Values:</b><br />")

  ' collect details of current selection from combobox
  sResult.Append("SelectedIndex: " & oCtrl.SelectedIndex & "<br />")
  sResult.Append("SelectedValue: " & oCtrl.SelectedValue & "<br />")
  sResult.Append("SelectedItem.Text: " & oCtrl.SelectedItem.Text & "<p />")

  ' collect all items in the combobox list
  sResult.Append("<b>ListItems Collection:</b><br />")
  For Each iItem as ListItem In oCtrl.Items
    sResult.Append(iItem.Text & "<br />")
  Next

  ' display results in the page
  lblResult.Text = sResult.ToString()

End Sub
```

Setting the Properties of the `ComboBox` User Control

The remaining buttons in the sample page set various properties of the selected `ComboBox` control, including `SelectedIndex`, `SelectedValue`, `Width`, and `Rows`. They validate the values first to make sure they are of the correct data types and within range. Then, after obtaining a reference to the current `ComboBox` control, as in the previous examples, they apply the property setting(s) to it. This chapter doesn't list all the code for these routines because it is extremely repetitive, but you can see it by using the [view source] link at the bottom of the page at `www.daveandal.net/books/6744/`.

Populating the `ComboBox` Control

The sample page described in this section (`populating.aspx`) demonstrates different ways of populating the `ComboBox` user control. As in the previous example, it registers the control with a `Register` directive and then declares three instances of it. This time, they are all of the default style. However, the lists are filled using three different techniques this time, as you can see in Figure 5.12.

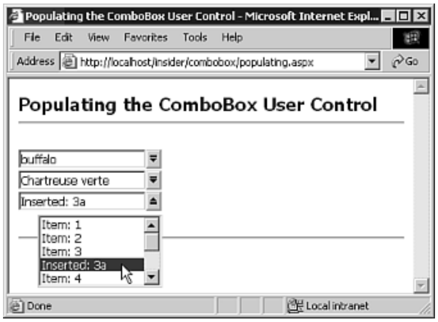


FIGURE 5.12 Filling the `ComboBox` control list, using different data sources and data binding.

The first list is filled using the same `ArrayList` instance as in the previous example. The second is filled from the Northwind sample database that is supplied with SQL Server, using the values from the `ProductName` column of the `Products` table. The third `ComboBox` control is filled by creating new `ListItem` instances in code and adding them to the `ComboBox` control's `Items` collection (the `ListItemCollection` instance exposed by the `Items` property). This section of code also demonstrates how you can access a specific item in the list and read or change its value.

> **Editing the Connection String**
>
> **You must edit the connection string in the `web.config` file to point to your database server, and you must specify the correct username and password if you run the examples on your own server.**

All this is done within the `Page_Load` event handler for the sample page, with the exception of a separate routine that creates a `DataReader` instance for the table in the Northwind database. Listing 5.31 shows the complete code for this sample page.

LISTING 5.31 Code That Demonstrates Techniques for Populating the `ComboBox` Control

```
Sub Page_Load()

  If Not Page.IsPostback Then

    ' populate combobox controls
```

LISTING 5.31 Continued

```
    ' databind ArrayList to first one
    Dim aVals As New ArrayList()
    aVals.Add("aardvark")
    aVals.Add("baboon")
    aVals.Add("buffalo")
    aVals.Add("cheetah")
    aVals.Add("frog")
    aVals.Add("giraffe")
    aVals.Add("lion")
    aVals.Add("lynx")
    cboTest1.DataSource = aVals

    ' databind second combobox to a DataReader
    cboTest2.DataSource = GetDataReader()
    cboTest2.DataTextField = "ProductName"

    ' insert values directly into list for third combobox
    Dim oList As ListItemCollection = cboTest3.Items
    For iLoop As Integer = 1 To 9
      oList.Add(New ListItem("Item: " & iLoop.ToString()))
    Next
    oList.Insert(3, New ListItem("Inserted: 3a"))

  End If

End Sub

Function GetDataReader() As OleDbDataReader

  ' get DataReader for rows from Northwind Products table
  Dim sConnect As String _
    = ConfigurationSettings.AppSettings("NorthwindOleDbConnectString")
  Dim sSelect As String _
    = "SELECT ProductName FROM Products WHERE ProductName LIKE 'c%'"
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSelect, oConnect)
    Return oCommand.ExecuteReader(CommandBehavior.CloseConnection)

  Catch oErr As Exception
```

LISTING 5.31 Continued

```
    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text = oErr.Message

  End Try

End Function
```

Summary

This chapter begins by looking at the techniques that are available in ASP.NET for creating reusable content and for reducing the amount of repetitive work you need to do when building Web sites and Web applications. While the server-side include approach is still valid, there are better ways. User controls and custom server controls both offer advantages, and they provide a natural approach that integrates well with ASP.NET in both style and effectiveness.

Other techniques briefly discussed in this chapter are using master pages and templates and wrapping existing COM/COM+ components for use in ASP.NET. Chapter 9, "Page Templates" looks in more depth at the first of these options, but this book does not pursue the COM/COM+ wrapper technique any further.

The second part of this chapter walks through the design and construction of a nontrivial user control that implements a feature that is missing from the standard range of browser-based control elements—a dual-mode combo box. You saw how it uses constituent controls, how to expose properties and methods, and how to use code within the control to manage its behavior and appearance.

Finally, to complete the chapter, you saw how you can use the new `ComboBox` control in ASP.NET pages. However, this chapter does not address a few issues. One is the way that the client-side script within the control works; we'll come back to this issue in Chapter 6. Another is the general compatibility of the control in different browsers. As you've seen in this chapter, the sample `ComboBox` control works fine in Internet Explorer 5.0, and it also works well in the latest versions of Opera. However, there are issues in other browsers, especially older ones. In subsequent chapters, you'll see how you must understand the issues, how you can address them, and how you can build controls that adapt to suit a wider range of browser types.

6

Client-Side Script Integration

IN THIS CHAPTER

ASP.NET provides plenty of clever server-side controls that ultimately generate HTML elements in the browser. However, with the notable exception of the validation controls and one or two other features, that's really all they do. In fact, when they start to build Web applications, most developers who are used to building Windows applications find that the interface features Web developers have become accustomed to using are quite poor.

We can't do much about the actual client-side HTML elements and controls that are available because that's the whole nature of the Web. Content is supposed to be universally supported in all browsers, and browsers are supposed to follow accepted standards. Therefore, if your application needs some fancy new kind of multistate psychedelic flashing button, you're going to have to find a way to build it yourself. And depending on how you implement it, you might then have to find a way to persuade all the people who use your site to download this great new control and install it on their machine.

> **Avoiding Meaningless and Annoying Content**
> In reality, most people have seen enough in the way of annoying Java applets, malicious ActiveX controls, time-wasting Flash animations, and pointless Shockwave effects. They expect an application to do *what it says on the box* by being intuitive and easy to understand and working seamlessly and as fast as possible, given the nature of Internet connections.

Client-side scripting has been a feature of Web development for almost as long as the Web in its current incarnation has been around. Scripting provides an increasing number of useful features that you can take advantage of to make Web applications appear more seamless, responsive, and interactive, while still running well in almost all popular browsers in use today.

This book is about ASP.NET and not client-side scripting, but, in fact, the two are no longer really divisible. ASP.NET generates client-side script in varying quantities, depending on the server controls you place on a page. Even simple effects such as auto-postback depend on some client-side script.

And you saw client-side script being used in the `ComboBox` control you created in Chapter 5, "Creating Reusable Content."

This chapter takes a look at the major client-side script issues that affect you when you create ASP.NET pages, as well as when you create reusable content such as user controls and server controls. This is by no means a reference work on client-side scripting, but it reinforces some of the basic techniques and demonstrates useful ways that even very simple script can solve common issues you come up against when building ASP.NET Web applications.

Client-Side Interaction on the Web

Client-side interaction is hard to achieve because of the disconnected nature of HTTP and the way that browsers and Web servers work. Information is passed to and from the client only during distinct phases of the Web-surfing process. The server builds the page and sends it to the browser, and the browser submits the page back to the server when it's ready for another one.

Okay, so there are some well-known ways that you can get around this issue, usually by installing a component in the browser that can send and receive HTTP requests without having to reload the current page. The XMLHTTP component within the MSXML parser in Internet Explorer 5 and above is a good example. You can also use Macromedia Flash and a range of third-party plug-ins or components for other browsers. However, the point is that if you want a page to be interactive to the extent that it "does stuff" while loaded into the browser, you need to find a way to execute code within the confines of the browser.

When you're building items of reusable content, as demonstrated in Chapter 5, client-side scripting allows you to push the envelope beyond the simple flow layout of HTML controls to provide extra features that are often seen in traditional executable applications. The following sections explore the fundamental aspects of where, when, and how—and then move on to look at some useful techniques that integrate client-side and server-side programming and provide examples you can use in your own pages.

Client-Side Scripting in the Browser

Client-side scripting has been supported in the mainline Web browsers since Netscape Navigator 2 and Internet Explorer 3. These browsers, and many others, support the simple HTML Document Object Model (DOM) by exposing specific elements to script that runs within the browser. Such elements include frames, forms, controls (such as `<input>` and `<select>`), images, links, and anchors (`<a>` elements with `name="..."` rather than `href="..."`). Script can also access the fundamental objects such as the current window and the document within a frame or a window.

This level of accessibility to the page content allows the traditional effects such as reading and setting the values of controls, submitting a form, or swapping images in an `<img>` element. It also supports a small set of useful events, such as detecting when a control gets or loses the focus or receives a click (via keyboard or mouse). However, this basic level of support for scripting does not offer the three main features that you often need when building better controls or interactive content:

- Access to all the elements on the page, with the ability to read and set the content of each one, show or hide it, and generally manipulate it.
- Access to a full range of keypress events, so that you can manage how a control behaves, depending on user interaction via the keyboard.
- The ability to position elements outside the flow model, using fixed (absolute) coordinates that are relative to a container (such as the page or a parent element). It's nice to be able to do this dynamically and even be able to move elements around while the page is displayed.

CSS2 and Dynamic HTML

While much has been made of the "browser wars" over the past few years, the situation today regarding the use of client-side scripting is actually a lot more favorable than it was. Microsoft and Netscape added a feature set they called Dynamic HTML to their version 4 browsers, although the blatant incompatibility between them (and the resulting outcry from Web developers and standards bodies alike) was perhaps one of the key factors in the evolution of more comprehensive client-side standards over the following years.

Today we have Cascading Style Sheets (CSS) at version 2, HTML at version 4, and XHTML at version 1.0; together, they provide not only a comprehensive display model based on the original CSS recommendation but also a standard set of methods for accessing and manipulating document content from script or code running on the client. While these recommendations are fundamentally similar to the original Microsoft implementation in Internet Explorer 4, there are subtle differences. However, the mainline manufacturers all have "version 6" browsers available that generally do meet the basic CSS2, HTML4, and XHTML recommendations. These include the following:

- Internet Explorer 5.x and 6.x, although CSS2 support is generally more comprehensive and less buggy in version 6 than in earlier versions. And there are still some issues with the way that the box display model works.
- Mozilla 1.x (effectively a version 6 browser) and Netscape 6.x, which use the same rendering engine (depending on minor version number) and generally support the latest standards very well. Minor exceptions are occasional buggy rendering, particular with absolutely positioned elements.
- Opera 6.x and 7.x, which both have comprehensive support for the latest standards, although problems with dynamic positioning have occurred in version 6.0. Opera 4.0 and 5.0 also supported CSS2 to a large extent.

CSS2 Support in Version 6 Browsers

In reality, some of the more esoteric features of CSS2 are not fully supported in all version 6 browsers or are less than totally compatible across the different version 6 browsers. However, the basic techniques that we take advantage of in our examples do work in all the current version 6 browsers.

Selecting Your Target

Are most users out there using a version 6 browser? Admittedly, our own Web site is mainly aimed at developers working with the latest Microsoft technologies, so the results we see are probably not representative of the population, but around 75% of our visitors are using Internet Explorer 5 or higher, Netscape/Mozilla 6 or higher, and Opera 6 or higher. Looking at the stats available on other sites, the percentage of visitors using these newer browsers varies from something over 55% to almost 90%.

Why Use the Latest Browser?

You probably wouldn't want to risk driving on an icy freeway during rush hour in a 1910 Model T Ford. Four-inch-wide tires, vague steering, and a distinct lack of braking performance when compared to those in modern vehicles, would make this a risky undertaking at the best of times. Likewise, using an old and unsupported browser is an equally foolhardy adventure these days, with the proliferation of malicious scripts, annoying Java applets, and downright dangerous ActiveX controls that are out there on the Web and being delivered daily in junk email messages. Most car drivers appreciate the added safety of antilock brakes, airbags, and seatbelts, and the sensible browser user does the same by choosing the latest browser so that he or she can stay secure with the updates and patches provided for it.

It's probably reasonable to assume that you can take advantage of CSS2 and HTML4 features to add client-side interactivity to your pages, without affecting the majority of users. Of course, that doesn't mean you can ignore the rest because there are issues such as providing accessibility to users of text-only browsers, page readers, and other devices aimed at specialist markets or disabled users.

The language of choice for client-side programming is, of course, JavaScript—because only Internet Explorer can natively support VBScript. There are several versions of JavaScript available, but the "vanilla" version 1.x satisfies almost all requirements for the simple client-side interactivity you need when building most user controls and server controls. And because Internet Explorer actually has its own JScript/ECMAScript interpreter rather than a real JavaScript one, staying with the features in JavaScript 1.0 or 1.1 provides the best compatibility option.

Version 6 Browser-Compatible Code Techniques

Given the three tasks listed earlier in this chapter that you most commonly need to accomplish in client-side script—access to all elements, access to keypress information, and dynamic positioning of elements—the following sections look at how these can be achieved in modern browsers using script.

Accessing Elements Within a Page

Internet Explorer 4 was the first mainstream browser to provide full access to all the elements in a page by exposing them from the `document` object as a collection called `all`. It also allowed selection of a set of elements by type, via the use of the `getElementsByTagname` method. While CSS2 provides the same `getElementsByTagname` method, it replaces the `document.all` collection with two methods named `getElementById` and `getElementByName`. Because ASP.NET sets the `id` and `name` attributes of an element that is created by a server control to the same value (with the exception of the `<input type="radio">` element), the `getElementById` and `getElementByName` methods generally provide the same result.

Therefore, the technique for getting a reference to an element within client-side script depends on whether you are only going to send the page to a CSS2-compliant client or whether you want the code to adapt to different client types automatically. The accepted technique for providing adaptive script in a page is to test for specific features that identify the browser type or the support it provides for CSS2. These features are summarized in Table 6.1.

TABLE 6.1
Features You Can Use to Detect the Browser Type or Its Feature Support

Feature	Description
`document.all collection`	Supported by Internet Explorer 4.0 and above
`document.layers collection`	Supported by Netscape Navigator 4.x only
`getElementById method`	Supported by CSS2-compliant browsers

By using the features described in Table 6.1, you can write code such as that shown in Listing 6.1 to execute different sections of script, depending on which browser loads the page. Notice that this causes Internet Explorer 5.x to execute the CSS2-compliant code. If you find that this does not perform correctly with your specific client-side scripts, you can change the tests so as to place Internet Explorer versions 4.x and 5.x into the same section by checking the value of the `navigator.appName` and `navigator.appVersion` properties as well.

> **Using the ASP.NET `BrowserCapabilities` Object**
>
> **You can use the ASP.NET `BrowserCapabilities` object to sniff the browser type and deliver the appropriate page or include the appropriate script or controls. Chapter 7, "Design Issues for User Controls," and Chapter 8, "Building Adaptive Controls," demonstrate this approach.**

LISTING 6.1 Detecting the Client's Feature Support in Script Code

```
if (document.getElementById) {
   ... code for CSS2-compliant browsers here ...
}
else if (document.all) {
  ... code for IE 4.x here ...
}
else if (document.layers) {
  ... code for Netscape Navigator 4.x here ...
}
else {
  ... code for older browsers here ...
}
```

However, as discussed earlier, the number of users still running Navigator 4.x and Internet Explorer 4.x is extremely low, so you generally need to test only for CSS2 support and provide fallback for all other browsers. There's not a lot of point in spending long development times on supporting browsers that only 1% of users may still be running.

Accessing Keypress Information

Microsoft's early implementation of Dynamic HTML exposed three keypress events for all the interactive elements on a page and for the `document` object itself. These are the `keydown`, `keypress`, and `keyup` events, and they occur in that order. The `keypress` event exposes the ANSI code of the key that was pressed, and the other two events expose a value that identifies the key itself (as located within the internal keyboard mappings) rather than the actual character.

Listing 6.2 shows the generally accepted technique for detecting a keypress that works in Internet Explorer version 4.x and higher and in CSS2-enabled browsers. If the event is exposed by the `window` object, as in Internet Explorer 4 and above, it is extracted from the `keyCode` property of the `event` object. In CSS2-compliant browsers, the event is passed to the function by the control to which the function is attached as a parameter, and it can be extracted from the `which` property.

LISTING 6.2 Detecting a Keypress Event and the Code of the Key That Was Pressed

```
<element onkeypress="showKey(event);">
...
<script language="javascript">
<!--
  var iKeyCode = 0;
  if (window.event)
    iKeyCode = window.event.keyCode
  else
    if (e)
      iKeyCode = e.which;
```

LISTING 6.2 Continued

```
  window.status = iKeyCode.toString();
//-->
</script>
```

Dynamic and Absolute Element Positioning

The final feature set that you often need a browser to support when creating user controls and server controls is a way of positioning elements within and outside the usual flow of the page, changing that setting dynamically, and specifying the size of elements. Again, the original Microsoft Dynamic HTML approach has survived almost intact in CSS2, so these features are available in Internet Explorer 4.x and above, as well as in CSS2-compliant browsers. In more strict terms, the features that you are most likely to take advantage of are summarized in Table 6.2.

TABLE 6.2
Dynamic and Absolute Element Positioning Features

Feature	Description
Showing and hiding elements	Set the `display` selector of the `style` attribute to `block`, `inline`, or `hidden`. Other values can be used, but these three are most useful. The value `block` forces this element to start on a new line and following content to wrap to a new line. The value `inline` means that preceding and following content will be on the same line, unless that content forces a new line. The value `hidden` removes the element and all child elements from the page.
Absolute positioning	Set the `position` selector of the `style` attribute to `absolute` to fix an element using the top and left coordinates provided as the `top` and `left` style selectors. This removes the element from the flow layout of the page. The alternative is `position:relative`, which forces the element to follow the flow layout of the page but also allows it to act as a container within which child elements can be absolutely positioned. If no parent element contains `position:absolute` or `position:relative`, the current element is positioned with respect to the top left of the browser window.
Specifying the actual size of elements	Set the `width` and `height` selectors of the `style` attribute to fixed values. These values can be specified with units `px` (pixels), `pt` (points), `in` (inches), `cm` (centimeters), `mm` (millimeters), or `pc` (picas) or the typographical units `em`, `en`, and `ex`. The default is `px`.
Positioning and moving elements dynamically	The values for the `display`, `position`, `top`, `left`, `width`, and `height` selectors can be changed while the page is loaded, and the page will immediately reflect these changes by showing, hiding, or moving the element.

The Client-Side Code in the `ComboBox` User Control

To demonstrate the feature sets described so far in this chapter, let's briefly review some of the code from Chapter 5, "Creating Reusable Content." That chapter shows how easy it is to build a `ComboBox` user control for use in browsers that support CSS2 (see Figure 6.1).

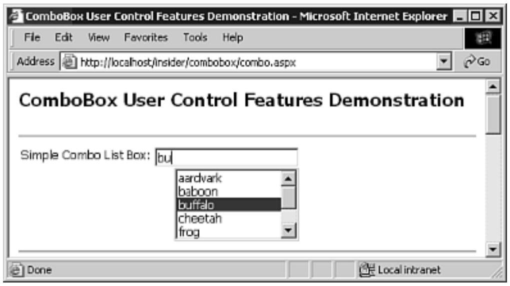


FIGURE 6.1 The customer `ComboBox` user control created in Chapter 5.

This control includes client-side code that manipulates the control elements and their values while the page is loaded into the browser, using most of the features just discussed. Listing 6.3 shows the complete client-side code section. In each of the three functions in Listing 6.3, you can see that you get a reference to the controls you want to manipulate by using the `getElementById` function that is exposed by the `document` object.

LISTING 6.3 The Client-Side Script for the `ComboBox` User Control

```
<script language='javascript'>
function selectList(sCtrlID, sListID, sTextID) {
  var list = document.getElementById(sCtrlID + sListID);
  var text = document.getElementById(sCtrlID + sTextID);
  text.value = list.options[list.selectedIndex].text;
  if (sListID == 'dropbox') openList(sCtrlID);
}

function scrollList(sCtrlID, sListID, sTextID) {
  var list = document.getElementById(sCtrlID + sListID);
  var text = document.getElementById(sCtrlID + sTextID);
  var search = new String(text.value).toLowerCase();
  list.selectedIndex = -1;
  var items = list.options;
  var option = new String();
  for (i = 0; i < items.length; i++) {
    option = items[i].text.toLowerCase();
    if (option.substring(0, search.length) == search ) {
      list.selectedIndex = i;
      break;
    }
  }
}

function openList(sCtrlID) {
  var list = document.getElementById(sCtrlID + 'dropbox');
  var btnimg = document.getElementById(sCtrlID + 'dropbtn');
  if(list.style.display == 'none') {
```

LISTING 6.3 Continued

```
    list.style.display = 'block';
    btnimg.src = document.getElementById(sCtrlID + 'imageup').src;
  }
  else {
    list.style.display = 'none';
    btnimg.src = document.getElementById(sCtrlID + 'imagedown').src;
  }
  return false;
}
</script>
```

Alternative Client Support Options

The code in Listing 6.3 doesn't provide support for non-CSS2 browsers. This is because the only ones that support another feature needed for this control (absolute positioning) are Internet Explorer 4.x and Netscape 4.x. Because the number of hits likely to be encountered from these two browsers is negligible, it doesn't seem worth supporting them.

However, extending support to Internet Explorer 4 isn't hard; you would just need to add the test for the `document.all` collection, as shown in Listing 6.4, and then access the elements by using this collection. The remaining code will work fine as it is.

> **Accessing the `document.all` Collection and the `getElementID` Method in JavaScript**
>
> **Remember that `document.all` is a collection (array) of elements, so in JavaScript, you must use square brackets (`[]`) to access the members. On the other hand, `getElementId` uses ordinary parentheses (`()`) because it's a method, and you are providing the element ID as a parameter.**

LISTING 6.4 Adapting the `selectList` Function to Work in Internet Explorer 4.x

```
function selectList(sCtrlID, sListID, sTextID) {
  var list;
  var text;
  if (document.all) {
    list = document.all[sCtrlID + sListID];
    text = document.all[sCtrlID + sTextID];
  }
  else {
    list = document.getElementById(sCtrlID + sListID);
    text = document.getElementById(sCtrlID + sTextID);
  }
  text.value = list.options[list.selectedIndex].text;
  if (sListID == 'dropbox') openList(sCtrlID);
}
```

Keypress Events in the ComboBox Control

The `scrollList` function shown in Listing 6.3 continually selects the first matching value in the list while the user is typing in the text box section of the `ComboBox`. To work, it must be called every time a key is pressed so that it can search the list for the appropriate value (if one exists). To achieve this, you handle the `onkeyup` event, which runs when the user releases a key.

You attach the `scrollList` function to the `input` element that implements the text box by using server-side code (as shown in Chapter 5). When the page gets to the client, the HTML declaration of the text box (with the nonrelevant style information omitted) looks like this:

```
<input name="cboTest2:textbox2" type="text" id="cboTest2_textbox2"
       onkeyup="scrollList('cboTest2_', 'dropbox', 'textbox2')" />
```

You can see that a `keyup` event will pass the three required parameters to the `scrollList` function. However, you aren't actually interested in detecting *which* key was pressed because the function just compares the values within the text box and the list to figure out which entry to select. This means that you don't have to pass the `event` object (required to detect which key was pressed in Netscape and Mozilla browsers) as a parameter. In later examples, you'll see occasions where you do need to detect the actual key value.

Element Positioning in the ComboBox Control

The version of the `ComboBox` control that provides a drop-down list uses absolute positioning to fix the width of the enclosing `<div>` element, the width of the text box within it, and the position and size of the `<select>` list that implements the drop-down list part of the control. You can see in Listing 6.5 that the top of the list is positioned 25 pixels below the top of the text box and 20 pixels to the left of the text box. The widths of the text box and list are adjusted accordingly, depending on the width of the enclosing `<div>` element. All these values are calculated on the server and are used to create the style selectors shown in Listing 6.5.

LISTING 6.5 The Style Selectors for Positioning the Text Box and List in the `ComboBox` User Control

```
<div id="cboTest1_dropdiv" Style="position:relative;width:150;">
 <input type="text" id="cboTest1_textbox2" ...
  style="vertical-align:middle;width:133;" />
 <input type="image" id="cboTest1_dropbtn" ... />
 <select size="5" id="cboTest1_dropbox" ...
  style="display:none;position:absolute;left:20;top:25;width:130;">
   <option value="aardvark">aardvark</option>
   ...
   <option value="lynx">lynx</option>
 </select>
 <img id="cboTest1_imageup" style="display:none" ... />
 <img id="cboTest1_imagedown" style="display:none" ... />
</div>
```

Notice that the list has the selector `display:none` so that it's not visible in the page when it loads. Likewise, the two `<img>` elements that hold the up and down button images are not visible either. They are simply there to preload the images so that they can be instantly switched when the user opens and closes the list.

Showing and Hiding the List Control

The code in the `openList` function shown in Listing 6.3 has the job of showing and hiding the drop-down list when the user clicks the up/down button or makes a selection from the list. It's simply a matter of switching the `display` selector for the list between `none` and `block`, depending on whether the list is already open or closed. At the same time, you switch the button image. The relevant code section is shown in Listing 6.6.

LISTING 6.6 Showing and Hiding the Drop-Down List Part of the `ComboBox` Control

```
if(list.style.display == 'none') {
  list.style.display = 'block';
  btnimg.src = document.getElementById(sCtrlID + 'imageup').src;
}
else {
  list.style.display = 'none';
  btnimg.src = document.getElementById(sCtrlID + 'imagedown').src;
}
```

Useful Client-Side Scripting Techniques

The following sections demonstrate some useful client-side scripting techniques. These techniques are some of the several that regularly crop up as questions on ASP.NET mailing lists and discussion forums:

- Trapping an event that occurs on the client and popping up a confirmation dialog before carrying out the action on the server (for example, getting the user to confirm that he or she wants to delete a row in a `DataGrid` control).
- Trapping a Return keypress to prevent a form from being submitted or trapping any other keypress that might not be suitable for a control or an application you are building.
- Handling individual keypress events (for example, implementing a `MaskedEdit` control).
- Creating a button that the user can click only once—effectively creating a form that can only be submitted once. This prevents the user from causing a second postback, which might interrupt server-side processing, when nothing seems to be happening at the client.

The following sections start by examining the ways you can inject client-side confirmation dialogs into ASP.NET code and then look at how to trap keypresses and prevent a form from being submitted.

Buttons, Grids, and Client-Side Script

A common scenario with the excellent ASP.NET grid and list controls is to allow users to edit and delete rows inline—while they are displayed within a `DataGrid` or `DataList` control. The `DataGrid` control can provide attractive and interactive pages, with minimum code requirement from the developer. However, one feature that many people ask for is to be able to prompt users before carrying out some action such as deleting a row.

One way would be to trap the delete event on the server and generate a confirmation page to send back to the user. However, this is counterintuitive, inefficient, and breaks the flow of the application. The user will probably expect something like what is shown in Figure 6.2.

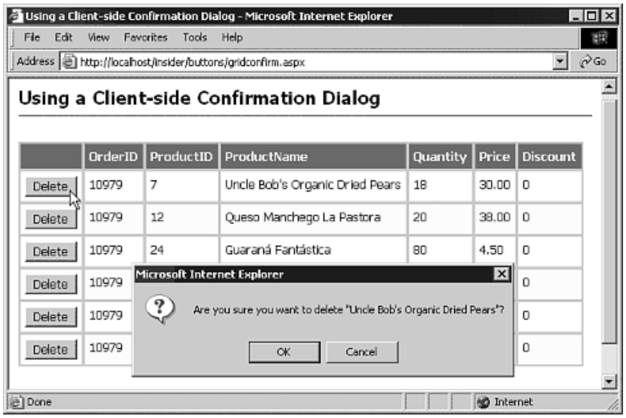


FIGURE 6.2 Confirming a button click before allowing a row to be deleted.

In fact, this kind of feature is extremely easy to add to the `DataGrid` control and other controls. All you need is a simple client-side script function that pops up a JavaScript `confirm` dialog and returns `true` or `false`, depending on which button the user clicked. You then return that value to the control that raised the event—in this example, the Delete button. If you return `true`, the event is processed and the row is deleted. If you return `false`, the event is canceled and the row is not deleted.

Listing 6.7 shows the client-side function, named `ConfirmDelete`, that is used in this example. You can see that this function is extremely simple, taking just the product name as a parameter. It displays the confirmation message in a `confirm` dialog and returns the value from the dialog (which will, of course, be `true` if the user clicked OK or `false` if the user clicked Cancel). You declare this function in the `<head>` section of the page, although you could inject it into the page by using the `RegisterClientScriptBlock` method (as described in Chapter 5) if you prefer.

LISTING 6.7 The Client-Side `ConfirmDelete` Function

```
<script language='javascript'>
<!--
function ConfirmDelete(sName) {
  var sMsg = 'Are you sure you want to delete "' + sName + '"?';
  return (confirm(sMsg));}
```

LISTING 6.7 Continued

```
//-->
</script>
```

The Declaration of the `DataGrid` Control

The visible part of the sample page is made up of the `<form>` section shown in Listing 6.8, which contains the declaration of the `DataGrid` control. A lot of this code sets the appearance of the `DataGrid`, but the important point to note is that you assign server-side event handlers to the `OnItemCommand` and `OnItemDataBound` attributes.

You also add a `TemplateColumn` element to the grid and declare a `Button` control within it, giving it the `CommandName` value `"Delete"`. This will appear as the first column of the grid, and because you haven't changed the `AutoGenerateColumns` property from its default of `True`, the `DataGrid` control will automatically generate bound display columns for all the columns in the source rows as well.

The `ItemCommand` event will normally be raised when the user clicks the Delete button (which submits the form), but the client-side function will prevent the form from being submitted by canceling the event on the client if the user clicks Cancel in the confirmation dialog.

LISTING 6.8 The Declaration of the Form and `DataGrid` Control

```
<form id="frmMain" runat="server">

  <asp:DataGrid id="dgr1" runat="server"
       Font-Size="10" Font-Name="Tahoma,Arial,Helvetica,sans-serif"
       BorderStyle="None" BorderWidth="1px" BorderColor="#deba84"
       BackColor="#DEBA84" CellPadding="5" CellSpacing="1"
       DataKeyField="ProductID"
       OnItemCommand="DoItemCommand"
       OnItemDataBound="WireUpDeleteButton">
    <HeaderStyle Font-Bold="True" ForeColor="#ffffff"
                 BackColor="#b50055" />
    <ItemStyle BackColor="#FFF7E7" VerticalAlign="Top" />
    <AlternatingItemStyle backcolor="#ffffc0" />
    <Columns>
      <asp:TemplateColumn>
        <ItemTemplate>
          <asp:Button id="blnDelete" Text="Delete"
               CommandName="Delete" runat="server" />
        </ItemTemplate>
      </asp:TemplateColumn>
    </Columns>
  </asp:DataGrid>

</form>
```

Item and AlternatingItem Rows

The ItemDataBound event is called for every row in the DataGrid control, including header, footer, separator, selected, and edit rows, as well as the item and alternating item rows that you want to process. Also bear in mind that, even if you don't specify an AlternatingItem template (or any styling information for the alternating rows), the event handler will identify alternate rows as being of AlternatingItem type, so you need to test for both Item and AlternatingItem row types.

So how do you attach the client-side ConfirmDelete function to the Delete buttons in each row? You use the ItemDatabound event of the DataGrid control. This occurs for each row as it is bound to the data source to create the output shown in the page, and the code in Listing 6.8 specifies that the event handler named WireUpDeleteButton will be called each time this event is raised by the DataGrid control.

The WireUpDeleteButton Event Handler

Listing 6.9 shows the WireUpDeleteButton event handler, and you can see that the first task (as is usual when handling this event) is to make sure that you only process the correct type of row. You want to access the Delete button in every row where it occurs, so you must handle the event for both Item and AlternatingItem rows.

LISTING 6.9 The Code for the WireUpDeleteButton Event Handler

```
Sub WireUpDeleteButton(source As Object, e As DataGridItemEventArgs)

  ' make sure this is an Item or AlternatingItem row
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)
  If oType = ListItemType.Item _
  Or oType = ListItemType.AlternatingItem Then

    ' get ProductName value from this row
    Dim sName As String = e.Item.Cells(3).Text

    ' escape any single quotes
    sName = sName.Replace("'", "\'")

    ' get a reference to the Delete Button in this row
    Dim oCtrl As Button _
      = CType(e.Item.FindControl("blnDelete"), Button)

    ' attach the client-side onclick event handler
    oCtrl.Attributes.Add("onclick", _
      "return ConfirmDelete('" & sName & "');")

  End If

End Sub
```

When you find a suitable type of row, you get the text from the third cell in that row (the product name). The DataGrid control knows what the values that will be used to populate this row are when the ItemDataBound event occurs, even though it has not yet created the final markup that will appear in the page. Although in this case you extract the value from the output row, you could equally well query the row in the data source to which it is bound by using the following:

```
Dim sName As String = e.Item.DataItem("ProductID")
```

After you've extracted the product name, you must escape any single quotes it might contain. Otherwise, you'll get an error when you try to use the value in your client-side JavaScript function because a single quote will be treated as a string-termination character.

Then you can get a reference to the Delete button by using the FindControl method and attach the client-side function to the client-side click event by specifying it as the onclick attribute. Notice that you insert the product name from this row into the attribute to create the function parameter, and you include the return keyword so that the value of the function will be returned to the button control in the browser.

If you view the source of the page in the browser, you'll see the output that ASP.NET actually creates for each row, as in this example:

```
<input type="submit" value="Delete" ...
       onclick="return ConfirmDelete('Vegie-spread');" />
```

Now, if the user clicks the Delete button and then clicks Cancel in the confirmation dialog, the function returns false and the click event is not processed. The result is that the page is not submitted, so the row is not deleted.

> **An Easy Way to Use Client-Side Dialogs**
>
> **Chapter 7, "Design Issues for User Controls," describes some useful techniques for including client-side dialogs in applications. The chapter describes a user control that makes it easy to attach client-side script dialogs to elements in your ASP.NET pages.**

This chapter doesn't list the code that creates the DataReader instance, performs the data binding to the DataGrid control, or deletes the row when the Delete button is clicked. All this is conventional and is just the same as you would normally use to fill a DataGrid control and process user interaction. You can view all the code for this example by using the [view source] link at the bottom of the example page.

Detecting and Trapping Keypress Events

Web browsers, by default, allow the user to submit a form by pressing the Return key—even when the input focus is on another control. If there is more than one <form> section on a page, the browser should submit the one containing the element that currently has the focus. In fact, each browser behaves slightly differently:

- Internet Explorer switches the focus to the form's submit button and activates (that is, clicks) it. Even if there is more than one submit button on the current form, Internet Explorer always moves to and activates the first one.

- Netscape and Mozilla don't move the focus, but they always activate the first submit button.
- Opera is more intelligent than Internet Explorer, Netscape, and Mozilla. As you move the focus to and between elements on a form, Opera automatically sets the submit button as the default button, which is then activated when Return is pressed. However, when there is more than one submit button on a form, Opera changes the one that is the default (the one with the darker gray outline) as you move between the controls on the form. When you press Return, the submit button that follows the current control (within the buttons' declaration order in the page source) is activated.

Generally, the default behavior of all the browsers is fine. However, ASP.NET imposes a limitation on Web page structure in that there can be only one server-side `<form>` section. In other words, only one `<form runat="server">` control can be placed on a page.

> **Multiple Forms on Pages That Use the ASP.NET Mobile Control**
>
> **The limitation of a single form doesn't apply to pages that inherit from `MobilePage` and that are designed for use in small-screen devices such as cellular phones. These devices usually require pages that contain more than one `<form>` section to create the individual screens (called *cards*) that the device will display. (The set of cards is, not surprisingly, called a *deck*.)**

So there are really two issues here. You might want to trap the Return key so that it doesn't submit the form (or trap some other key so that it does not produce a character or carry out some other action). Alternatively, you might have more than one submit button on a form, perhaps because you want to offer the user more than one option when submitting the form. If you allow the Return key to be processed, the effect will always be that of the user clicking the first submit button on the form.

Listing 6.10 shows the code to detect a keypress event and discover which key was pressed. Notice that you enclose the key detection code in a function that accepts both a reference to the event and a key code value. If the user presses a key that generates a key code equal to the specified code, you return the value `false` from the function. Otherwise, you return the value `true`.

LISTING 6.10 A Function to Detect the Keypress Code and Return `true` or `false`

```
function trapKeypress(e, theKey) {
  var iKeyCode = 0;
  if (window.event) iKeyCode = window.event.keyCode
  else if (e) iKeyCode = e.which;
  return (iKeyCode != theKey);
}
```

You can attach the `trapKeypress` function to any control that exposes keypress events (`keydown`, `keypress`, or `keyup`). The important point is that you must return the value from the function to the element that raised the event, as in this example:

```
<element onkeypress="return trapKeypress(event, 13);">
```

Now the browser will ignore the keypress (in this example, the Return key with ANSI code 13) if the `trapKeypress` function returns `false` or process it as usual if the function returns `true`. Therefore, you can prevent the Return key from being processed by a control by attaching the `trapKeypress` function to that control (or to more than one control). To trap a different key, or more than one key, you would just have to pass the appropriate key code(s) to the function.

It's also possible to detect the state of the Ctrl, Shift, and Alt keys within a keypress event. The event object passed to the event handler for CSS2-compliant browsers exposes three `Boolean` properties named `altKey`, `ctrlKey`, and `shiftKey` that are `true` if the corresponding key was pressed when the event occurred. Internet Explorer 6.0 extends this by adding three more properties that allow you to tell if it was the Alt, Ctrl, or Shift key on the left side of the keyboard: `altLeft`, `ctrlLeft`, and `shiftLeft`. You'll see these properties in use in the next example.

> **Some Keypress Events Cannot Be Canceled**
>
> **For security reasons, you cannot trap and cancel keypresses that initiate system events. Although you can detect the keypress event and extract the key code, you cannot prevent key combinations that open menus or close the browser.**

Discovering the Key Codes You Need

As mentioned earlier, the key code returned from the `keypress` event is different from the key code returned from the `keydown` and `keyup` events for non-alphanumeric keys. To help you discover the key code you want, we've included a simple page within the examples for this book that displays the key codes for each event and the states of the Ctrl, Shift, and Alt keys.

Figure 6.3 shows that the `keydown` and `keyup` events always return the key code 65 for the *A* key, regardless of whether the Shift key is pressed as well; the `keypress` event returns the correct ANSI codes for both uppercase and lowercase letters.

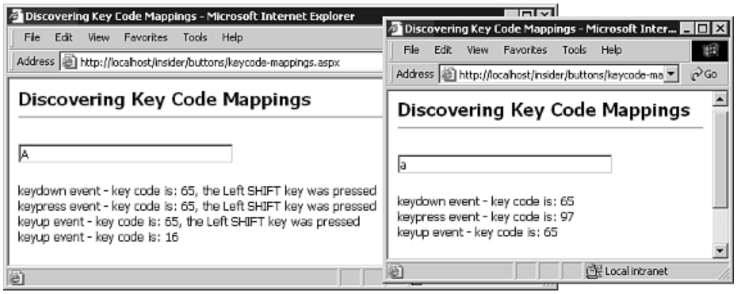


FIGURE 6.3 A sample page that displays key mappings and keypress information.

The relevant sections of the code in this page are shown in Listing 6.11, which demonstrates how you can collect the states of the Ctrl, Shift, and Alt keys as well as the actual key code.

LISTING 6.11 The Code for the Key Mappings Sample Page

```
<form>
 <input type="text" size="40" id="txtTest"
  value="Put cursor here and press a key"
```

LISTING 6.11 Continued

```
  onkeydown="showKeycode(event, 'keydown');"
  onkeypress="showKeycode(event, 'keypress');"
  onkeyup="showKeycode(event, 'keyup');" />
 <p />
 <div id="divResult"></div>
</form>
...
...
function showKeycode(e, sEvent) {
  var iKeyCode = 0;
  if (window.event) iKeyCode = window.event.keyCode
  else if (e) iKeyCode = e.which;
  var theDiv = document.getElementById('divResult');
  var theTextbox = document.getElementById('txtTest');
  if (sEvent == 'keydown') {
    theDiv.innerHTML = '';
    theTextbox.value = '';
  }
  theDiv.innerHTML += sEvent + ' event - key code is: '
                   + iKeyCode.toString()
  if (e.altKey == true)
    if (e.altLeft == true)
      theDiv.innerHTML += ', the Left ALT key was pressed'
    else theDiv.innerHTML += ', the ALT key was pressed';
  if (e.ctrlKey == true)
    if (e.ctrlLeft == true)
      theDiv.innerHTML += ', the Left CTRL key was pressed'
    else  theDiv.innerHTML += ', the CTRL key was pressed';
  if (e.shiftKey == true)
    if (e.shiftLeft == true)
      theDiv.innerHTML += ', the Left SHIFT key was pressed'
    else theDiv.innerHTML += ', the SHIFT key was pressed';
  theDiv.innerHTML += '<br />';
}
```

The `<form>` section of Listing 6.11 contains just the text box and the `<div>` element that displays the results. All three keypress events are wired up to a function named `showKeycode`, and they pass to this function a reference to the event object, together with the event name as a string to use to create the output seen in the page. Because you don't intend to cancel any keypresses, you don't return the value of the function to the control.

The next section of code in Listing 6.11 shows the function (`showKeycode`) that handles the three keypress events and displays the values you see in the page. If the event name is `keydown`, code in the `showKeycode` function first removes any existing content (generated by previous keypresses) from the text box and from the `<div>` element that displays the results. Then the output is generated, using the key code detected at the start of the function, and by appending the settings of the Ctrl, Shift and Alt keys.

Notice how the code sets the Internet Explorer extension properties for the left-hand keys to `true`, as well as the CSS2 standard properties, so you have to do a quick check to see what output to generate for each one. Browsers other than Internet Explorer will return `false` for the left-hand key properties that don't exist, so the `else` part of the `if` construct is executed in that case.

Trapping the Return Key in a Form

We need to look at one more detail of trapping keypress events. You've seen how to trap a keypress for a control, but often you'll have several controls on a form, so you'll need to attach the function to each one. However, by default, events bubble up through the control hierarchy of the page. This means that you can handle a keypress event for the containing element as well as at the individual control level. Obviously, the form itself is a container, so it's a good place to trap keypress events. You could also trap them at the page level by attaching the function directly to the opening `<body>` tag.

The sample page, shown in Figure 6.4, contains a single server-side `<form>` element that contains a selection of controls, including two submit buttons. The effect of the `<hr />` element used to separate the two sets of controls makes it look like there are two separate forms, but of course this isn't possible in an ASP.NET page. If you experiment with this page, you'll discover that you cannot submit the form by pressing the Return key.

> **Setting the Tab Order of Controls**
>
> **You should consider including the `TabIndex` attribute on form elements, especially where you want to trap keypress events. This allows you to control the order in which the input focus moves from one control to another when the Tab key is pressed, and providing a logical sequence makes it easier to work with complex forms. You set the `TabIndex` attribute when declaring server controls in ASP.NET, or you can set the `TabIndex` property dynamically at runtime, to an `Integer` value that denotes the index of the control within the tab order of the page:**
>
> ```
> <asp:TextBox runat="server"
>      TabIndex="3" />
> ```
>
> **For non-server controls, you just add a `TabIndex` attribute in the usual way:**
>
> ```
> <input type="text" tabindex="3" />
> ```

The client-side code used in this page (see Listing 6.12) is basically the same as the code in Listing 6.10. However, notice that this time you declare a page-level variable named `blnReturn`, and you use the keypress event to set its value to `false` if the user pressed the Return key or `true` otherwise.

LISTING 6.12 The Client-Side Code to Trap and Store the Keypress Information

```
<script language="javascript">
<!--

var blnReturn = true;
```

LISTING 6.12 Continued

```
function trapReturn(e) {
  var iKeyCode = 0;
  if (window.event) iKeyCode = window.event.keyCode
  else if (e) iKeyCode = e.which;
  blnReturn = (iKeyCode != 13);
}

//-->
</script>
```

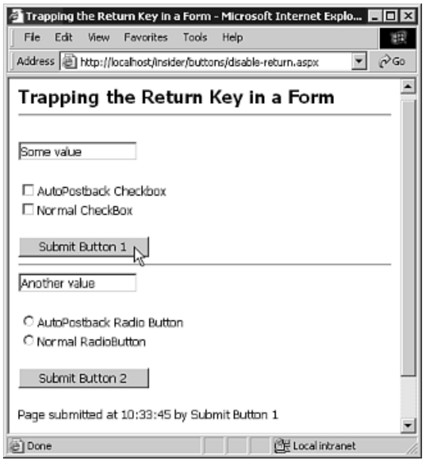


FIGURE 6.4 The sample page that traps the Return key.

Storing the Key Code Test Result

The reason for using a separate variable to store the result of the key code test is that you need to handle the `submit` event of the form and return the value `false` from the `onsubmit` event handler if you want to prevent the form from being submitted. You can't perform the key code test in the `onsubmit` event handler because the keypress information is not available when this event is raised. Instead, you capture the result of the test in the keypress event and store it in a variable, as shown in Listing 6.12, when the keypress event occurs.

Effectively, the `blnReturn` variable reflects the validity of the last keypress, and you can then use the value of this variable in the `submit` event of the form. If the last keypress was the Return key, `blnReturn` is `false` and the form is not submitted. Listing 6.13 shows the HTML declarations for the page in Figure 6.4, and you can see the two event handler attributes attached to the opening `<form>` tag.

LISTING 6.13 The Declaration of the HTML `<form>` Section Within the Sample Page

```
<form id="frmMain" runat="server"
      onkeydown="trapReturn(event);"
      onsubmit="return blnReturn;">

  <asp:TextBox id="txtTest1" Text="Some value" runat="server" />
  <p />
  <asp:CheckBox id="chkTest1" AutoPostback="True"
       Text="AutoPostback Checkbox" OnCheckedChanged="ButtonClick"
       runat="server" /><br />
  <asp:CheckBox id="chkTest2" Text="Normal CheckBox" runat="server" />
  <p />
  <asp:Button id="btnOne" CommandName="Button 1"
       Text="Submit Button 1" runat="server" OnClick="ButtonClick" />

<hr />

  <asp:TextBox id="txtTest2" Text="Another value" runat="server" />
  <p />
  <asp:RadioButton id="optTest1" GroupName="grp1"
       AutoPostback="True" Text="AutoPostback Radio Button"
       OnCheckedChanged="ButtonClick" runat="server" /><br />
  <asp:RadioButton id="optTest2" GroupName="grp1" Checked="True"
       Text="Normal RadioButton" runat="server" />
  <p />
  <asp:Button id="btnTwo" CommandName="Button 2"
       Text="Submit Button 2" OnClick="ButtonClick" runat="server" />
  <p />
  <asp:Label id="lblMsg" EnableViewState="False" runat="server" />

</form>
```

Listing 6.13 also shows the declarations of the other controls placed on the form, as well as the two submit buttons. None of these controls require any client-side event handlers because the keypress events will bubble up to the `<form>` element and be trapped there. However, you need *server-side* event handler declarations for some of the controls so that they call the ASP.NET routine named `ButtonClick` if they are used to initiate a postback.

The `ButtonClick` event handler is shown in Listing 6.14. You can see that all it does is display the current time (so that you can easily tell whether the form was submitted) and the text of the control that caused the postback.

LISTING 6.14 The Server-Side Code That Displays Information when the Page Is Submitted

```
<script runat="server">

Sub ButtonClick(sender As Object, e As EventArgs)

  ' display time page was last submitted
  lblMsg.Text = "Page submitted at " _
              & DateTime.Now.ToString("hh:mm:ss") _
              & " by " & sender.Text

End Sub

</script>
```

Creating a `MaskedEdit` Control

As well as the `ComboBox` control described in Chapter 5, "Creating Reusable Content," there is at least one other control missing from the standard set provided by Web browsers—a `MaskedEdit` control. This is really just a text box that allows only specific characters to be entered, depending on the *mask* (that is, the character-by-character definition of the string that is acceptable).

Let's look at a simple example of a `MaskedEdit` control that demonstrates some useful techniques you can adapt to your own applications. You'll convert it into a user control and a custom server control in later chapters, but for now, you should just look at the actual control implementation.

One other feature of the sample control is interesting. You can see in Figure 6.5 that the text box displays the mask as a series of light gray underscores and literal characters (such as the hyphens between the number groups in this case). You'll see how this is achieved after you look at the rest of the code in the page.

FIGURE 6.5 The `MaskedEdit` control sample page in action.

Trapping and Handling the Keypress Events

You've seen techniques for trapping keypress events and extracting the key code information in previous examples in this chapter. The `MaskedEdit` control obviously uses much the same techniques to catch each keypress and figure out whether the character the user typed is valid for the current location in the text box (in other words, whether it matches the mask).

The client-side code section of the sample page comprises four functions and some page-level variable declarations. These are the functions:

- **`doKeyDown`**—This function is executed when the user presses a key. Its task is to cancel any keypresses that the control cannot support. With a few exceptions, it cannot handle nonprintable characters.
- **`doKeyPress`**—This function is executed when the user releases a key. It checks the key code against the mask and cancels it if it is not valid. In cases where an uppercase letter is expected, the code automatically converts lowercase letters to uppercase and accepts them.
- **`doKeyUp`**—This function is executed when the user releases a key. Its task is to add to the text box any literal characters that follow the current character so that the user does not have to enter them manually. It also creates a message in the status bar that indicates the next character that is expected.
- **`doFocus`**—This function is executed when the control first receives the focus. It just has to make sure that any literal characters at the start of the mask are inserted into the text box. It does this by calling the `doKeyUp` function.

The mask can contain only the characters shown in Table 6.3.

TABLE 6.3

The Characters That Can Be Used to Define the Mask for the `MaskedEdit` Control

Character	Allows Only...
a	Lowercase or uppercase letters, or the numbers 0 to 9.
A	Uppercase letters or the numbers 0 to 9.
l	Lowercase or uppercase letters, but not numbers.
L	Uppercase letters, but not numbers.
n	Only the numbers 0 to 9.
?	Any printable character.

Listing 6.15 shows the page-level variables. You can see the string that contains the mask characters and a string you use to define alphabetic characters. This second string is also used to extract the ANSI/Unicode character code and to convert letters to uppercase. The `bStarting` variable is used by the `doKeyUp` function to force it to check whether there are any literal characters at the start of the mask, which it must insert into the text box when it first gets the focus.

LISTING 6.15 The Page-Level Variables and the Handler for the `keydown` Event

```
var sMaskSet = 'aAlLn?'
var sUAscii = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
var bStarting = true;
```

LISTING 6.15 Continued

```
function doKeyDown(e, textbox, sMask) {
  // trap and cancel keys that are not appropriate
  var iKeyCode = 0;    // collect key code
  if (window.event) iKeyCode = window.event.keyCode;
  else if (e) iKeyCode = e.which;
  if (iKeyCode == 32 || iKeyCode == 39 || iKeyCode == 35
  || iKeyCode == 8 || iKeyCode == 9)
    return true;       // space left end backspace tab
  if (iKeyCode < 47)   // non-printable character
    return false;
}
```

Handling the `keydown` Event

Listing 6.15 shows the `doKeyDown` function, which returns `false` if the key code represents one of the nonprintable values that cannot be accepted. This forces the text box to ignore that keypress event.

Handling the `keypress` Event

Listing 6.16 shows the `doKeyPress` event handler, which is executed next if the `doKeyDown` function returned `true`. After clearing the status bar, you extract the key code and then see whether the end of the mask has already been reached. If it has, the only keypress you can accept is the Backspace key (code 8). Otherwise, you return `false` to cancel the keypress and leave the text box value as it already stands.

LISTING 6.16 The Client-Side Handler for the `keypress` Event

```
function doKeyPress(e, textbox, sMask) {

  window.status = '';
  var iKeyCode = 0;    // collect key code
  if (window.event) iKeyCode = window.event.keyCode;
  else if (e) iKeyCode = e.which;

  // check if mask already filled, and not backspace
  var iLength = textbox.value.length;
  if ((iLength == sMask.length) && (iKeyCode != 8))
    return false;

  // get mask character for this position in textbox
  var sMaskChar = sMask.charAt(iLength);

  // see if it's a special character
  if (sMaskSet.indexOf(sMaskChar) > -1) {
```

LISTING 6.16 Continued

```
// masked character required
switch (sMaskChar) {

  case 'a':   // any alphanumeric character
    if ((iKeyCode > 47 && iKeyCode < 58)
    || (iKeyCode > 64 && iKeyCode < 91)
    || (iKeyCode > 96 && iKeyCode < 123))
      return true
    else return false;

  case 'A':   // uppercase alphanumeric character
    if ((iKeyCode > 47 && iKeyCode < 58)
    || (iKeyCode > 64 && iKeyCode < 91))
      return true
    else if (iKeyCode > 96 && iKeyCode < 123) {
      textbox.value += sUAscii.charAt(iKeyCode - 97);
      return false;
    }
    else return false;

  case 'l':   // any letter
    if ((iKeyCode > 64 && iKeyCode < 91)
    || (iKeyCode > 96 && iKeyCode < 123))
      return true
    else
      return false;

  case 'L':   // uppercase letter
    if (iKeyCode > 64 && iKeyCode < 91)
      return true
    else if (iKeyCode > 96 && iKeyCode < 123) {
      textbox.value += sUAscii.charAt(iKeyCode - 97);
      return false;
    }
    else return false;

  case 'n':   // any numeric character
    if (iKeyCode > 47 && iKeyCode < 58)
      return true
    else return false;
  case '?':   // any character
    return true;

  default: return false;
```

LISTING 6.16 Continued

```
    }
  }
  else
    return true;
}
```

If the mask is not yet filled, you then examine it to see whether the character the user typed matches the mask. This code uses a rather long `switch` statement, mainly to make it easy to see how it works. You might prefer to create smaller functions or more compact code for your own implementation. You return `true` if the key matches the mask or `false` to cancel the event if the key does not match the mask. Notice that lowercase letters are automatically converted to uppercase where the mask value is `'A'` or `'L'`.

Handling the `keyup` Event

After the `doKeyPress` event has been processed, the `keyup` event is raised, and you handle it with the `doKeyUp` function shown in Listing 6.17. If the `bStarting` variable is `false`, you know that the user has typed something into the text box, so you again extract the key code from the event passed to the function.

LISTING 6.17 The Client-Side Handler for the `keyup` Event

```
function doKeyUp(e, textbox, sMask) {
  if (bStarting != true) {
     var iKeyCode = 0;    // collect key code
     if (window.event) iKeyCode = window.event.keyCode;
     else if (e) iKeyCode = e.which;
     if (iKeyCode < 47 && iKeyCode != 32) return;
  }
  // check if next mask characters are literals
  // and add to text box if they are
  while ((textbox.value.length < sMask.length) &&
  (sMaskSet.indexOf(sMask.charAt(textbox.value.length)) == -1)) {
    textbox.value += sMask.charAt(textbox.value.length);
  }
  var sNext;
  if (textbox.value.length == sMask.length)
    sNext = 'Complete'
  else
    switch (sMask.charAt(textbox.value.length)) {
     case 'a':
       sNext = 'Expecting any alphanumeric character (0-9,A-Z,a-z)';
       break;
     case 'A':
       sNext = 'Expecting an uppercase alphanumeric char (0-9,A-Z)';
```

LISTING 6.17 Continued

```
        break;
      case 'l':
        sNext = 'Expecting any letter (A-Z, a-z)';
        break;
      case 'L':
        sNext = 'Expecting an uppercase letter (A-Z)';
        break;
      case 'n':
        sNext = 'Expecting any numeric character (0-9)';
        break;
      case '?':
        sNext = 'Expecting any character';
        break;
      default: sNext = '';
    }
  window.status = sNext;
}
```

Then, using a `while` construct, you add to the text box any literal characters that appear at the start of the mask. The two conditions that must be met for the `while` loop to execute are that the length of the mask must be greater than the length of the text in the text box and the current character must not be one of the special mask characters which indicate that the user must enter a value.

So the code first compares the length of the mask with the length of the text in the text box to make sure that execution of the `while` loop stops at the end of the mask. Then it checks whether the current character in the value in the text box is also present in the string that contains the valid mask characters (`sMask`). If it is, this means that it *is* one of the special placeholders that indicate the kind of value that the user must enter, so the `while` loop just moves to the next character. If it is *not* a valid mask character, then it must be a literal character, so it is added to the string value in the text box.

After this, you can create the prompt indicating the next character type that is expected and display that in the browser's status bar.

Meanwhile, the variable `bStarting` will be `true` if the user hasn't entered anything into the text box yet (in other words, if this is the first keypress event). At this point, you want to insert any literal characters that appear at the start of the mask string, and you achieve this by handling the `focus` event for the text box, as shown in the next section.

Handling the `focus` Event

Listing 6.18 shows the function that is executed when the text box gets the focus. You simply set the `bStarting` value to true, call the `onKeyUp` function, and then set `bStarting` back to `false` again. This causes the `doKeyUp` function to add any literal characters and display the prompt in the status bar, but without attempting to extract the key code first.

LISTING 6.18 The Client-Side Event Handler for the `focus` Event

```
function doFocus(e, textbox, sMask) {
  bStarting = true;
  doKeyUp(e, textbox, sMask);
  bStarting = false;
}
```

Validating the Value the User Enters

Although the `MaskedEdit` control works reasonably well, you might decide to add an ASP.NET `RegularExpressionValidator` control to the page as well to ensure that the input actually does match the mask when submitted. This would also have the advantage of validating the value on the server side after the page is submitted—something you should always do to prevent the server from being spoofed by the user creating a dummy page that contains invalid values.

Together, the four functions `doKeyDown`, `doKeyPress`, `doKeyUp`, and `doFocus` implement the complete `MaskedEdit` control feature. There are some limitations, due mainly to the fact that the browser security model prevents canceling of some keypress and other events, and the text box control in the browser does not offer all the features of controls you might be used to in, for example, a Windows Forms or executable application. One particular issue is that users can click on the text box to reposition the input cursor, thereby breaking the mask code.

Using the `MaskedEdit` Control

Listing 6.19 shows the HTML declarations of the controls in the page shown in Figure 6.5. The two drop-down lists are populated with the four sample mask strings and three text sizes, from which you can select to experiment with the control. The `MaskedEdit` control is declared as an ordinary ASP.NET `TextBox` control, and you'll add the event handlers that perform the magic to it in the server-side `Page_Load` event later in this chapter.

LISTING 6.19 The HTML Declarations in the `MaskedEdit` Control Sample Page

```
<form id="frmMain" runat="server">

  <asp:DropDownList id="selMask" AutoPostback="True" runat="server">
    <asp:ListItem Value="nnnn-nn-nnTnn:nn:nn" Text="UTC Date and Time" />
    <asp:ListItem Value="Qnnnnn-LLnn" Text="Part Number" />
    <asp:ListItem Value="(nnn)-nnn-nnnn" Text="US Phone Number" />
    <asp:ListItem Value="LLn? nLL" Text="UK Postal Code" />
  </asp:DropDownList>
  <asp:DropDownList id="selSize" AutoPostback="True" runat="server">
    <asp:ListItem Value="10" Text="10 pt" />
    <asp:ListItem Value="12" Text="12 pt" />
    <asp:ListItem Value="16" Text="16 pt" />
  </asp:DropDownList><p />
```

LISTING 6.19 Continued

```
  <asp:TextBox id="txtMaskEdit" Columns="25" runat="server" /> <p />

</form>
```

The Server-Side `Page_Load` Event Handler

The `Page_Load` event handler is shown in Listing 6.20. In it you collect the mask string and font size from the drop-down lists in the page, and you specify the font name. Then you apply these font details to the text box. Here you're using the Courier New font. You need a monospaced (fixed-pitch) font so that the characters typed into the text box will line up correctly with the light-gray placeholders.

LISTING 6.20 The `Page_Load` Event Handler for the `MaskedEdit` Control Demonstration Page

```
Sub Page_Load()

  Dim sMask As String = selMask.SelectedValue
  Dim sFont As String = "Courier New"
  Dim sSize As String = selSize.SelectedValue

  txtMaskEdit.Text = ""
  txtMaskEdit.Style("font-family") = sFont
  txtMaskEdit.Style("font-size") = sSize & "pt"

  Dim sQuery As String = sMask
  sQuery = sQuery.Replace("a", "_")
  sQuery = sQuery.Replace("A", "_")
  sQuery = sQuery.Replace("l", "_")
  sQuery = sQuery.Replace("L", "_")
  sQuery = sQuery.Replace("n", "_")
  sQuery = sQuery.Replace("?", "_")
  sQuery = Server.UrlEncode(sQuery)
  sFont = Server.UrlEncode(sFont)

  txtMaskEdit.Style("background-image") _
    = "url(mask-image.aspx?mask=" _
    & sQuery & "&font=" & sFont & "&size=" & sSize & "&cols=" _
    & txtMaskEdit.Columns.ToString() & ")"

  Dim sTip As String = sMask
  sTip = sTip.Replace("a", "[a]")
  sTip = sTip.Replace("A", "[A]")
  sTip = sTip.Replace("l", "[l]")
  sTip = sTip.Replace("L", "[L]")
  sTip = sTip.Replace("n", "[n]")
```

LISTING 6.20 Continued

```
  sTip = sTip.Replace("?", "[?]")
  txtMaskEdit.ToolTip = "Mask: " & sTip & vbCrlf & " where:" _
    & vbCrlf & "[a] = any alphanumeric character (0-9, A-Z, a-z)" _
    & vbCrlf & "[A] = an uppercase alphanumeric char (0-9, A-Z)" _
    & vbCrlf & "[l] = any letter character (A-Z, a-z)" _
    & vbCrlf & "[L] = an uppercase letter character (A-Z)" _
    & vbCrlf & "[n] = any numeric character (0-9)" _
    & vbCrlf & "[?] = any character"

  txtMaskEdit.Attributes.Add("onkeydown", _
    "return doKeyDown(event, this, '" & sMask & "')")
  txtMaskEdit.Attributes.Add("onkeypress", _
    "return doKeyPress(event, this, '" & sMask & "')")
  txtMaskEdit.Attributes.Add("onkeyup", _
    "return doKeyUp(event, this, '" & sMask & "')")
  txtMaskEdit.Attributes.Add("onfocus", _
    "return doFocus(event, this, '" & sMask & "')")

End Sub
```

Where do the light-gray placeholders come from, and how do you get them into the text box? In this example you're taking advantage of the fact that you can specify an image for the background of most controls—including a text box—under the CSS2 recommendations. So all you have to do is create a suitable image that contains the placeholder characters and assign it to the text box's `background-image` style selector. The text that the user types into the text box will then overlay the image, giving the effect shown in Figure 6.5.

The sample page uses a separate ASP.NET page named `mask-image.aspx` to generate the required image dynamically at runtime. The code in the `Page_Load` event creates the URL that will load this page. It also appends as the query string the mask string as it will appear in the text box (all the special characters that denote values the user must type are replaced with underscores), the font name, the font size, and the value of the `Columns` property of the text box. All this information is required to be able to create the appropriate image.

You also want to provide a pop-up ToolTip for the text box that makes it easy for the user to understand what input is required. So the next stage in the `Page_Load` event handler is to build a suitable string and assign it to the `ToolTip` property of the text box. If you embed carriage returns into the ToolTip string, Internet Explorer will break up the string to give a neater display (although unfortunately other browsers ignore the carriage returns). Figure 6.6 shows the ToolTip as it appears in Internet Explorer 6.

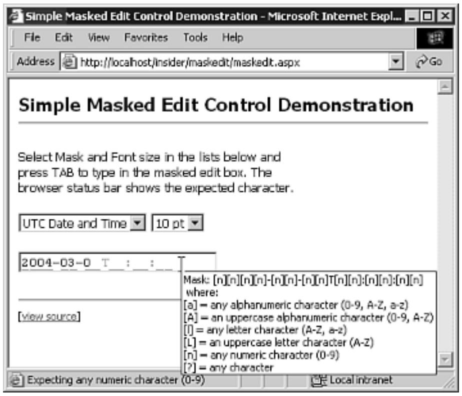


FIGURE 6.6 The MaskedEdit control page, showing a mask and the corresponding ToolTip.

The final stage in the Page_Load event is to attach the client-side functions to the text box to turn the text box into a MaskedEdit control. As demonstrated several times already in this chapter, you specify the values required for the client-side function parameters. For each function, you must pass in a reference to the client-side event (using the event keyword), a reference to the current control (using the this keyword), and the mask to use. Here's an example:

```
txtMaskEdit.Attributes.Add("onkeydown", _
  "return doKeyDown(event, this, '" & sMask & "')")
```

The result is that the Page_Load event now creates an <input type="text"> control with a range of attributes. Listing 6.21 shows the output that is generated when this page is viewed in a browser. This code shows the ToolTip with embedded carriage returns, the four event handler attributes, and the style declarations that specify the font and the background image.

LISTING 6.21 The Output Generated for the MaskedEdit Control when the Page Is Viewed in a Browser

```
<input name="txtMaskEdit" type="text" size="25" id="txtMaskEdit"
  title="Mask: [n][n][n][n]-[n][n]-[n][n]T[n][n]:[n][n]:[n][n]
         where:
         [a] = any alphanumeric character (0-9, A-Z, a-z)
         [A] = an uppercase alphanumeric character (0-9, A-Z)
         [l] = any letter character (A-Z, a-z)
         [L] = an uppercase letter character (A-Z)
         [n] = any numeric character (0-9)
         [?] = any character"
  onkeydown="return doKeyDown(event, this, 'nnnn-nn-nnTnn:nn:nn')"
  onkeypress="return doKeyPress(event, this, 'nnnn-nn-nnTnn:nn:nn')"
  onkeyup="return doKeyUp(event, this, 'nnnn-nn-nnTnn:nn:nn')"
  onfocus="return doFocus(event, this, 'nnnn-nn-nnTnn:nn:nn')"
  style="font-family:Courier New;font-size:10pt;background-image
     :url(mask-image.aspx?mask=____-__-__T__%3a__%3a__&
     font=Courier+New&size=10&cols=25);" />
```

Generating the Background Mask Image

The only remaining code you need to examine now is that which generates the image for the background of the text box. Generating images dynamically before the .NET Framework came along was hard, and most Web developers relied on custom COM components created by third-party suppliers. However, the .NET Framework removes much of the complexity from creating images dynamically. This is not to say that you still won't find many great image components and server controls around—and for complex tasks, these components and controls can save a huge amount of development time.

Nevertheless, the requirements for this example are simple. You just need to create an image that contains a text string. Listing 6.22 shows the complete code for the page `mask-image.aspx`, which generates the image and returns it as a stream that represents a GIF file.

LISTING 6.22 The ASP.NET Page That Generates the Mask Image for the Text Box

```
<%@Page Language="VB" %>
<%@Import Namespace="System.Drawing" %>
<%@Import Namespace="System.Drawing.Imaging" %>

<script runat="server">

Sub Page_Load()

  ' set content-type of response so client knows it is a GIF image
  Response.ContentType="image/gif"

  ' get mask and font details from query string and URL-decode
  Dim sText As String = Server.UrlDecode(Request.QueryString("mask"))
  Dim sFont As String = Server.UrlDecode(Request.QueryString("font"))
  Dim sSize As String = Request.QueryString("size")
  Dim sCols As String = Request.QueryString("cols")

  Dim iWidth, iHeight As Integer
  iWidth = Integer.Parse(sSize) * Integer.Parse(sCols)
  iHeight = Integer.Parse(sSize) * 3

  ' create a new bitmap
  Dim oBitMap As New Bitmap(iWidth, iHeight)

  ' create new graphics object to draw on bitmap
  Dim oGraphics As Graphics = Graphics.FromImage(oBitMap)

  ' create the rectangle to hold the text
  Dim oRect As New RectangleF(0, 0, oBitMap.Width, oBitMap.Height)

  'create a solid brush for the background and fill it
```

LISTING 6.22 Continued

```
  Dim oBrush As New SolidBrush(Color.White)
  oGraphics.FillRectangle(oBrush, oRect)

  ' create a Font object for the text style
  Dim oFont As New Font(sFont, Single.Parse(sSize))

  ' create a brush object and draw the text
  oBrush.Color = Color.FromArgb(153, 153, 153)
  oRect = New RectangleF(-1, 1, oBitMap.Width, oBitMap.Height)
  oGraphics.DrawString(sText, oFont, oBrush, oRect)

  ' write bitmap to response
  oBitmap.Save(Response.OutputStream, ImageFormat.Gif)

  ' dispose of objects
  oBrush.Dispose()
  oGraphics.Dispose()
  oBitmap.Dispose()

End Sub

</script>
```

Notice that you have to import the `Drawing` and `Imaging` namespaces to be able to use the classes they contain. You also have to set the `ContentType` value for the page to `"image/gif"` so that the browser will treat it as an image. After this, you extract the values from the query string that you need to build the image. You URL-encoded them in the `Page_Load` event handler when you created the query string (because some of them contain spaces or other non-URL-legal characters), so you have to decode them first.

Calculating the Size of the Image and the Bitmap

You need to make sure your image is large enough to fill the text box, or you'll get multiple copies tiled over the background. However, you don't want to make it any bigger than necessary because you want to minimize download times to achieve the fastest possible rendering. You use the number of columns and the font size to give an image of sufficient width and height.

The code then creates a new `Bitmap` instance of that size and from it a `Graphics` object that you can use to draw and write on the image. By default the image is black (with a pixel value zero), so you create a rectangle the same size as the image and a new white `SolidBrush` object. The `Fill` method of the brush then paints the image white.

Drawing the Text

To draw the text, you need to create an instance of a `Font` object that represents the font and size specified by the values in the query string, and then you need to change the color of the

SolidBrush object to light gray. Then it's simply a matter of defining a rectangle where you want to draw the text (you have to adjust the top and left values slightly to get the best lineup possible in the text box) and writing the text onto the Bitmap instance. To return the Bitmap instance, you save it directly to the ASP.NET Response object's current OutputStream instance, specifying the image format you want.

Usability Issues with the MaskedEdit Control

Although the MaskedEdit control is neat, easy to use, and works quite well, you'll probably discover a few shortcomings when you start to experiment with it. We mentioned the difficulties in absolutely controlling the user's keypresses and mouse clicks earlier in this chapter, and you might want to extend the code to try to handle these more accurately. There is also the issue that, if you type quickly, the client-side event handler code cannot keep up, and it misses the literal characters in the mask string.

Another issue that you'll come across concerns the background mask image. The actual size and spacing of the characters on the bitmap that the mask-image.aspx page generates depend on the environment of the server (the screen resolution, the installed fonts, and other internal parameters). However, the text that is displayed in the text box as the user types depends on the settings of the user's machine (that is, the client machine). You are likely to find slight misalignment occurring in some cases.

Having said all this, the techniques demonstrated for creating images dynamically and for handling keypress events are still valid—and you will no doubt find many other uses for them in your own applications.

Creating a One-Click Button

Now that you've seen how to use client-side code to detect keypress events, let's move on to talk about how you can use client-side code and/or server-side code to prevent users from clicking a button on a page more than once—or at least detecting if they do so.

There are several ways to approach this problem, and the example used here demonstrates four of the most obvious solutions:

- Disable the button as soon as it is clicked, by handling the onclick event with *client-side* code, and setting the disabled property to true. If the form has more than one submit button or uses AutoPostback on other controls, you also have to disable those controls at the same time. Remember that some browsers (especially older ones) do not allow controls to be disabled.
- Trap the client-side onclick event of the button and set a client-side variable to true; then prevent the button from being clicked again while this value is true by returning false from the event handler.
- Set a client-side variable to true as soon as the form is submitted and prevent it from being submitted again while this value is true. This is useful if the form has more than one submit button or uses AutoPostback on other controls because you don't need to change the properties of these controls (as you would with the first method). However, this approach does not give the user visual feedback that the button is disabled.

- Allow the user to submit the form multiple times but detect this on the server and carry out the required processing only the first time the form is submitted. Again, there is no visual feedback for the user with this method, but this approach works when the client's browser does not support client-side scripting (or the user has disabled it).

> **Disabled Buttons in Opera**
>
> **Opera, even in version 7, does not gray out a button or control that is disabled. However, it does correctly prevent the user from clicking a button or activating a control that has the `disabled` property set to `true`. Someone once told me that Opera was so named because it was designed to keep the other browser manufacturers on their toes with regard to performance, usability, and features. But if this were the case, surely it would have been named Ballet.**

Figure 6.7 shows the sample page after the button has been clicked. You can see that, under the text box, a message indicating how many times the page has been submitted so far is displayed. By default, all three methods of preventing the page from being submitted more than once are enabled, and they are processed in the same order as the check boxes on the page. You can turn off each one to see the remaining methods in action; we'll look at what effects this has shortly.

FIGURE 6.7 The one-click button demonstration page in action.

Figure 6.8 shows a schematic view of how the controls on the sample page affect the way that it runs and which of the four techniques for preventing multiple button clicks or multiple server-side page processing are employed. The three decision boxes correspond to the three check boxes in the page.

The Code to Implement a One-Click Button

The visible part of the sample page is created using the HTML shown in Listing 6.23. None of the controls has a client-side event handler attached in the declaration shown here; you'll be adding them dynamically at runtime.

LISTING 6.23 The Form Section of the One-Click Button Sample Page

```
<form id="frmMain" runat="server">

 <asp:TextBox id="txtTest" Text="Required value" runat="server" />
 <asp:Button id="btnOneClick" Text="Click me" runat="server" />
```

LISTING 6.23 Continued

```
<asp:Label id="lblMsg" EnableViewState="False" runat="server" />

<asp:Checkbox id="chkNoDisable" runat="server"
     Text="Do not disable button after first click." />
<asp:Checkbox id="chkAllowClick" runat="server"
     Text="Allow multiple button clicks to be processed." />
<asp:Checkbox id="chkAllowSubmit" runat="server"
     Text="Allow page to be submitted again while processing." />

</form>
```

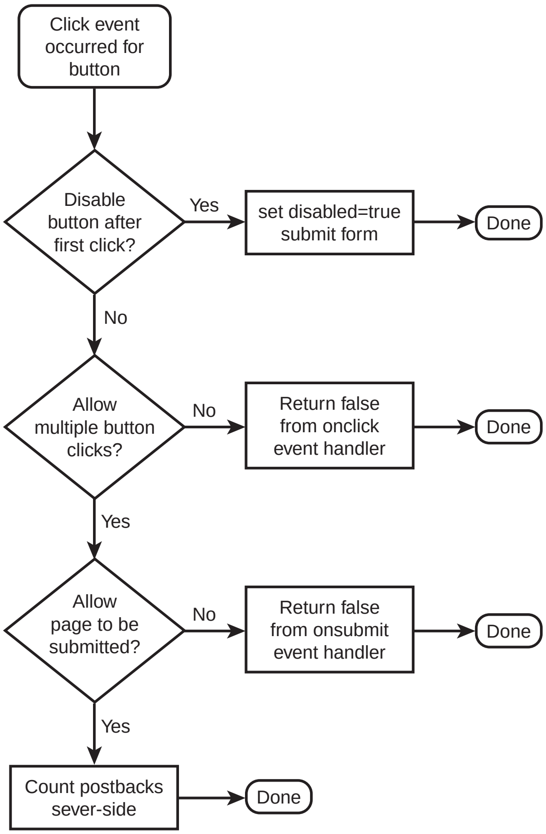


FIGURE 6.8
A schematic of the processes for preventing multiple page submissions in the one-click button example.

Setting the `disabled` Property of the Button to `true`

When all three methods for preventing multiple form submissions are enabled, the one that actually prevails is the one that disables the submit button as soon as it's clicked. It's easy enough to do this; you just attach a client-side function that sets the `disabled` property of the button to `true` in that button's `onclick` event.

However, you can't do this directly within the declaration of the submit button in this example because you've used an ASP.NET `Button` server control, and the `OnClick` attribute sets the *server-side* event handler (not the client-side one). If you write this:

```
<asp:Button text="Submit" runat="server"
     onclick="MyServerCode" />
```

you can expect the server-side routine or function named `MyServerCode` to be executed when the button is clicked, after the page has been submitted to the server. One way you can get around this is to use the ordinary HTML server controls instead of the ASP.NET Web Forms controls. The button control implemented by the `HtmlInputButton` class exposes the `OnServerClick` event handler property to define code that runs on the server, allowing you to use the `onclick` attribute to specify the client-side event handler:

```
<input type="submit" value="Submit" runat="server"
       onserverclick="MyServerCode"
       onclick="MyClientSideCode();"/>
```

The other approach is to use a Web Forms control but add the client-side attribute dynamically when creating the page on the server. This allows the client-side `onclick` functionality to coexist with the server-side event handling. When the button is clicked, the client-side code runs first, and then, after the page is posted back to the server, any ASP.NET server-side event handler attached to the control is invoked:

> **`HtmlControls` Versus `WebControls` Property Names**
>
> **Remember that you have to use the ordinary HTML attribute names with the standard controls from the `System.Web.UI.HtmlControls` namespace. For example, the caption of a button is set with the `Value` property and not with the `Text` property.**

```
control.Attributes.Add("onclick", "MyClientSideCode()")
```

The sample page uses this technique. In the server-side `Page_Load` event handler, you specify that the client-side function named `buttonClick` will be executed when the button is clicked. You pass to this function a reference to the current control (using the `this` keyword) and assign the return value to the event:

```
btnOneClick.Attributes.Add("onclick", "return buttonClick(this);")
```

The Client-Side `buttonClick` Event Handler

Listing 6.24 shows the `buttonClick` client-side event handler that is called when the button on the sample page is clicked. In theory, all you actually need to do to prevent it from being clicked again is to set the `disabled` property to `true`, using the following:

```
buttonOneClick.disabled = true;
```

However, in Internet Explorer and Opera, this prevents the form from being submitted the first time as well (although it works as expected in Netscape and Mozilla). This means that you have to submit the form programmatically within the code, after setting the `disabled` property of the button:

```
buttonOneClick.disabled = true;
document.forms[0].submit();
```

You could even do this directly in the declaration of the button, rather than writing a function and calling it from the `onclick` attribute. However, because you are implementing several techniques in the same page, the event handlers are a little more complicated. When the button is clicked, you look to see if the first check box is selected. If it is not, you disable the button to prevent it from being clicked again.

LISTING 6.24 The Client-Side `buttonClick` Event Handler and Timer Routines

```
var bButtonClicked = false;

function buttonClick(ctrl) {
  // check value of first checkbox
  var theForm = document.forms[0];
  if(theForm.elements['chkNoDisable'].checked == false) {
    // first checkbox is not ticked
    // disable submit button
    ctrl.disabled = true;
    startTimer();
    theForm.submit();
  }
  // check value of second checkbox
  if(theForm.elements['chkAllowClick'].checked == false) {
    // second checkbox is not ticked
    if (bButtonClicked == false) {
      // first time button was clicked
      bButtonClicked = true;
      startTimer();
      return true;
    }
    else {
      // prevent button event from being executed
      return false;
    }
  }
  else {
    // second checkbox is ticked
    // allow button event to continue
    startTimer();
    return true;
  }
```

LISTING 6.24 Continued

```
}

function startTimer() {
  // display "Please wait" message
  var label = document.getElementById('lblMsg');
  label.innerHTML = '<b>Please wait.</b>';
  // start interval timer for one second
  window.setTimeout('showProgress()', 1000);
}

function showProgress() {
  // update "Please wait" text
  var label = document.getElementById('lblMsg');
  label.innerHTML += '<b>.</b>';
  // restart interval timer for one second
  window.setTimeout('showProgress()', 1000);
}
```

The sample page contains a couple routines that start and then reset a timer within the page, to provide a progress indicator showing that the server is processing the page. (You simulate a long process taking place on the server side, as you'll see shortly.) You can see the two timer functions, named `startTimer` and `showProgress`, at the end of Listing 6.24. After disabling the button, you call the routine to start the timer and then submit the form by calling its `submit` method (as discussed earlier). The result is shown in Figure 6.9.

FIGURE 6.9 The progress indicator that runs while the page is being processed.

Trapping the `click` Event for the Button

The `buttonClick` event shown in Listing 6.24 continues by looking to see if the second check box is selected. If it isn't, you want to prevent more than one button click from being processed. (Remember that if the first check box is selected, the button will not be disabled after the first click.) In other words, you allow the first button click to be handled normally, but you trap and prevent any subsequent clicks by returning `false` from the event handler.

This is similar to the techniques used in the previous examples to trap a keypress event. You declare a page-level variable named `bButtonClicked` that is initially set to `false` (shown at the start of Listing 6.24). When a `click` event occurs, code in the `buttonClick` event handler tests to see if `bButtonClicked` is `false`. If it is, `bButtonClicked` is set to `true`, and the code starts the progress indicator timer and returns `true` from the function to allow the click to be processed by the browser.

If the button has already been clicked, `bButtonClicked` will be `true`, so the function can return `false` to prevent this `click` event from being processed. Finally, if the second check box is not selected, you start the timer and return `true` to allow the click to be processed.

Trapping the `submit` Event for the Form

Having seen how you can prevent multiple click events from being processed by using a page-level variable, you won't be surprised to see how the sample page prevents multiple submissions of a form. Listing 6.25 shows the `formSubmit` function, which is attached to the opening `<form>` element when the page is created (in the server-side `Page_Load` event), using the following:

```
frmMain.Attributes.Add("onsubmit", "return formSubmit(this);")
```

A page-level variable named `bFormSubmitted` is initially set to `false` and then switched to `true` when the form is first submitted. The progress indicator timer is also started at this point, and the function returns `true` to allow the form to be submitted. Subsequent attempts to submit the form fail because the function returns `false`. However, if the third check box is selected, the function always returns `true` to allow the form to be submitted multiple times—whereupon the final approach to handling multiple form submissions comes into play.

LISTING 6.25 The Client-Side `formSubmit` Function

```
var bFormSubmitted = false;

function formSubmit(ctrl) {
  // check value of third checkbox
  if(ctrl.elements['chkAllowSubmit'].checked == false) {
    // third checkbox is not ticked
    if (bFormSubmitted == false) {
      // first time form was submitted
      bFormSubmitted = true;
      startTimer();
      return true;
    }
    else {
      // prevent form from being submitted
      return false;
    }
  }
  else {
    // third checkbox is ticked
```

LISTING 6.25 Continued

```
    // allow form to be submitted
    startTimer();
    return true;
  }
}
```

Counting Postbacks with Server-Side Code

If all three check boxes are selected in the sample page, the user will be able to submit the form multiple times before the postback has completed and the page is reloaded into the browser. To prevent this from interrupting resource-intensive processing, you can use the final technique demonstrated by this example.

This technique involves counting postbacks. A counter variable is added to both the page and the user's session. When the page is created, the same value is placed into the viewstate of the page and stored in a session variable. Each time the page is posted back, the counter is incremented and the new value is placed in the viewstate and in the session.

However, if the user submits the same page more than once, the value in the viewstate will remain the same, whereas the value in the session variable will have been incremented when the initial postback from this instance of the page occurred. Figure 6.10 shows the process as a schematic diagram to make it easier to see how this works.

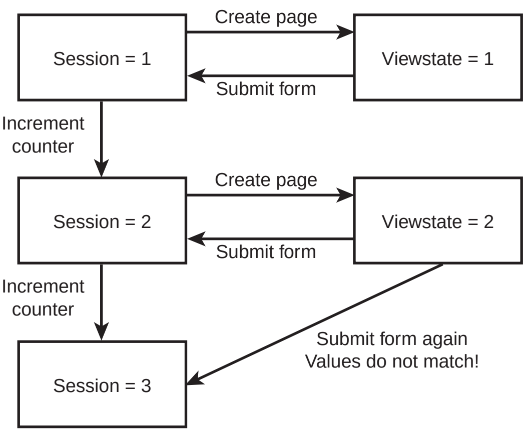


FIGURE 6.10 Counting postbacks to prevent multiple processing of the same page.

The `Page_Load` *Event Code for Counting Postbacks*

All the processing required to implement counting of postbacks is performed within the `Page_Load` event of the page, although you could attach it to server-side event handlers instead if required. Listing 6.26 shows the complete `Page_Load` event handler. After you add the client-side event handlers required for the previous techniques to the button and form elements on the page, you check to see if this is a postback or if the page is being loaded for the first time.

LISTING 6.26 The Page_Load Event Handler Code for Counting Postbacks

```
Sub Page_Load()

  ' add client-side event attributes to button and form here
  ...

  If Page.IsPostBack Then

    ' collect session and viewstate counter values
    Dim sPageLoads As String = Session("PageLoads")
    Dim sPageIndex As String = ViewState("PageIndex")

    If sPageLoads = "" Then
      lblMsg.Text &= "<b>WARNING:</b> Session support " _
        & "is not available."
    Else
      Dim iPageLoads As Integer = Integer.Parse(sPageLoads)
      Dim iPageIndex As Integer = Integer.Parse(sPageIndex)

      ' see if this is the first time the page was submitted
      If iPageLoads = iPageIndex Then
        lblMsg.Text &= "Thank you. Your input [" _
          & iPageLoads.ToString() & "] has been accepted."

        ' ************************************
        ' perform required page processing here
        ' ************************************

        ' delay execution of page before sending response
        ' page is buffered by default so no content is sent
        ' to the client until page is complete
        Dim dNext As DateTime = DateTime.Now
        dNext = dNext.AddSeconds(7)
        While DateTime.Compare(dNext, DateTime.Now) > 0
          ' wait for specified number of seconds
          ' to simulate long/complex page execution
        End While

      Else
        lblMsg.Text &= "<b>WARNING:</b> You clicked the button " _
          & (iPageLoads - iPageIndex + 1).ToString() & " times."
      End If

      ' increment counters for next page submission
      Session("PageLoads") = (iPageLoads + 1).ToString()
```

LISTING 6.26 Continued

```
      ViewState("PageIndex") = (iPageLoads + 1).ToString()

    End If
  Else

    ' preset counters when page first loads
    Session("PageLoads") = "1"
    ViewState("PageIndex") = "1"
    lblMsg.Text="Click the button to submit your information"

  End If
End Sub
```

If you look at the code at the end of Listing 6.26, you can see that when it's *not* a postback, you just set the viewstate and session values to `"1"` (remember that they are stored as `String` values). The viewstate of the page is a useful bag for storing small values. These values are encoded into the rest of the viewstate that ASP.NET automatically generates for the page it is creating.

If this is a postback, the first step is to check whether sessions are supported by looking for the value you stored against the `PageLoads` key when the page was initially created. The process will not work if there is no value in the session, and at this point, you need to decide what you want to do about it. If you absolutely need to perform the postback counting process, you can warn the user that he or she must enable sessions, or perhaps you would redirect the user to a page that uses ASP.NET cookieless sessions. You might even decide to use cookieless sessions for all clients.

> **Using a Hidden Control to Store Values**
>
> **An alternative approach would be to store the value in a `hidden`-type `input` control on the page. However, this is less secure than using the viewstate because the value can be viewed by users, who might be tempted to try to spoof the server by changing the value (although this is probably an unlikely scenario).**

> **Using Cookieless Sessions in ASP.NET**
>
> **The ASP.NET cookieless sessions feature provides session support for clients that do not themselves support HTTP cookies. It works by "munging" (that is, inserting) the session ID into the URL of the page and automatically updating all the hyperlinks in the page to reflect the updated URL. All you need to do to enable cookieless sessions is place in the root folder of the application a `web.config` file that contains the following:**
>
> ```
> <configuration>
>   <system.web>
>     <sessionState cookieless="true" />
>   </system.web>
> </configuration>
> ```

Comparing the Postback Counter Values

The next step in the process of checking for multiple postbacks is to compare the values in the viewstate and the session. If they are the same, you can accept the postback and start processing any submitted values. The sample page displays a message to indicate the current postback counter value. The code in the page uses a loop that waits seven seconds to simulate a

Trigger-Happy Button Clicks

Note that it's possible to click the button so quickly that ASP.NET does not have time to start processing the page and update the session value. In this case, the page reports fewer clicks than actually occurred when the final submit action has been processed.

long process. Afterward, you can increment the counter values in the viewstate and session, and then you can allow the page to be created and sent to the client.

However, if the viewstate and session values are different, you know that the postback has occurred from a page that you are already processing. Rather than try to cancel the existing processes that were started by previous postbacks from this instance of the page, you just ignore the current postback and don't carry out the processing again. Instead, you return a message to the user, indicating how many times he or she clicked the button. You can see the result in Figure 6.11.

FIGURE 6.11 The result in the one-click button example when the form is submitted more than once.

Summary

This chapter takes a more comprehensive look at how the client-side script used in the `ComboBox` control, described at the end of Chapter 5, works. It also discusses the three main requirements for producing interactive pages when using client-side script:

- Access to all the elements on the page, with the ability to read and set the content of each one, show or hide it, and generally manipulate it
- Access to a full range of keypress events, so that you can manage how a control behaves, depending on user interaction via the keyboard
- The ability to statically and dynamically position elements outside the flow model, using fixed (absolute) coordinates that are relative to a container

Following this discussion, the chapter delves deeper into integrating client-side code with ASP.NET server-side code to produce useful controls and interactive pages. This chapter considers four topics:

- Trapping an event that occurs on the client and popping up a confirmation dialog before carrying out the action on the server, by displaying a `confirmation` dialog before deleting a row in a `DataGrid` control.

- Trapping the Return key to prevent a form from being submitted, or in fact trapping any keypress that might not be suitable for a control or an application you are building.
- Handling individual keypress events, by implementing a `MaskedEdit` control.
- Creating a button that can be clicked only once, to prevent the user from causing a second postback when nothing seems to be happening at the client.

So, as you've seen, getting exactly the performance, appearance, or usability you want is not always easy (or even possible!). However, you can create components and build reusable content that far exceeds the standard output that ASP.NET can provide on its own. Chapter 7 continues this theme by looking at some more user controls that combine with the features of ASP.NET to make building interactive pages easier.

7

Design Issues for User Controls

Chapters 5, "Creating Reusable Content," and 6, "Client-Side Script Integration," look at some techniques for building reusable content for Web pages and Web applications and the advantages these techniques can provide. This chapter continues the theme by looking in detail at some more user controls. You'll see more useful ways that you can create different types of controls and provide functionality that is not available using the standard set of ASP.NET server controls and the HTML elements supported by the browser.

In Chapter 5, you built a combo box as a user control and learned about the basic issues involved. Then, in Chapter 6 you built a page that implements a `MaskedEdit` control.

In this chapter you'll see how you can convert that control into a user control. You'll also learn about another useful control—the `SpinBox` control.

User controls do not have to provide a user interface. In this chapter you'll also see a couple user controls that provide extra functionality for Web applications, but without actually creating elements in the browser.

IN THIS CHAPTER

Instead, they expose methods that make it easier to integrate dialogs and other client-side features with your ASP.NET code.

The final topic in this chapter is something that, to some extent, previous chapters glossed over: how to cope with different browser types. This chapter discusses some of the major issues and shows how to build controls that adapt to suit different browsers.

The Effect of User Controls on Design and Implementation

Converting sections of an ASP.NET page into a reusable user control is usually a reasonably simple task. HTML and text (content) work just the same way, as does any client-side script. And server controls declared in a user control produce the same visible output and work the same way, whether they're placed directly into an ASP.NET page or encapsulated in a user control.

The things that do change and that you need to bear in mind, are listed next. They may not all apply to the controls you build, but you'll see all these issues in this chapter:

- The position of the server controls within the hierarchy of the final ASP.NET page changes when the server controls are placed into a user control. The user control becomes a container, and its constituent server controls are located within the `Controls` collection of the user control. This changes the ID of the contained controls.
- User controls should support being used more than once within the same page, so they must avoid containing HTML or controls that can only appear once in the final ASP.NET page (for example, the `<html>`, `<head>`, and `<body>` elements, and the server-side `<form runat="server">` element).
- If you need client-side script to be injected into a page, you must be sure that only one instance of the script is created, regardless of how many user controls reside on the final ASP.NET page (unless each code section is specific to *that instance* of the user control).
- If your user control requires any images or other resources to be loaded from disk, you must decide how these will be referenced. For example, if an `Image` control within a user control uses `ImageUrl="myfile.gif"`, ASP.NET will expect the image to reside in the same folder as the user control. It will modify the path automatically, depending on the location of the page that hosts the user control.
- You need to consider whether to expose settings for the elements and behavior of a user control as properties rather than expecting people who use the user control to reference individual items within it. Exposing useful values as properties can make working with a user control a great deal simpler, and it allows you to validate values and perform other actions when property values are read or set.
- User controls can also expose methods, which can be functions that return values or just routines (for example, `Sub` in Visual Basic .NET, `void function` in C#) that do something within the control. You need to think about whether to allow the user to pass in the

values required for these methods as parameters or expect them to set any required values by using `Public` properties of your user control.

- If controls contained within your user control will have client-side event handlers attached, you must pass in all the values you need as parameters and not embed generated values within the client-side script unless they are the same for every instance of the user control. You'll see what we mean by this in more detail in the following section.
- If the contained controls raise events that you want to handle, you must handle these events within the user control. You cannot write event handlers in the hosting page for events exposed by server controls you declare within a user control.

Converting the `MaskedEdit` Control Page to a User Control

The `MaskedEdit` control example in Chapter 6 was written as an ASP.NET page (`maskedit.aspx`), although it uses a second ASP.NET page (`mask-image.aspx`) to generate the background image for the text box (see Figure 7.1).

FIGURE 7.1 The `MaskedEdit` user control sample page.

To create a user control that implements the `MaskedEdit` control, you just need to lift out the relevant code and declarations and place them into an `.ascx` file. Because the file implements a user control, it must start with a `Control` directive. You can turn on debugging during development to make it easier to see what's happening if an error occurs:

```
<%@Control Language="VB" Debug="True" %>
```

Then you can declare the user interface section. In this case, it's just an ordinary ASP.NET Web Forms `TextBox` control. You specify the default value for the columns and provide an ID so that you can refer to it in code within the user control:

```
<asp:TextBox id="txtMaskEdit" Columns="25" runat="server" />
```

Defining the User Control Interface

As you go through the process of converting content from an ASP.NET page into a user control, you must decide what properties and methods you want to expose from that user control. For this example, only two properties are exposed: a `String` value that defines the mask for the text box and the size of the font to use within the text box as an `Integer` value. Because you won't validate the values that are applied to the properties in this example, you can use the simplest approach and just declare them as `Public` variables, as shown here:

```
Public Mask As String
Public FontSize As Integer
```

The values effectively become *fields* of the user control that can be accessed from the hosting ASP.NET page.

You also need to declare one internal variable, which you will use to store the font name for the text box and the image you generate to represent the mask. You know that it must be a monospaced (fixed-pitch) font, so this example is limited to the Courier New font that is installed with Windows:

```
Private _font As String = "Courier New"
```

Notice that this (intentionally) small set of `Public` properties severely limits opportunities for users of the user control to affect how the control behaves. Users create an instance of the control (either declaratively or in code), but they cannot easily access the controls within it. For example, if you declare an instance of the `MaskedEdit` control like this:

```
<ahh:MaskEdit id="oCtrl" runat="server" />
```

you might be tempted to try to access the text box named `txtMaskEdit` within it (perhaps to change the number of columns), by using this:

```
oCtrl.txtMaskEdit.Columns = 100    ' produces a compiler error
```

User Controls Cannot Hide Their Content

Bear in mind that you can't encapsulate (and hide) controls and content in a user control as you can with a custom server control—like those you'll be meeting in Chapter 8, "Building Adaptive Controls." However, user controls are only plain-text files anyway, so developers who make use of a user control can always open it to see what's inside (and modify it as well, if they wish!).

This fails because the text box declared within the user control is generated as a `Protected` member of the control. The preceding code will result in the error "txtMaskEdit is not accessible in this context because it is 'Protected'." However, users can get around this by using the built-in `FindControl` method of the user control. This searches the `Controls` collection and returns the control with the matching ID value as a reference to the `Control` type. If you convert this into a `TextBox` type, the text box can be accessed:

```
CType(oCtrl.FindControl("txtMaskEdit"), TextBox).Columns = 100
```

This introduces an interesting point. If you or developers who use your control in their pages can access the controls it contains, do you need to expose properties that provide access to the controls? Maybe it's just as easy to allow developers to set the number of columns on a text box by using the technique just demonstrated.

In fact, that is probably not a good idea. It means that developers have to dig about inside the user control in a text editor to find the value of ID for the control they want to access and risk runtime errors through using the `FindControl` method (which cannot perform type checking at compile time). And if they are using the control in a development environment that provides IntelliSense or lists of properties and methods, only the `Public` interface members exposed by the control will be visible.

If you want to expose the constituent controls from a user control, you should do so as properties of the user control. For example, Listing 7.1 shows how you could expose the text box (which has the `id` attribute value `txtMaskEdit`) as a read-only property from the `MaskedEdit` user control.

LISTING 7.1 Exposing a Constituent Control from a User Control

```
Public ReadOnly Property Textbox As Textbox
  Get
    Return txtMaskEdit
  End Get
End Property
```

Users of the control can then access the text box and its properties in the usual way:

```
oCtrl.Textbox.Columns = 100
```

The issue now is that the users can set any properties they want on the control. In this case, specifying the number of columns, the font name, or the background image will effectively break the control. The only redeeming feature is that users are likely to make changes in the `Page_Load` event of the hosting page, which runs before the `Page_Load` event of the user control. Therefore, you can make sure that any specific properties that might break the control if set to inappropriate values are set back to suitable values in the `Page_Load` event of the user control.

The `Page_Load` Event Handler

Not surprisingly, most of the code used in the `MaskedEdit` page to create and set the attributes and properties of the controls just needs to be lifted out of the page and placed into the `Page_Load` event handler of the user control. This includes the code shown in Listing 7.2, which sets the `style` attributes for the text box, generates the correct format for the background mask image, and creates the ToolTip.

LISTING 7.2 The `Page_Load` Event Handler for the `MaskedEdit` User Control

```
Sub Page_Load()

  ' add style attributes to Textbox
  txtMaskEdit.Style("font-family") = _font
  txtMaskEdit.Style("font-size") = FontSize & "pt"

  ' create mask for display as Textbox background
```

LISTING 7.2 Continued

```
Dim sQuery As String = Mask
sQuery = sQuery.Replace("a", "_")
sQuery = sQuery.Replace("A", "_")
sQuery = sQuery.Replace("l", "_")
sQuery = sQuery.Replace("L", "_")
sQuery = sQuery.Replace("n", "_")
sQuery = sQuery.Replace("?", "_")

' encode it for query string to pass to page
' mask-image.aspx that generates the image
sQuery = Server.UrlEncode(sQuery)
_font = Server.UrlEncode(_font)

' create and add background style attribute
txtMaskEdit.Style("background-image") _
  = "url(mask-image.aspx?mask=" _
  & sQuery & "&font=" & _font _
  & "&size=" & FontSize & "&cols=" _
  & txtMaskEdit.Columns.ToString() & ")"

' create string to use as Tooltip for control
Dim sTip As String = Mask
sTip = sTip.Replace("a", "[a]")
sTip = sTip.Replace("A", "[A]")
sTip = sTip.Replace("l", "[l]")
sTip = sTip.Replace("L", "[L]")
sTip = sTip.Replace("n", "[n]")
sTip = sTip.Replace("?", "[?]")
txtMaskEdit.ToolTip = "Mask: " & sTip & vbCrlf & " where:" _
  & vbCrlf & "[a] = any alphanumeric character (0-9, A-z)" _
  & vbCrlf & "[A] = an uppercase alphanumeric character" _
  & vbCrlf & "[l] = any letter character (A-Z, a-z)" _
  & vbCrlf & "[L] = an uppercase letter character (A-Z)" _
  & vbCrlf & "[n] = any numeric character (0-9)" _
  & vbCrlf & "[?] = any character"
...
```

Injecting the Client-Side Code into the Page

One aspect of using client-side code within a user control deserves some serious rethinking when you develop reusable content such as the `MaskedEdit` control shown in this example. In previous examples, you've generated the client-side JavaScript code you need to make controls work by building it up as a string within the control.

This is perfectly valid, and it does encapsulate the code nicely. All you have to do is provide the `.ascx` file (or the custom server control, if that's how you implement the reusable content). It means that no other bits need to be installed in specific folders, and there's no need for any separate configuration settings.

However, it often makes sense to pool and reuse resources, as well as to separate them to make maintenance, debugging, and upgrades easier. For example, take a look at the ASP.NET validation controls. When the browser is Internet Explorer 5 or above, these server controls inject considerable amounts of JavaScript code into the page to handle client-side validation and display the error indicators next to controls without requiring the client to submit the page.

This JavaScript code runs to more than 400 lines and is common to all the validation controls. Rather than include all this code within every control, the ASP.NET installation routine places it into a separate file named `WebUIValidation.js`, within the special folder named `aspnet_client` in the root of all your Web sites. The `aspnet_client` folder contains a subfolder named `system.web`, and within that is a subfolder for each version of ASP.NET installed on the machine.

So, in version 1.1, the validation controls inject a `<script>` element into the page that specifies this file (in the folder `/aspnet_client/system_web/1_1_4322/`) rather than dumping all the JavaScript directly into the page:

```
<script language="javascript"
  src="/aspnet_client/system_web/1_1_4322/WebUIValidation.js">
</script>
```

This has other advantages besides reducing the size of the compiled server controls. It makes updating the JavaScript to cope with changes and updates to browsers it must support much easier. The browser also caches this code file the first time it loads a page that uses it, thus reducing subsequent download times for pages that take advantage of client-side validation.

There's no reason you can't use the same technique as the validation controls to expose client-side script code for your own user and server controls, although this example uses a new subfolder named `custom` within the `aspnet_client` folder to avoid confusion. The script file itself must contain complete JavaScript functions or sections of code but *not* the `<script>` and `</script>` tags. To capture this script for the `MaskedEdit` user control, you can simply display the page in the browser, select View, Source, and copy the code into a new text file saved with the `.js` file extension (the accepted extension for JavaScript; for VBScript files, you use `.vbs` instead).

Creating an `aspnet_client` Folder Manually

The `aspnet_client` folder and its contents are generated by the program `aspnet_regiis.exe`, which runs as part of the installation program for ASP.NET. However, the program only creates the `aspnet_client` folder within any existing Web sites. If you add a new site to IIS on your server, you must manually copy this folder to it. The folder also contains scripts for other features of ASP.NET, such as the `SmartNav.js` script for implementing smart navigation.

Using Script Files Across Multiple Applications

Recall that the scope rules of ASP.NET limit a user control to the same virtual application as the pages that host it. In other words, you can reference an `.ascx` user control from an `.aspx` page only if the user control is in a folder located within the same ASP.NET application. You can't share a single user control across multiple applications. However, some resources in a Web page are requested *directly by the client*—for example, the JavaScript `.js` files considered here. They can be loaded from any folder in any application, or even from a different Web site or a different machine.

Listing 7.3 shows how you now inject the client-side code you need into the page during the `Page_Load` event. Instead of creating a string containing all the code, you create a string that contains just the following:

```
<script language='javascript'
  src='/aspnet_client/custom/maskedit.js'>
</script>
```

Of course, you still use the `RegisterClientScriptBlock` and `IsClientScriptBlockRegistered` methods to make sure that this `<script>` element is injected into the page only once, for the first instance of the user control.

LISTING 7.3 Attaching Client-Side Event Handlers and Injecting Client-Side JavaScript Code into a Page

```
  ...
  ' see if previous instance of this control has already
  ' added the required JavaScript code reference to the page
  If Not Page.IsClientScriptBlockRegistered("AHHMaskEdit") Then
    Dim sPath As String = "/aspnet_client/custom/"
    Dim sScript As String = "<script language='javascript' " _
      & "src='" & sPath & "maskedit.js'><" & "/script>"
    ' add this JavaScript code to the page
    Page.RegisterClientScriptBlock("AHHMaskEdit", sScript)
  End If

  ' add client-side event handler attributes
  txtMaskEdit.Attributes.Add("onkeydown", _
             "return doKeyDown(event, this, '" & Mask & "')")
  txtMaskEdit.Attributes.Add("onkeypress", _
             "return doKeyPress(event, this, '" & Mask & "')")
  txtMaskEdit.Attributes.Add("onkeyup", _
             "return doKeyUp(event, this, '" & Mask & "')")
  txtMaskEdit.Attributes.Add("onfocus", _
             "return doFocus(event, this, '" & Mask & "')")

End Sub
```

Adding the Event Handler Attributes

The final task in working with the `Page_Load` event handler is to link the elements in the user control to the client-side script functions to make the control react to events as it is used. Recall from earlier in this chapter the issue regarding passing parameters into the client-side script.

If the client-side script were still declared within the control, as part of the output it generates, you might be tempted here to include the value of the mask directly within that code. You have the value stored in the `Mask` property at the moment, so you could use it as you create the client-side script string:

```
Dim sScript As String = ...
  & "var sMask = '" & Mask & "';" & vbCrlf _
  ...
```

Centralizing Images in the `aspnet_client` Folder

The `aspnet_client` folder can also be used to centralize any images that are required by user controls. For example, the `combo box` control described in previous chapters requires the up and down button images. In the sample control you created in Chapter 5, you stored these images in a folder within the same application as the user control; you could instead load them from any folder (or any site or server). So, for example, you could create a `control_images` folder within the `aspnet_client` folder and use it so that only one copy of each image is required for all your applications, and this image will be cached by the browser and reused every time.

This would be okay if the mask were the same for every instance of the `MaskedEdit` control that will use this script. Because there can be only one instance of the script on a page, all the `MaskedEdit` controls on a page would have to use the same value for the mask. This is obviously unnecessarily restrictive, so instead you pass the mask into each function that requires it as a parameter.

For example, the code in Listing 7.3 provides three parameters to the `doKeyDown` method described in Chapter 6. The signature of the function is as follows:

```
function doKeyDown(e, textbox, sMask)
```

The code in the `Page_Load` event attaches this to the `keydown` event of the text box, using the following:

```
txtMaskEdit.Attributes.Add("onkeydown", _
          "return doKeyDown(event, this, '" & Mask & "')")
```

The value of the `Mask` property can be different for each instance of the `MaskedEdit` user control, and each instance will pass its own value for the mask into the client-side function.

Adding Validation Controls to the `MaskedEdit` Control

Having completed the conversion of the `MaskedEdit` page into a user control, let's briefly consider the suggestion made in Chapter 6 for adding validation controls to it. One problem with the control has to do with limitations in the HTML `TextBox` control provided by the browser that mean you can't absolutely guarantee preventing the user from entering values that do not match the mask. In addition, an application may require the user to enter a value before the page can be submitted.

You can add validation controls to the text box within the user control quite easily, and it makes sense to do it this way because you already know what the mask is, so you can automatically generate the appropriate validation rules. Of course, this doesn't stop users from adding custom validation code themselves—either client-side code in the page or in their server-side code—but you can make the control easier to use by building validation into the control.

Listing 7.4 shows the declaration of the three validation controls added to the basic `MaskedEdit` control. The first prevents the page from being submitted if there is no value in the text box, and the second matches the value with a regular expression. Note that the regular expression is not specified (there is no `ValidationExpression` attribute within the declaration of the control). You'll be setting that at runtime in the `Page_Load` event handler.

The third control you add is a `ValidationSummary` control that displays the error messages from the other two controls when the user tries to submit an empty or invalid value. However, bear in mind that, when you build your own user and server controls, adding features like this might make the controls less useful or less flexible. Such features can also upset the layout of pages in which they are used.

Empty Values in the ASP.NET Validation Controls

Remember that the only validation control that detects an empty value is the `RequiredFieldValidator` control. The others intentionally treat empty controls as being valid. If they didn't work like this, the user would always have to fill in every control on the page. If you separate out the two tasks of validating a value that exists and preventing an empty value from being accepted, the controls can support validation in pages where some values are optional.

LISTING 7.4 Attaching a `RequiredFieldValidator` Control and a `RegularExpressionValidator` Control to the `MaskedEdit` Control

```
<asp:TextBox id="txtMaskEdit" Columns="25" runat="server" />
<asp:RequiredFieldValidator id="valRequired" runat="server"
     ControlToValidate="txtMaskEdit"
     ErrorMessage="* You must enter a value"
     Display="dynamic">
  *
</asp:RequiredFieldValidator>
<asp:RegularExpressionValidator id="valRegex" runat="server"
     ControlToValidate="txtMaskEdit"
     ErrorMessage="* Your entry does not match the mask"
     Display="dynamic">
  *
</asp:RegularExpressionValidator>
<asp:ValidationSummary id="valSummary" runat="server"
     HeaderText="<b>The following errors were found:</b>"
     ShowSummary="true" DisplayMode="List" />
```

For example, the `ValidationSummary` control, when visible, is implemented as a `<div>` element, and this might prevent the control from being properly positioned within the flow (inline layout) of other controls in the page. Or the user of the control might want to place the error messages elsewhere on the page or create his or her own messages. The user might even want to be able to turn client-side validation on and off, allow empty values, customize the error messages, and so on.

Before long, you might end up implementing a long list of properties in your user control to allow this kind of configuration. In fact, it might even be easier just to expose the validation controls as properties from your user control; you saw how to do this in Listing 7.1.

Creating the Validation Expression

The only other task related to adding the validation controls to the sample user control is to build the appropriate regular expression for the `RegularExpressionValidator` control. Regular expressions are a complex topic, and some aficionados like to make them appear even more complicated than they actually are. However, you can use very simple constructs to build the regular expression for this example.

Regular expressions use the forward slash character to signify characters that have a special meaning (sometimes called *metacharacters*); for example, `\d` means the digits 0 to 9. So the first step is to replace any instances of the `\` character in your mask with the sequence `\\`, to prevent what follows from being treated as a special character.

With regular expressions, you can identify characters as sequential sets by specifying the first and last value, enclosed in square brackets. You can combine sets by using a comma, so the sequence `[A-Z,a-z,0-9]` will give a match to the character at the current position in the target string if it is an upper- or lowercase letter or a digit.

So you can see how you build the regular expression you need, without resorting to any of the many metacharacters that are available. Listing 7.5 shows the code you add to the `Page_Load` event of the user control to create the regular expression and assign it to the `ValidationExpression` property of the `RegularExpressionValidator` control.

The Regular Expression Party Game

A party trick I've seen demonstrated (although thankfully not at all the parties I attend) is to produce the shortest regular expression possible that matches or modifies a specific mask or string, without using pen and paper. Regular expressions are extremely powerful, can be used to produce modified versions of a string, and can save a lot of code in certain situations. The concept of regular expressions might not be the easiest of topics to grasp, but it is definitely worth adding to your "I must learn more about..." list if you are not familiar with it already.

LISTING 7.5 Creating a Regular Expression for a Validation Control in the `Page_Load` Event Handler

```
' create regular expression for validation control
Dim sRegex As String = Mask
sRegex = sRegex.Replace("\", "\\")
sRegex = sRegex.Replace("a", "[A-Z,a-z,0-9]")
sRegex = sRegex.Replace("A", "[A-Z,0-9]")
sRegex = sRegex.Replace("l", "[A-Z,a-z]")
```

LISTING 7.5 Continued

```
sRegex = sRegex.Replace("L", "[A-Z]")
sRegex = sRegex.Replace("n", "[0-9]")
sRegex = sRegex.Replace("?", ".")
valRegex.ValidationExpression = sRegex
```

The result of trying to submit a page with a partially completed value in the control is shown in Figure 7.2. You can see the asterisk that the client-side validation script displays as soon as focus moves from the control, and you can also see the output generated by the `ValidationSummary` control underneath the text box.

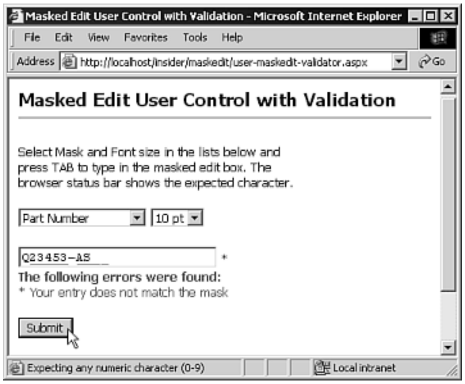


FIGURE 7.2 The `MaskedEdit` user control, with validation sample page.

Building a `SpinBox` User Control

A third control that complements the controls available in a normal Web browser is the `SpinBox` control. This useful control makes it easy for users to enter numeric values, either by typing them into a text box or by changing the existing value with the up and down buttons located at the end of the text box. Users can also change the value by pressing the up, left, right, and down arrow keys or the Home and End keys. A page that demonstrates the `SpinBox` control is shown in Figure 7.3.

The example in this chapter is implemented as a user control, just like the `ComboBox` and `MaskedEdit` controls you've worked with previously. Therefore, much of the code and many of the techniques are similar. However, we'll discuss the particularly interesting points of the code in more depth in the following sections. The specific points of interest are:

- Implementing `AutoPostback` so that the control behaves like a standard ASP.NET Web Forms control
- Ensuring that the value within the control is always valid when a page is submitted
- Ensuring that values provided for properties of the control are valid and deciding what to do if they are not
- Raising an exception when something goes wrong

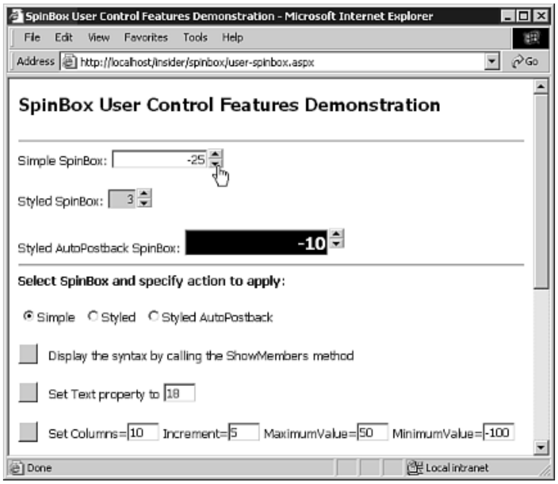


FIGURE 7.3 The `SpinBox` control demonstration page.

First, however, you'll see the HTML and control declarations that are used to generate the user interface.

The User Interface Declaration for the `SpinBox` Control

The `SpinBox` control (`user-spinbox.ascx`) uses the same technique as the `ComboBox` control you built in Chapter 5 to position the elements it requires. A `<span>` element with the style selector `position:relative` forms the container, and within this you place an ASP.NET `TextBox` control and two `ImageButton` controls. The `ImageButton` controls use `position:absolute` and have the `top` selector set so that they will be correctly positioned vertically in relationship to the text box (see Listing 7.6).

LISTING 7.6 The Declaration of the Constituent Controls for the `SpinBox` User Control

```
<span id="spindiv" Style="position:relative" runat="server">
<asp:TextBox Style="top:0;left:0;text-align:right" id="textbox"
     runat="server"/>
<asp:ImageButton id="imageup" Style="position:absolute;top:0"
     ImageUrl="~/images/spin-up.gif" runat="server" />
<asp:ImageButton id="imagedown" Style="position:absolute;top:10"
     ImageUrl="~/images/spin-down.gif" runat="server" />
</span>
```

Of course, at this point you don't know how wide the text box will be, so you can't set the `left` selector for the `ImageButton` controls. This is done in the `Page_Load` event, together with the specification of the text box width, using a value that is calculated from the property settings specified by the hosting page.

Notice that, as with the `ComboBox` control, you use the tilde (~) character here to specify that the images for the `ImageButton` controls reside in a subfolder named `images` that is located within the root of the current application. You could instead specify a machinewide location (such as aspnet_client, as intimated in the sidebar “Centralizing Images in the aspnet_client Folder,” earlier in this chapter).

The Private and Public Members of the Control

The SpinBox control uses four Private internal variables (see Listing 7.7) to hold values assigned to properties of the control. It also exposes a ShowMembers method in the same way as the ComboBox control example in Chapter 6.

LISTING 7.7 The Internal Variables and the ShowMembers Method

```
Private _columns As Integer = 3
Private _increment As Integer = 1
Private _maxvalue As Integer = 99
Private _minvalue As Integer = 0

Public Function ShowMembers() As String
  Dim sResult As String = "<b>SpinBox User Control</b>" _
    & "</p><b>Properties:</b><br />" _
    & "AutoPostback (Boolean, default False)<br />" _
    & "CssClass (String)<br />" _
    & "Columns (Integer, default 3)<br />" _
    & "Increment (Integer, default 1)<br />" _
    & "MaximumValue (Integer, default 99)<br />" _
    & "MinimumValue (Integer, default 0)<br />" _
    & "Text (String)<br />" _
    & "Value (Integer)<br />"
  Return sResult
End Function
```

The Property Fields and Accessor Routines

The properties of the SpinBox control are declared next. You declare two of them as fields by using Public variables because you don’t need to perform any validation of their values when they are set or read. The first of these is a String property that can be used to specify the CSS style class for the text box within the control. The second is a Boolean property that is used to indicate whether AutoPostback is required:

```
Public CssClass As String = ""
Public AutoPostback As Boolean = False
```

You want the control to behave like other Web Forms controls in that the user should be able to choose whether to force it to post back to the server every time the value is changed (AutoPostback = True) or allow repeated interaction with it without a postback occurring (AutoPostback = False).

Regarding clicking the ImageButton controls (which are implemented in the browser as <input type="image"> elements), this is easy. You just have to trap the click event on the client and return false from this event handler to prevent the page from being submitted. To implement AutoPostback, you return true from these event handlers.

However, the issue is not quite as obvious where the text box is concerned. If you set the built-in AutoPostback property to True for a standard ASP.NET TextBox control, the page will be posted back to the server automatically when the text box loses the input focus. (ASP.NET does this by injecting client-side script into the page to handle the blur event.)

However, you want to use the blur event to validate the text in the text box section of the control to ensure that it represents a valid Integer value that is within the range of the current maximum and minimum values. It's also likely that users will type in the text box and then interact with the up and down buttons. This would cause two postback events—one when the text box loses the focus and one for the click on the button.

So you do not set the built-in AutoPostback property of the TextBox control to True, even if the AutoPostback property of the user control is set to True. This is a good example of how you often need to carefully consider how a user will interact with a compound control like the SpinBox control when you implement properties for it.

Implementing Behavior and Appearance Properties for the SpinBox Control

Four properties specify the behavior and appearance of the SpinBox control. The Columns property specifies the width of the text box within the control, in the same way that it is used to specify the width of a normal ASP.NET TextBox control. The value is of type Integer, approximately representing the number of characters that will be visible in the text box.

The three properties that specify the behavior of the control are Increment, MaximumValue, and MinimumValue. It should be obvious what these do; the only things worth pointing out here are that the maximum and minimum values are of type Integer and are inclusive (the control can be set to the maximum or the minimum value) and that the increment must be a positive Integer value.

Listing 7.8 shows the declaration of the properties. All four are read/write, and the Get section simply returns the value of the matching internal variable. Because these internal variables all have default values specified (refer to Listing 7.7), you can use the control without setting these properties, and the default values will be available if these properties are read without first being set.

LISTING 7.8 The Behavior and Appearance Property Declarations

```
Public Property Columns As Integer
  Get
    Return _columns
  End Get
  Set
    If (value > 0) And (value < 1000) Then
      _columns = value
    Else
```

LISTING 7.8 Continued

```
      Throw New Exception("Columns must be between 1 and 999")
    End If
  End Set
End Property

Public Property Increment As Integer
  Get
    Return _increment
  End Get
  Set
    If value > 0 Then
      _increment = value
    Else
      Throw New Exception("Increment must be greater than zero")
    End If
  End Set
End Property

Public Property MaximumValue As Integer
  Get
    Return _maxvalue
  End Get
  Set
    If value > _minvalue Then
      _maxvalue = value
    Else
      Throw New Exception("MaximumValue must be greater " _
                          & "than current MinimumValue")
    End If
  End Set
End Property

Public Property MinimumValue As Integer
  Get
    Return _minvalue
  End Get
  Set
    If value < _maxvalue Then
      _minvalue = value
    Else
      Throw New Exception("MinimumValue must be less " _
                          & "than current MaximumValue")
    End If
  End Set
End Property
```

Raising an Exception for Invalid Property Settings

When you create the property accessors for controls, you'll often want to perform some validation of the values that are applied to these properties. This can prevent exceptions from being raised within a control if users set inappropriate values, and it can ensure that the behavior of the control is predictable.

In the `SpinBox` control, you make one design decision by limiting the number of columns for the text box to fewer than 1,000. It seems extremely unlikely that the user would want more than this, and if more columns were allowed, the resulting page would most likely be too wide to display anyway. You also force the number of columns to be greater than 0 because otherwise the control won't be visible. Bear in mind that you should try to avoid applying design limitations that will limit the usefulness of a control.

The other type of decision regarding property values is based on practicality. For example, you make sure that the minimum value can only be set if the new value is less than the current maximum value and vice versa. You also make sure that the increment is greater than 0 (otherwise, the up and down buttons would work the wrong way around).

Of course, practicality decisions should also involve the prevention of errors. You prevent the `Columns` property from being 0 or greater than 1,000 for basically cosmetic and usability reasons, but you must prevent it from being *less than* 0 as well, or you'll get a runtime error when you try to apply the value to the `TextBox` control.

So what do you do if the user specifies an invalid value? With the `ComboBox` control in Chapter 5, you faced a similar issue with properties such as `SelectedIndex` and `SelectedValue`. In those two cases, you simply ignored the value if it was out of range. For example, if the user set the `SelectedIndex` property to a value less than `-1` or greater than the index of the last item in the list, you just ignored the setting and left the current selection unchanged. If the user specified a value for the `SelectedValue` property that was not in the list, you just ignored it. However, this is not the way most of the ASP.NET controls work. If you specify a `SelectedValue` property value that is not in the list for a `ListBox` control, for example, an `ArgumentOutOfRangeException` error is thrown.

> **Validating Input Values for Methods and Properties**
>
> **Validating input and raising appropriate exceptions is a necessity when you are exposing methods from controls, as well as in your property accessors. You really should make sure that your code is protected from invalid parameter values.**

In the `SpinBox` control, you follow the same approach as the `standard ASP.NET server controls`. If the user specifies an invalid value for any of the four properties we've just examined (`Columns`, `Increment`, `MaximumValue`, and `MinimumValue`), you create a new `Exception` instance that contains a description of the error and throw it back to the calling routine. There, the text description can be extracted from the `Message` property of the exception.

> **Creating a Specific Exception Type**
>
> **You could create instances of more specific types of `Exception`, such as `ArgumentOutOfRangeException`, and you might prefer to do this with your controls. This approach allows the hosting page to catch the exceptions by type and handle the different types in different ways.**

Implementing the `Text` and `Value` Properties

The two remaining properties of the `SpinBox` user control are `Text` and `Value`. The only real difference between them is in the data type they accept and return. It seems intuitive to offer the value of a control aimed at collecting numeric whole-number values as an `Integer` property, yet the accepted property name for the value of a text box is the `Text` property. So, in line with the typical programmer's capability for indecision, the sample control implements both.

Listing 7.9 shows these property declarations. You just return the `Text` property of the text box within the user control in the `Get` sections. The code attempts to convert it to an `Integer` type for the `Value` property, and this will automatically return `0` if the text is not a valid representation of a number.

When the `Text` or `Value` property is set, you make sure that the new value is within the current maximum and minimum values. In the case of the `Text` property, you also have to check that the value provided represents a valid Integer type.

LISTING 7.9 The `Text` and `Value` Property Declarations

```
Public Property Text As String
  Get
    Return textbox.Text
  End Get
  Set
    Dim iValue As Integer
    Try
      iValue = Int32.Parse(value)
    Catch
      Throw New Exception("Text property must represent " _
                        & "a valid Integer value")
    End Try
    If (value >= _minvalue) And (value <= _maxvalue)
      textbox.Text = value
    Else
      Throw New Exception("Text property must be within " _
            & "the current MinimumValue and MaximumValue")
    End If
  End Set
End Property

Public Property Value As Integer
  Get
    Try
      Return Int32.Parse(textbox.Text)
    Catch
    End Try
  End Get
  Set
```

LISTING 7.9 Continued

```
    If (value >= _minvalue) And (value <= _maxvalue)
      textbox.Text = value.ToString()
    Else
      Throw New Exception("Value property must be within " _
            & "the current MinimumValue and MaximumValue")
    End If
  End Set
End Property
```

The Server-Side Code Within the SpinBox Control

Other than the property accessors you've just seen, there is very little code in the remainder of the `SpinBox` control. There is a `Page_Load` event handler, which we'll discuss shortly, and there are a couple auxiliary routines that are used to set features of the control and make sure that the current value is within the maximum and minimum values set in the control. Listing 7.10 shows these two auxiliary routines.

LISTING 7.10 The `SetColumns` and `SetMaxMinValues` Routines

```
' set width of Textbox and position images
Private Sub SetColumns()
  textbox.Columns = _columns
  textbox.Style("width") = Columns * 10
  imageup.Style("left") = textbox.Style("width")
  imagedown.Style("left") = textbox.Style("width")
End Sub

' check if current value of Textbox is within
' current max and min limits, and reset if not
Private Sub SetMaxMinValues()
  Dim iValue As Integer
  Try
    iValue = Int32.Parse(textbox.Text)
  Catch
    iValue = _minvalue
  End Try
  If iValue < _minvalue Then
    iValue = _minvalue
  End If
  If iValue > _maxvalue Then
    iValue = _maxvalue
  End If
  textbox.Text = iValue.ToString()
End Sub
```

Setting the TextBox Control Width and Positioning the Images

When you originally implemented the `SpinBox` control, an interesting issue came to light about setting the size of the control. The `ComboBox` control from Chapter 5 exposes a property named `Width`, which is used to set the size of the control, in pixels. However, to match the properties of the ASP.NET `TextBox` control, you expose a property named `Columns` for the `SpinBox` control.

But how do you relate the width of the text box with the setting of the `Columns` property? The browser uses the current font style and size to work out how wide to make the text box (and often doesn't do so very accurately—try creating a `TextBox` control with `Columns="3"`, and you'll probably find that there is room to type five or six characters).

> **Setting the Width of a Text Box**
>
> **The user of the control will probably just fiddle with the `Columns` value until it seems right for the page where it's used, so you need to ask yourself whether it is actually worthwhile to spend a lot of time and effort on calculating the width. One other approach would be to use the current maximum and minimum values to work out how wide it should be, taking into account the font style and size. But then, of course, you would need to be convinced that the user will appreciate the control changing its size every time the user changes the maximum and minimum values.**

You need to specify the exact size, in pixels, so that you can accurately locate the up and down buttons at the end of the text box. Experimentation reveals that simply multiplying the value of `Columns` by 10 gives a generally similar width in pixels, although you might want to substitute a more realistic calculation here. When you know the width, you can apply it to the text box and also use it to set the left position of the two `ImageButton` controls. Notice that you set the `Columns` property of the `TextBox` control as well, although that will not actually affect the width of the text box if the browser supports CSS2.

Checking the MaximumValue and MinimumValue Properties

The second routine shown in Listing 7.10 is used to verify that the current value in the text box of the control is within the maximum and minimum values. If it's outside these values, you simply set it to the current maximum or minimum value, depending on which is closest. You use this routine in the `Page_Load` event of the control so that it validates the value each time the hosting page loads.

The Page_Load Event Handler

Let's now look at the `Page_Load` event handler, shown in full in Listing 7.11. There's nothing surprising here: You just collect the ID of the user control from the `UniqueID` property, for use when connecting the client-side event handlers. You also check whether `AutoPostback` is turned on and create an appropriate `String` value (`"true"` or `"false"`) to use when creating the parameter string for the event handlers attached to the two `ImageButton` controls. From all these values, you can then create a `String` value that represents the complete set of parameters for each client-side event call.

LISTING 7.11 The Page_Load Event Handler for the SpinBox Control

```
Sub Page_Load()

  ' control ID prefix for contained controls
  Dim sCID As String = Me.UniqueID & "_"

  ' create true/false string for JavaScript code
  Dim sAutoPostback As String = "false"
  If AutoPostback Then
    sAutoPostback = "true"
  End If

  ' create JavaScript parameter string - used to set
  ' parameters for client-side control event handlers
  Dim sParams As String = "'" & sCID & "textbox', " _
    & _minvalue.ToString() & ", " _
    & _maxvalue.ToString() & ", " _
    & _increment.ToString() & ", " _
    & sAutoPostback

  ' see if previous instance of this control has already
  ' added the required JavaScript code to the page
  If Not Page.IsClientScriptBlockRegistered("AHHSpinBox") Then
    Dim sPath As String = "/aspnet_client/custom/"
    Dim sScript As String = "<script language='javascript' " _
      & "src='" & sPath & "spinbox.js'><" & "/script>"
    ' add this JavaScript code to the page
    ' add this JavaScript code to the page
    Page.RegisterClientScriptBlock("AHHSpinBox", sScript)

  End If

  If CssClass <> "" Then
    textbox.CssClass = CssClass
  End If

  ' set client-side event handlers for controls
  imageup.Attributes.Add("onclick", _
          "return incrementValue(" & sParams & ")")
  imagedown.Attributes.Add("onclick", _
          "return decrementValue(" & sParams & ")")
  textbox.Attributes.Add("onblur", _
          "return checkValue(" & sParams & ")")
  textbox.Attributes.Add("onkeydown", _
          "return keyDown(event, " & sParams & ")")
```

LISTING 7.11 Continued

```
  SetColumns()
  SetMaxMinValues()

End Sub
```

Using the Same Parameter Lists for All Functions

You'll see later in this chapter that you don't actually need to provide the `bAutoPost` parameter for one of the functions, but it makes no difference if you do. You simply ignore it in the client-side function, and then you can use the same parameter string that you generate here for all the event handlers.

The next step is to inject a `<script>` element into the output that will reference and load a file named `spinbox.js` from the same `/aspnet_client/custom` folder you used with the `MaskedEdit` control earlier in this chapter. You'll see this client-side script file a little later in this chapter.

Then, if the user of the control has specified a value for the `CssClass` property, you can add that to the text box, and then you can attach the event handlers to the `TextBox` and `ImageButton` controls. You finish up with a call to the two auxiliary routines described earlier in this chapter. The `SetColumns` routine adds the style attributes and properties to the constituent controls to specify their width and position (replacing any conflicting settings applied by the `CssClass` property value), and the `SetMaxMinValues` routine ensures that the text box value is within the prescribed range.

The Client-Side Script Code

As you can see in Listing 7.11, you are handling four client-side events—the `click` event for the two `ImageButton` controls and the `blur` and `keydown` events for the `TextBox` control. Each event calls a separate function in the client-side script and passes in the five parameters defined in the `sParams` variable earlier in the listing. These are the five parameters:

- The full ID of the `<input type="text">` control that is generated by the ASP.NET `TextBox` control (for example, `"MyUserControl_textbox"`)
- The current minimum value, as set in the `MinimumValue` property of the control
- The current maximum value, as set in the `MaximumValue` property of the control
- The current value of the increment for each button click, as set in the `Increment` property of the control
- The value `true` or `false`, reflecting the setting of the `AutoPostback` property of the control

Listing 7.12 shows the complete contents of the `spinbox.js` file that the `SpinBox` user control loads through the `<script>` element injected into the page. The `incrementValue` and `decrementValue` functions are similar to each other, simply incrementing or decrementing the contents of the text box by the value of the increment passed in as the `iInc` parameter. However, they also check that the value of the control is a valid number, and if it is not, they set it to the current minimum value. If the increment or decrement takes it beyond the current valid range, they set it to the current maximum (`iMaxVal`) or minimum (`iMinVal`) value.

LISTING 7.12 The Client-Side Script Functions for the SpinBox User Control

```
function incrementValue(sTextID, iMinVal, iMaxVal, iInc, bAutoPost) {
  var textbox = document.getElementById(sTextID);
  var textval = parseInt(textbox.value);
  if (isNaN(textval) || textval < iMinVal)
    textval = iMinVal;
  else {
    if (textval < (iMaxVal - iInc))
      textval += iInc;
    else
      textval = iMaxVal;
  }
  textbox.value = textval.toString();
  return bAutoPost;
}

function decrementValue(sTextID, iMinVal, iMaxVal, iInc, bAutoPost) {
  var textbox = document.getElementById(sTextID);
  var textval = parseInt(textbox.value);
  if (isNaN(textval) || textval < iMinVal)
    textval = iMinVal;
  else {
    if (textval > (iMinVal + iInc))
      textval -= iInc;
    else
      textval = iMinVal;
  }
  textbox.value = textval.toString();
  return bAutoPost;
}

function checkValue(sTextID, iMinVal, iMaxVal, iInc, bAutoPost) {
  var textbox = document.getElementById(sTextID);
  var textval = parseInt(textbox.value);
  if (isNaN(textval) || textval < iMinVal)
    textval = iMinVal;
  if (textval > iMaxVal)
    textval = iMaxVal;
  textbox.value = textval.toString();
  return false;
}

function keyDown(e, sTextID, iMinVal, iMaxVal, iInc, bAutoPost) {
  var textbox = document.getElementById(sTextID);
  var iKeyCode = 0;
```

LISTING 7.12 Continued

```
  if (window.event) iKeyCode = window.event.keyCode
  else {
    if (e) iKeyCode = e.which;
  }
  if (iKeyCode == 38)
    incrementValue(sTextID, iMinVal, iMaxVal, iInc, bAutoPost);
  if (iKeyCode == 40)
    decrementValue(sTextID, iMinVal, iMaxVal, iInc, bAutoPost);
  if (iKeyCode == 37 ¦¦ iKeyCode == 36)
    textbox.value = iMinVal.toString();
  if (iKeyCode == 39 ¦¦ iKeyCode == 35)
    textbox.value = iMaxVal.toString();
  return true;
}
```

These two functions are called when the client clicks the up button or the down button. Because these are `<input type="image">` controls, they will cause a postback to the server unless you return the value `false` from these functions. By returning the value of the `bAutoPost` parameter (which is set to either `true` or `false`, depending on the setting of the `AutoPostback` property), you can allow or prevent the postback, as required.

The `checkValue` function runs when the `blur` event occurs for the text box, usually when the user clicks a different control in the page (including one of the up or down buttons). All this function does is ensure that the value in the text box represents a valid number and is within the range specified. If it is not, the `checkValue` function sets it to the minimum or maximum value, depending on which is closer.

The fourth function, `keyDown`, runs when the text box has the input focus and the user presses a key. It detects the four arrow keys and the Home and End keys, using their key code values, and changes the value in the text box accordingly. The up and down arrows change it by the current increment by simply calling the `incrementValue` and `decrementValue` functions with the same set of parameters. The left and right arrows, and the Home and End keys, set it to the minimum value or the maximum value.

> **Trapping Keypresses to Prevent Errors**
>
> **You could use the `keyDown` event to trap nonnumeric characters and prevent them from being typed into the text box. However, you would then have to be sure to properly handle the Delete key, Backspace key, and other keys as well, to allow the user to react with the text box in the usual way. Because you've provided plenty of protection against allowing invalid values to be posted to the server, this is unnecessary, but you could easily add it if required. Chapter 6 provides examples of handling keypress events and detecting key code values with the `MaskedEdit` control.**

Integrating Client-Side Script Dialogs

Although we're continuing with our theme of user controls, let's now change direction a little to look at a couple sample user controls that don't provide any user interface at all. What's the point of building such a control? This kind of control allows you to reuse code (such as functions or methods that are defined within the user control) or inject other types of nonvisible output into the page.

One of the things that regularly confuses new users of ASP.NET, especially those who are used to building traditional client-based executable applications, is that they can't just pop up a message box to ask the user to confirm an action or to let the user know something is going to happen. However, this isn't impossible to do, as you saw in Chapter 6, where you used a JavaScript `confirm` dialog to make sure that the user wanted to delete rows from a database table. However, all this really does is prevent the page from being submitted by returning `false` to the control that raised the event on the client if the user clicks the Cancel button.

What happens if you want to ask the user a question and then access the reply on the server in ASP.NET code? The easy answer is that you let the user submit a separate page containing HTML controls, but often it's nicer (and more intuitive) to use a JavaScript `alert`, `confirm`, or `prompt` dialog.

Figure 7.4 shows the demonstration page provided with the samples for this book. An `alert` dialog is attached to the first button on the page (a simple `<input type="submit">` button), so the dialog is displayed to the user when he or she clicks this button, before the form is submitted.

Using the VBScript `MsgBox` Function

It would be even nicer to be able to offer one of the more attractive and configurable VBScript `MsgBox` dialogs if you know that the browser is Internet Explorer. However, in this case, the client-side script will be written in VBScript (not JavaScript) and it would therefore not work in other browsers. The solution is to sniff the browser type on the server and insert a different `<script>` section into the page, depending on the browser type. You'll see how you can build *adaptive controls* like this later in this chapter and throughout Chapter 8.

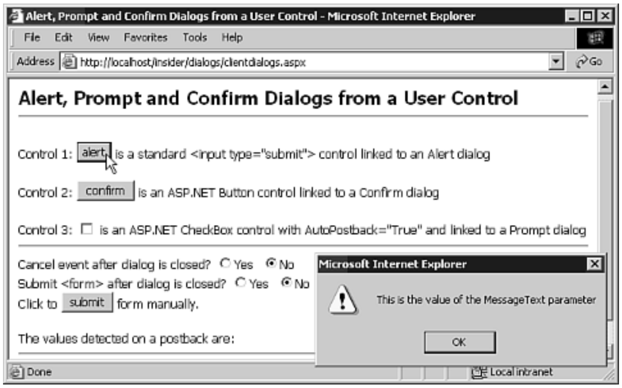


FIGURE 7.4
Displaying an `alert` dialog in response to a button click.

The second button (this one is an ASP.NET `Button` control) is connected to a `confirm` dialog, as shown in Figure 7.5. After the user clicks a button in the `confirm` dialog, the page is posted back to the server. You can see that the value detected on the server during the postback is displayed at the bottom of the page. This means you can write server-side code that reacts in different ways, depending on the user's response in the dialog.

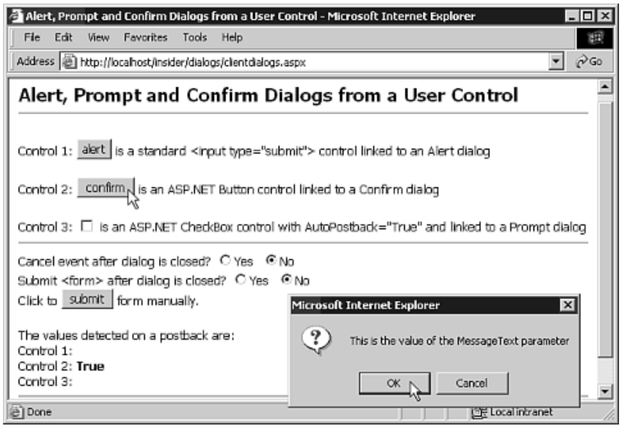


FIGURE 7.5
Detecting the value selected by the user in a `confirm` dialog.

Finally, the check box in the sample page has a `prompt` dialog attached to its client-side `click` event. Changing the setting of the check box opens the `prompt` dialog and allows the user to enter a value. After the dialog closes and the form is submitted, the value the user entered is detected on the server and displayed in the page that is returned (see Figure 7.6). Notice that the addition of the dialog to this control prevents `AutoPostback` from working in the usual way, so you have to use the Submit button to submit the page.

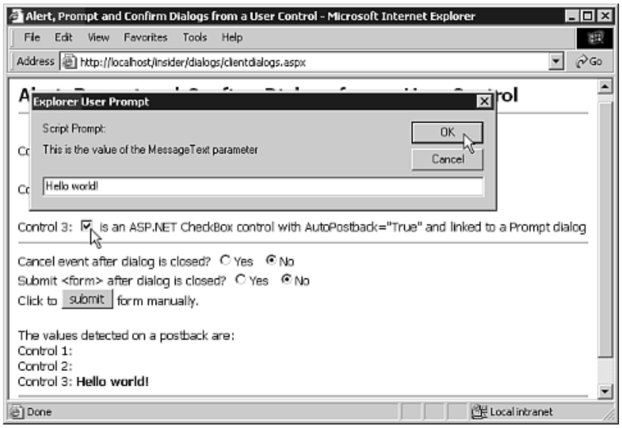


FIGURE 7.6
Detecting the value entered by the user into a prompt dialog.

However, you can use the two sets of option buttons shown in Figures 7.4 through 7.6 to change the behavior of the page. You can instruct the client-side code that displays the dialogs to cancel any action that the control they are attached to would usually initiate (for example, you can prevent the postback from a button to which they are attached). You can also force the code to automatically submit the page after the dialog is closed. Setting this option forces the page to be submitted when the `prompt` dialog attached to the check box is closed.

How the Client Dialogs Example Works

Obviously, the action of opening the dialogs uses client-side script. However, what's useful here is that the return value from each dialog is passed back to the server the next time the page is submitted. This is the case because you place the return value in a `hidden`-type control on the form. Even if the user interacts with other controls before the form is submitted, the value will remain in the hidden control. The only time it will change is if the user reopens the dialog and makes a different selection or enters a different value.

This is what the ASP.NET code is actually doing when a dialog is attached to a control:

- Injecting into the page a hidden-type control (an `<input type="hidden">` element) that has a unique name and ID
- Injecting into the page some client-side script that will handle the specified client-side event of the control (such as the `click` event of a button)
- Building up the function name and parameters for the appropriate function within this client-side script and assigning it to the appropriate event attribute of the control

Then, when the user fires the event (for example, by clicking a button), the client-side script displays the appropriate dialog. When the dialog is closed, the function places the return value into the hidden control and then—depending on the parameters provided by the user when attaching the dialog—allows or prevents the form from being submitted or forces it to be submitted.

On the server, following this postback, the value is extracted from the hidden control and displayed in the page. Of course, in a real application, you would use the value within your server-side code as required.

The `clientdialog.ascx` User Control

As you no doubt expect, the code just described is packaged up into a user control for reuse in any pages that require it. It's relatively simple, and it demonstrates the features of user controls discussed earlier in this chapter. It provides no user interface at all, but instead it exposes two methods that can be called from the hosting page to perform the magic you've just seen:

- The `AttachDialog` method takes as parameters all the information required to display the dialog, including the ID and event of the control to which the dialog will be attached and the text to display. There are also optional parameters that allow you to specify whether the underlying control event should be canceled and whether the form should be automatically submitted when the dialog is closed.
- The `GetDialogResult` method takes a single parameter—the ID of the control to which the dialog was attached—and returns the value as a `String` data type. In the case of a `confirm` dialog, the returned value will be one of the strings `"True"` or `"False"`.

The `DialogType` Enumeration

The `clientdialog.ascx` page also exposes an enumeration of the dialog types. Using enumerations is a useful way to define a closed set of values from which the user must choose one. You can use enumerations to ensure that the user of the control can set only one of the specified values for a property or—as you'll see in this example—for a parameter to a method. The `DialogType` enumeration is declared in the `clientdialog.aspx` page, as shown in Listing 7.13.

LISTING 7.13 The `DialogType` Enumeration Exposed by the User Control

```
' enumeration of dialog types
Public Enum DialogType
  Alert = 0
  Confirm = 1
  Prompt = 2
End Enum
```

Because the enumeration is defined as being `Public`, you can also reference it in the page that hosts the user control. For example, assuming that the user control has the ID value `oCtrl`, you can create an `Integer` variable that equates to the `Confirm` value of the enumeration by using this:

```
Dim iValue As Integer = oCtrl.DialogType.Confirm
```

The `AttachDialog` Method

Listing 7.14 shows the `AttachDialog` method. This is a subroutine because no return value is required. You can see the parameters that the function takes, including the dialog type (as a value from the `DialogType` enumeration just declared), the ID value of the control, the name of the client-side event to which you'll attach the dialog, the text to display in the dialog, and the two optional parameters that control postbacks.

The first thing the method does is to inject into the page the hidden control for this dialog to use to pass its return value back to the server. Because all the controls on the hosting page have unique IDs, you just have to add some prefix to this to get a unique ID for the hidden control. Listing 7.14 follows the standard ASP.NET approach of using the `$` character to separate the name prefix from the ID.

LISTING 7.14 The `AttachDialog` Method in the User Control

```
Sub AttachDialog(DlgType As DialogType, _
          ControlID As String, _
          EventName As String, _
          MessageText As String, _
          Optional CancelEvent As Boolean = False, _
          Optional SubmitForm As Boolean = False)

  ' create hidden field in page for any return value
  Dim sHidFieldName As String = "AHHClientDlg$" & ControlID
```

LISTING 7.14 Continued

```
  Page.RegisterHiddenField(sHidFieldName, "")

  ' create function name to attach to control
  Dim sFunctionName, sParams As String
  sParams = "('" & sHidFieldName & "', '" _
    & MessageText.Replace("'", "\'") & "', " _
    & (Not CancelEvent).ToString().toLower() & ", " _
    & SubmitForm.ToString().toLower() & ");"
  Select Case DlgType
    Case DialogType.Alert:
      sFunctionName = "return AlertDlgEvent" & sParams
    Case DialogType.Confirm:
      sFunctionName = "return ConfirmDlgEvent" & sParams
    Case DialogType.Prompt:
      sFunctionName = "return PromptDlgEvent" & sParams
  End Select

  ' attach client-side event handler to element
  ' need to determine base control type and cast to the
  ' appropriate type that has an Attributes collection
  Dim oCtrl As Control = Parent.FindControl(ControlID)
  If TypeOf oCtrl Is HtmlControl Then
    CType(oCtrl, HtmlControl).Attributes.Add(EventName, _
                                       sFunctionName)
  ElseIf TypeOf oCtrl Is WebControl Then
    CType(oCtrl, WebControl).Attributes.Add(EventName, _
                                      sFunctionName)
  Else
    Throw New Exception("Control Type Not Supported")
  End If

  ' create client-side script if not already registered
  If Not Page.IsClientScriptBlockRegistered("AHHClientDlg") Then
    Dim sScript As String = vbCrlf _
    & "<script language='javascript'>" & vbCrlf _
    & "<!--" & vbCrlf _
    ... rest of client-side script here ...
    & "//-->" & vbCrlf _
    & "<" & "/script>" & vbCrlf
    Page.RegisterClientScriptBlock("AHHClientDlg", sScript)
  End If

End Sub
```

> **Accessing the Hidden Control Created by the `RegisterHiddenField` Method**
>
> **The `RegisterHiddenField` method inserts the hidden control into the output as literal text, and it does not generate a server control. This means that you cannot access it on the server as you normally do with server controls. To get the value, you have to extract it from the `Request.Form` collection—as you'll see later in this chapter.**

The hidden control is injected into the page using the `RegisterHiddenField` method (this is what ASP.NET uses to inject the hidden controls it requires, such as the one that contains the viewstate for the page). You need a separate hidden control every time the method is called. You could, of course, just dynamically create an `HtmlInputHidden` control instance and insert it into the control tree, but using `RegisterHiddenField` is quicker and easier.

Next, you build up a `String` value that contains the parameters to be used when calling the client-side functions. All these functions require at a minimum the same four parameters: the ID of the hidden control that carries the return value, the text to display in the dialog, and the two Boolean values that control postbacks (the two parameters that control postbacks are optional). For example, this is the signature of the function that displays a `confirm` dialog:

```
function ConfirmDlgEvent(sField, sMsg, bCancel, bSubmit)
```

The parameter string is then added to the function name (notice that you have to return the function value to the underlying control to be able to cancel the event). This is followed by code to attach the function to the control itself. You can get a reference to the control in the hosting page with the `FindControl` method. By calling this for the `Parent` instance (the `Page` object that is hosting the user control), you are actually searching the `Controls` collection of the hosting page.

Converting the Target Control to Its Base Class

The reference that is returned by the `FindControl` method (provided that the control was located) is of type `Control` (the base class for all ASP.NET server controls). However, this class does not have an `Attributes` collection, so you have to cast the reference to either an `HtmlControl` class or a `WebControl` class. These are the namespace-specific base classes for the HTML controls and the Web Forms controls, respectively, and each has an `Attributes` collection.

This means that you have to figure out which type the underlying control actually is. The easiest way to do this in Visual Basic .NET is to use the `TypeOf` statement. The following code:

```
If TypeOf oCtrl Is HtmlControl Then ...
```

> **Testing a Control Object Type in C#**
>
> **In C#, you can use the `is` operator to compare two classes:**
>
> ```
> if (oCtrl is HtmlControl)
> ```
>
> **or you can use the similar `typeof` operator:**
>
> ```
> if (typeof(oCtrl) is HtmlControl)
> ```

returns `True` if `oCtrl` is of type `HtmlControl` or is descended from the class `HtmlControl`, and the reference can therefore be safely converted into an instance of that type.

Provided that the control specified by the user is descended from `HtmlControl` or `WebControl`, you can then add the event attribute to it. If it's not one of these two types, you can do nothing with it, so you throw an exception instead.

The Client-Side JavaScript Code

The final section of code in Listing 7.14 should be immediately familiar by now. You just inject the JavaScript code you need into the page, making sure you only do this once—for the first instance of the user control in the hosting page. Listing 7.15 shows the actual output this creates, rather than the code that shows the JavaScript string being created. This is much easier to read!

LISTING 7.15 The Client-Side Script That Is Injected by the `AttachDialog` Method

```
function AlertDlgEvent(sField, sMsg, bCancel, bSubmit) {
  var hidfield = document.forms[0].elements[sField];
  hidfield.value = sMsg;
  alert(sMsg);
  if (bSubmit) document.forms[0].submit();
  return bCancel;
}

function ConfirmDlgEvent(sField, sMsg, bCancel, bSubmit) {
  var hidfield = document.forms[0].elements[sField];
  if (confirm(sMsg) == true)
    hidfield.value = 'True'
  else
    hidfield.value = 'False';
  if (bSubmit) document.forms[0].submit();
  return bCancel;
}

function PromptDlgEvent(sField, sMsg, bCancel, bSubmit) {
  var hidfield = document.forms[0].elements[sField];
  hidfield.value = prompt(sMsg, '');
  if (bSubmit) document.forms[0].submit();
  return bCancel;
}
```

You can see that Listing 7.15 provides a separate function for each dialog type, although they all take the same parameters and work virtually the same way. After each function gets a reference to its own hidden field, the ID of which is passed in by the `sField` parameter, the function displays the appropriate dialog. Where there is a return value, this value is inserted into the hidden control. Because there seems to be no better option, the `alert` dialog method returns the text it displayed.

> **Using Multiple Forms with the Sample User Control**
>
> **One point to note is that the functions assume that the controls all reside in the first form on the page (`document.forms[0]`). This is likely to be the case with ASP.NET pages, which support only a single server-side form per page. However, it's possible that there may be another client-side form (a `<form>` element that does not carry the `runat="server"` attribute) located before the server-side form. In this case, you have to change the index to the `forms` collection in the script or perhaps pass it in as a parameter instead.**

Then the functions check to see if the user specified that the form should be submitted automatically. If he or she did, the code calls its `submit` method. Otherwise, the functions return the value of the `bCancel` parameter. If this value is `false`, it will prevent the underlying control event from taking place (as you've seen in several earlier examples).

The `GetDialogResult` Method

The second method of the sample user control, `GetDialogResult`, provides an easy way to extract the value for a specific dialog when the page is next posted back to the server. If the dialog has been shown, you know that the return value is in a hidden control, so you just have to pull it out of the `Request.Form` collection. Remember that because you used the `RegisterHiddenField` method to insert the hidden control, it is not actually a server control.

The single parameter to the `GetDialogResult` method is the ID of the control you attached the dialog to, so it's simply a matter of building up the full control ID and returning the matching value from the `Request.Form` collection, as shown in Listing 7.16.

LISTING 7.16 The `GetDialogResult` Method in the User Control

```
Function GetDialogResult(ControlID As String) As String

  ' build hidden field name
  Dim sHidFieldName As String = "AHHClientDlg$" & ControlID

  ' get posted value from Request collection
  Return Request.Form(sHidFieldName)

End Function
```

Browser-Adaptive Script Dialogs

The user control that you've just seen works fine with all browsers that support client-side scripting in JavaScript (which is effectively all modern browsers and most of the older ones). However, you can go further to achieve more useful and attractive results in Internet Explorer, as intimated earlier in this chapter.

Internet Explorer supports VBScript and, through it, the VBScript `MsgBox` and `InputBox` functions that can display more configurable and attractive dialogs. Figures 7.7 and 7.8 demonstrate this; in Figure 7.7, Internet Explorer displays a VBScript `MsgBox` function, and in Figure 7.8 Mozilla displays the standard JavaScript `alert` dialog equivalent.

FIGURE 7.7 A VBScript `MsgBox` function, displaying a custom icon and title.

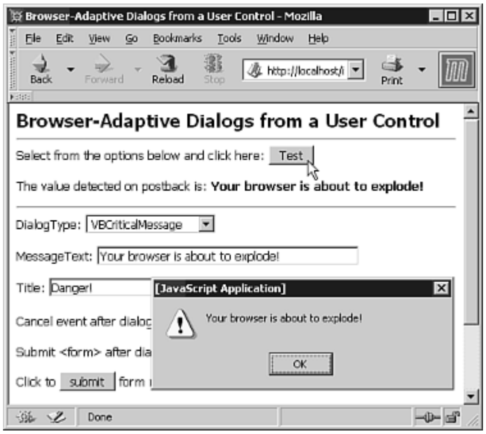


FIGURE 7.8 The standard JavaScript `alert` dialog equivalent to the VBScript `MsgBox` function in Mozilla and other browsers.

Is it worth the effort of building code to display different types of dialogs, when only Internet Explorer users will see any benefit? That depends on what percentage of your visitors use Internet Explorer. According to many independent sites that report traffic figures, more than half of their visitors are using Internet Explorer 5 or Internet Explorer 6 now. Only you can decide, of course, but if you take advantage of a user control like the one here, adding the feature to your sites and applications is a painless process.

To give you an idea what you can achieve, Figure 7.9 shows the seven Internet Explorer–specific dialog types exposed from the sample user control that is described next. There are other types and variations of the `MsgBox` function in VBScript as well (including three-button dialogs), but this example doesn't include them. You could easily add them if required.

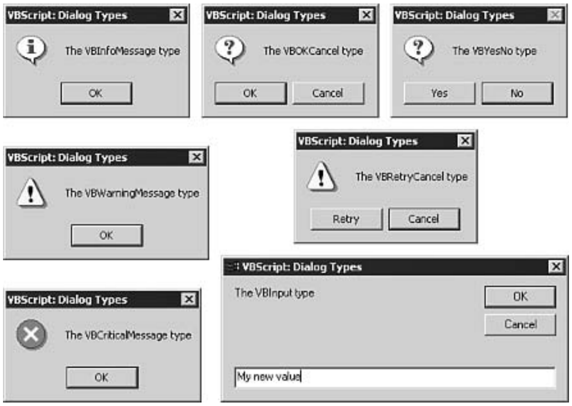


FIGURE 7.9
The dialog types exposed by the sample browser-adaptive script dialog user control.

How the Adaptive Client Dialogs Example Works

This is your first real taste of building adaptive controls that produce different results in different browsers, although you'll see more on this topic through the remainder of this chapter and Chapter 8. To provide two different sets of dialogs, you have to solve at least a couple issues:

- What happens if an Internet Explorer–specific dialog type is specified when the client browser is not Internet Explorer?
- How do you handle the requirement for two different client-side scripting languages?

The following sections look at how the code in this user control differs from the previous JavaScript-only example.

The Changes to the `DialogType` Enumeration in This Example

When using VBScript, you have more dialog types, so the `DialogType` enumeration is different (see Listing 7.17). Notice that you retain the existing three JavaScript types but supplement them with the seven VBScript types. The ordering within the enumeration reflects the grouping of the three main types of dialog—information only, yes/no, and input.

LISTING 7.17 The `DialogType` Enumeration for the Browser-Adaptive Example

```
' enumeration of dialog types
Public Enum DialogType
  Alert = 0
  VBInfoMessage = 1
  VBWarningMessage = 2
  VBCriticalMessage = 3
  Confirm = 4
  VBOKCancel = 5
  VBYesNo = 6
  VBRetryCancel = 7
```

LISTING 7.17 Continued

```
  Prompt = 8
  VBInput = 9
End Enum
```

The Changes to the `AttachDialog` Method in This Example

The VBScript dialogs allow you to specify the title (in the title bar) as well as the message text. So the first change to the `AttachDialog` method allows users to specify the title text as an extra parameter. You can declare it as an optional parameter in this example, slotting it in after the message text:

```
Sub AttachDialog(DlgType As DialogType, _
    ControlID As String, EventName As String, _
    MessageText As String, Optional Title As String = "", _
    Optional CancelEvent As Boolean = False, _
    Optional SubmitForm As Boolean = False)
```

Inside this method, you have to detect which browser you are serving the current page to so that you can decide what output to send back. The ASP.NET `BrowserCapabilities` object is ideal for this, and a reference to it is obtained from the `Browser` property of the current `Request` object:

```
Dim oBrowser As HttpBrowserCapabilities = Request.Browser
```

Among the properties exposed by the `BrowserCapabilities` object is a `Boolean` value that indicates whether the current browser supports VBScript (just what you want):

```
Dim bUseVBS As Boolean = oBrowser("VBScript")
```

Having stored this away in a `Boolean` variable named `bUseVBS`, you continue by creating the two language-specific variables you'll need in order to create parts of the output from the method. These are the language and file extension to use when creating the `<script>` element in the page (because you'll be injecting only a reference to the script file into the page—not the complete script section, as you did in the previous example). These are the two variables you create, with the default values that specify JavaScript for the script file:

```
Dim sLang As String = "javascript"
Dim sExt As String = "js"
```

Now you can use the `Boolean` variable you collected earlier to see if the current browser is Internet Explorer and supports VBScript. If it is, you can specify the language and file extension for the VBScript code file. If it is not, you have to modify the dialog type the user asked for because only the three JavaScript dialogs can be used outside Internet Explorer (see Listing 7.18).

LISTING 7.18 Setting the Language-Specific Variables and Dialog Type

```
If bUseVBS = True Then
  ' set language specific variables
  sLang = "VBScript"
  sExt = "vbs"
Else
  ' can only use JavaScript dialogs
  Select Case DlgType
    Case 1,2,3: DlgType = 0
    Case 5,6,7: DlgType = 4
    Case 9: DlgType = 8
  End Select
End If
```

Identifying and Specifying the Dialog Type

You need to define the function name for the selected dialog. You now have 10 different types of dialogs available, but several of them are just repetitions of the `MsgBox` function, with different values for the buttons parameter. This parameter defines the icon that is displayed in the dialog, the number of buttons and their captions, and which is the default button. Table 7.1 shows the values; basically, you select one value from each of the three groups (Dialog Type, Icon, and Default Button) and add them together to arrive at the button value that will create that type of dialog.

TABLE 7.1

The Button Values for the Different Types of `MsgBox` Dialogs in VBScript

Feature	Value
Dialog Types	
OK (only)	0
OK/Cancel	1
Abort/Retry/Ignore	2
Yes/No/Cancel	3
Yes/No	4
Retry/Cancel	5
Icons	
Critical	16
Question	32
Exclamation	48
Information	64
Default Buttons	
Button 1	0
Button 2	256
Button 3	512
Button 4	768

Listing 7.19 shows the code that identifies the dialog when the user calls the `AttachDialog` method and assigns the appropriate values to two `String` variables. The first is the name of the function (with the `return` keyword so that the result is passed back to the underlying control as before), and the second is the buttons value.

LISTING 7.19 Specifying the Dialog Type and Buttons Value

```
' set dialog type details
Dim sFunction, sButtons As String
Select Case DlgType
  Case DialogType.Alert:
    sFunction = "return AlertDlgEvent"
    sButtons = "0"
  Case DialogType.VBInfoMessage:
    sFunction = "return VBInfoDlgEvent"
    sButtons = "64"
  Case DialogType.VBWarningMessage:
    sFunction = "return VBInfoDlgEvent"
    sButtons = "48"
  Case DialogType.VBCriticalMessage:
    sFunction = "return VBInfoDlgEvent"
    sButtons = "16"
  Case DialogType.Confirm:
    sFunction = "return ConfirmDlgEvent"
    sButtons = "0"
  Case DialogType.VBOKCancel:
    sFunction = "return VBQuestionDlgEvent"
    sButtons = "33"
  Case DialogType.VBYesNo:
    sFunction = "return VBQuestionDlgEvent"
    sButtons = "292"
  Case DialogType.VBRetryCancel:
    sFunction = "return VBQuestionDlgEvent"
    sButtons = "309"
  Case DialogType.Prompt:
    sFunction = "return PromptDlgEvent"
    sButtons = "0"
  Case DialogType.VBInput:
    sFunction = "return VBInputDlgEvent"
    sButtons = "0"
End Select

' create function name to attach to control
sFunction &= "('" & sHidFieldName & "', '" _
  & MessageText.Replace("'", "\'") & "', '" _
  & Title.Replace("'", "\'") & "', " _
```

LISTING 7.19 Continued

```
  & sButtons & ", " _
  & (Not CancelEvent).ToString().toLower() & ", " _
  & SubmitForm.ToString().toLower() & ");"
```

Notice that you need only 6 different functions to cope with the 10 different dialog types. You can generate the 3 different information (single-button) dialogs by using the same function with different button values. The same principle applies to the 3 different question (2-button) dialogs.

Finally, you can build up the `String` value that contains the parameters required for the client-side functions. Then, using the same code as the previous example (not repeated here; refer to Listing 7.14), you attach the functions and their parameters to the event attributes of the target control.

Injecting the Client-Side Script Element

The final task in this example is to inject the appropriate `<script>` element into the output, as shown in Listing 7.20. The signature of the client-side functions contains a couple extra parameters—the title for the dialog and the buttons value:

```
return VBQuestionDlgEvent(sField, sMsg, sTitle,
                          iBtns, bCancel, bSubmit)
```

Notice in Listing 7.20 that you escape any single quotes to prevent a script error because you're using single quotes as the string delimiter in the function calls.

LISTING 7.20 Injecting the Client-Side Script Reference Element

```
' create client-side script if not already registered
If Not Page.IsClientScriptBlockRegistered("AHHClientDlg") Then
 Dim sPath As String = "/aspnet_client/custom/"
 Dim sScript As String = "<script language='" & sLang & "' " _
   & "src='" & sPath & "adaptive-dialog." & sExt & "'><" & "/script>"
 ' add this code to the page
 Page.RegisterClientScriptBlock("AHHClientDlg", sScript)
```

Even though you use JavaScript-style syntax to declare the event attributes, the controls will successfully call into functions written in VBScript as well as functions written in JavaScript.

The Changes to the Client-Side Script Functions in This Example

As you can see from Listings 7.18 through 7.20, you have two different client-side script files (stored in the `/aspnet_client/custom/` folder), one each in JavaScript and VBScript. Depending on which browser hits the page, you'll deliver one or the other of these two files via the `<script>` element you saw being created in the preceding section.

The JavaScript functions are virtually unchanged from the ones used in the previous (non-adaptive) example. The only difference is that they accept the two extra parameters passed to the functions, as in this example:

```
function ConfirmDlgEvent(sField, sMsg, sTitle,
                         iBtns, bCancel, bSubmit) {
```

Of course, they can't make use of these parameters because the JavaScript `alert`, `confirm`, and `prompt` dialogs don't provide the features these parameters are here to support. However, these extra parameters are used by some of the functions in the VBScript code that is delivered to Internet Explorer browsers.

The VBScript Client-Side Script File

If the current browser is Internet Explorer, the `<script>` element you inject into the page will look like this:

```
<script language='VBScript'
        src='/aspnet_client/custom/adaptive-dialog.vbs'></script>
```

Listing 7.21 shows the complete client-side script code in the file `adaptive-dialog.vbs`. You can see that, other than using VBScript syntax rather than JavaScript syntax, the three functions for the `Alert`, `Confirm`, and `Prompt` dialogs are the same as in the JavaScript version. These dialogs are supported by Internet Explorer for compatibility reasons, and they work just the same way (although they differ slightly in appearance).

The other three functions generate either `MsgBox` or `InputBox` dialogs and use the parameters passed to them to control the appearance, the icon, the buttons layout, and the title in the title bar of the dialog. However, each still accesses its own hidden control on the page and inserts the return value into it just as the JavaScript versions of the functions do.

LISTING 7.21 The VBScript Client-Side Functions

```
Function AlertDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  hidfield.Value = sMsg
  Alert(sMsg)
  If (bSubmit) = True Then Document.Forms(0).Submit()
  AlertDlgEvent = bCancel
End Function

Function VBInfoDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  hidfield.Value = sMsg
  MsgBox sMsg, iBtns, sTitle
  If (bSubmit) = True Then Document.Forms(0).Submit()
  VBInfoDlgEvent = bCancel
End Function
```

LISTING 7.21 Continued

```
Function ConfirmDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  If (Confirm(sMsg) = True) Then
    hidfield.Value = "True"
  Else
    hidfield.Value = "False"
  End If
  If (bSubmit) = True Then Document.Forms(0).Submit()
  ConfirmDlgEvent = bCancel
End Function

Function VBQuestionDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  iResult = MsgBox(sMsg, iBtns, sTitle)
  If iResult = 1 Or iResult = 4 Or iResult = 6  Then
    hidfield.Value = "True"
  Else
    hidfield.Value = "False"
  End If
  If (bSubmit) = True Then Document.Forms(0).Submit()
  VBQuestionDlgEvent = bCancel
End Function

Function PromptDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  hidfield.Value = Prompt(sMsg, "")
  If (bSubmit) = True Then Document.Forms(0).Submit()
  PromptDlgEvent = bCancel
End Function

Function VBInputDlgEvent(sField, sMsg, sTitle, iBtns, bCancel, bSubmit)
  Set hidfield = Document.Forms(0).Elements(sField)
  hidfield.Value = InputBox(sMsg, sTitle)
  If (bSubmit) = True Then Document.Forms(0).Submit()
  VBInputDlgEvent = bCancel
End Function
```

The one main difference between the JavaScript and VBScript dialogs is that the return value from a call to the VBScript `MsgBox` function is an `Integer` value that identifies which button the user clicked. These values are summarized in Table 7.2. In the case of the `VBQuestionDlgEvent` function, you have to test for the three different possible values that signify OK, Retry, or Yes and return `"True"` or `"False"`, as appropriate.

TABLE 7.2

The Return Values for the Buttons in a VBScript `MsgBox` Dialog

Button Clicked	Value
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

The hidden controls will contain the same types of values when submitted to the server as in the previous JavaScript-only example, regardless of the language and dialog type used. Therefore, the `GetDialogResult` method is unchanged, and the user control produces exactly the same performance and results as the non-adaptive version—but it takes advantage of the extra capabilities of Internet Explorer to give a more attractive (and often intuitive) outcome.

Integrating Internet Explorer Dialog Windows

You've just seen how you can build a user control that makes it easy to leverage features in Internet Explorer, without compromising support for other browsers. Another dialog-related feature of Internet Explorer is useful if you want to present users with a custom dialog that contains more than the simple buttons or text input combination used in the previous example.

Internet Explorer contains a feature that allows you to pop up both modal and modeless dialogs. The user control you've just seen in the previous example integrates the standard range of modal dialog boxes. The sample page you see here implements modal *dialog windows* in Internet Explorer. Figure 7.10 shows the sample page, with the various options you can select for the dialog window. As well as specifying the size and position, you can control whether the window is resizable, whether it has a status bar and scrollbars, the border style, and the presence of a Help button.

> ### Modeless Dialog Windows in Internet Explorer
> **This example uses only a modal dialog window. Modeless dialog windows are harder to implement, but they are powerful in that you can have code executing in both the main page and the dialog page concurrently, and that code can call methods and pass values between the dialog and main windows as it executes. You can search the MSDN online SDK for `showModelessDialog Method` for more details.**

This example also provides an option to specify whether the main page will be posted back to the server when the dialog window is closed or whether the `click` event of the button that opens the dialog will be canceled (the `CancelEvent` property). Finally, you can pass an optional string parameter (as the `Arguments` property) to the dialog.

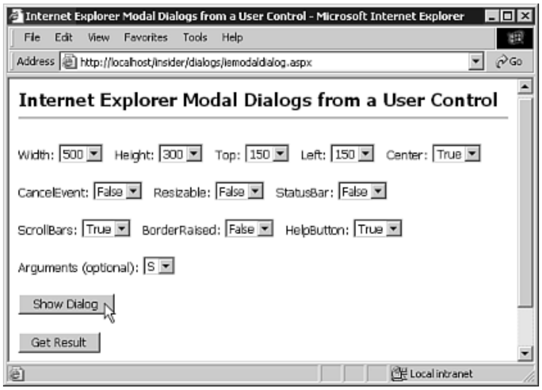


FIGURE 7.10 A sample page that opens an Internet Explorer modal dialog.

The dialog window itself is shown in Figure 7.11. This dialog window displays a list of customers, and the ID of each customer is a clickable link. This is in fact just an ordinary Web page, allowing you to create whatever content you want to appear in the dialog window.

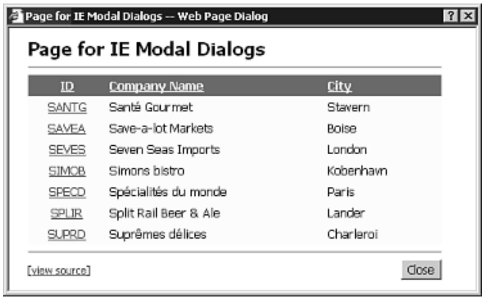


FIGURE 7.11 A modal dialog page in Internet Explorer.

Using Hyperlinks in a Modal Dialog Window

The Web page you open in the dialog window can contain hyperlinks, ordinary `<form>` elements, and/or an ASP.NET server-side `<form runat="server">` element. However, because the dialog is modal, the default behavior is for these to open in a new browser window rather than within the dialog window. To force them to open or post back to the dialog window, you must add a `<base>` element to the `<head>` section of the page that is displayed in the dialog, as in this example:

```
<base target="_self" />
```

This chapter doesn't list the code used to create the page displayed in the dialog window because it's not really relevant to the discussion here. In fact, this example takes advantage of the Internet Explorer–specific feature that can implement *client-side* data binding through the Tabular Data Control (TDC). Because you know that only Internet Explorer will display this page, you can make it more interactive by adopting this useful feature (for example, you can sort the rows by using the links in the column headings, without requiring a postback). You can examine the code if you wish by clicking the [view source] link at the bottom of the page.

Clicking one of the customer ID links closes the dialog window and passes the value back to the main page. When the page is submitted (either automatically when the `CancelEvent` property is

`False` or when the user clicks the Get Result button), the selected value is displayed in the page, as shown in Figure 7.12. If you close the dialog by clicking the Close button rather than selecting a customer, the result comes back as `undefined`. You can detect and use the result in your server-side code, just as you can with the previous dialog examples in this chapter.

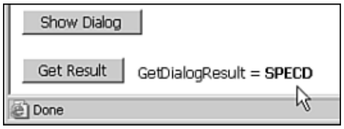


FIGURE 7.12 The result from the sample dialog is displayed in the page after a postback.

Notice that the dialog window contains a Help button in the title bar. If you click this button, the cursor changes to the familiar Help style. Then, when you click on an element within the window, a simple `alert` dialog containing the help text pops up. You have to implement this pop-up dialog yourself by attaching a client-side event handler to the `onhelp` event attribute of each element. Take a look at the source code by clicking the [view source link] at the bottom of the page in the dialog window to see how it's done.

How the Modal Dialog Window Example Works

The general approach and techniques used in this example are the same as those for the previous dialog examples. The principle of attaching client-side code to a control in the page to open the dialog window and then saving the result in a hidden control from where it can be collected by the `GetDialogResult` method is the same. What differs in this example is the client-side code used to show the dialog and extract the result when it is closed. The next section looks at how this is done.

The Internet Explorer `showModalDialog` Method

The syntax of the `showModalDialog` method used in this example is as follows:

```
returnValue = window.showModalDialog(URL, arguments, features)
```

where:

- *`URL`* is the relative or absolute URL of the page to display in the dialog window. Remember to include `<base target="_self">` in the `<head>` section of the page if you want to be able to use ASP.NET pages, forms, or hyperlinks in the dialog.
- *`arguments`* is a `String` value or an array of `String` values that are available to code running in the dialog window page. This example uses a simple `String` value.
- *`features`* is a `String` value that contains instructions on what features to display for the dialog window. These features are summarized in Table 7.3. Notice how the syntax of the string, and the names of some of the features, are subtly different from the features string used to open a new browser window via the more commonly used `window.open` method.

TABLE 7.3

The Features You Can Specify for a Modal Dialog Window

Feature Name	Value	Description
`dialogHeight`, `dialogLeft`, `dialogTop`, `dialogWidth`	An integer or a fractional number	The size and position of the dialog window relative to the top-left corner of the screen. The recommended unit is pixels (`px`), and the minimum width and height are 100 pixels.
`center`	`yes`, `no`, `1`, `0`, `on`, `off`	Locates the dialog at the center of the current browser window. The default is `yes`.
`dialogHide`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the dialog will be shown in print and print preview. The default is `no`. This option is not available for nontrusted applications.
`edge`	sunken, raised	Specifies the style of the dialog border. The default is `raised`.
`help`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the dialog will display the Help button in the title bar. The default is `yes`.
`resizable`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the user can resize the dialog window. The default is `no`. This feature is available only in Internet Explorer 5.5 and higher.
`scroll`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the dialog window will display scrollbars. The default is `yes`.
`status`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the dialog window will display the status bar. The default is `yes` for nontrusted applications and `no` for trusted applications. This feature is available only in Internet Explorer 5.5 and higher.
`unadorned`	`yes`, `no`, `1`, `0`, `on`, `off`	Specifies whether the dialog window will display a border. The default is `no`, but this feature is not available for nontrusted applications.

Listing 7.22 shows the property (field) declarations for the user control that displays a modal dialog window. These declarations equate to the options shown in Figure 7.10 and to the values in Table 7.3 for the features of the dialog window. The code applies sensible default values to each one in case the user of the control does not set them. This example uses simple `Public` fields rather than property accessor routines to avoid repetition and unnecessary complexity—but you might prefer to implement accessors in your own code.

LISTING 7.22 The Property Declarations for the User Control

```
Public Arguments As String = ""
Public BorderRaised As Boolean = False
Public CancelEvent As Boolean = False
Public Center As Boolean = True   'Top and Left must be empty
Public Height As Integer = 400
Public HelpButton As Boolean = False
Public Left As Integer = 150
Public Resizable As Boolean = False    'IE 5.5 and above only
Public ScrollBars As Boolean = True
Public StatusBar As Boolean = False    'IE 5.5 and above only
Public Top As Integer = 150
Public Width As Integer = 600
```

The Changes to the `AttachDialog` Method in This Example

Due to the number of variables the user can specify for a modal dialog, this example exposes the properties in the preceding section rather than passing them all into the `AttachDialog` method (as happens in the two previous examples). So the `AttachDialog` method in this example requires only three parameters: the ID of the control to attach the dialog to, the name of the event for that control, and the URL of the page to display in the dialog window.

Listing 7.23 shows the first of the changes you have to make to the `AttachDialog` method. This example validates the values provided in the parameters, to the extent that they are not empty `String` values, and throws an exception if they are.

Then you can build up the `String` value that represents the features you want to make available for the dialog window. You set the values of a range of individual `String` variables that you'll use to create the final features string later on.

LISTING 7.23 The First Part of the `AttachDialog` Method

```
Public Sub AttachDialog(ControlID As String, _
                        EventName As String, _
                        SourceURL As String)

  ' check values are provided for parameters
  If ControlID = "" Then
    Throw New Exception("Must specify ID of target control")
  End If
  If EventName = "" Then
    Throw New Exception("Must specify name of event to handle")
  End If
  If SourceURL = "" Then
    Throw New Exception("Must specify URL of page to display")
  End If

  ' variables used to build client-side script
  Dim sFeatures, sScript As String
  Dim sResize As String = "no"
  If (Resizable = True) Then
    sResize = "yes"
  End If
  Dim sStatus As String = "no"
  If (StatusBar = True) Then
    sStatus = "yes"
  End If
  Dim sBorder As String = "sunken"
  If (BorderRaised = True) Then
    sBorder = "raised"
  End If
  Dim sScroll As String = "no"
```

LISTING 7.23 Continued

```
  If (ScrollBars = True) Then
    sScroll = "yes"
  End If
  Dim sHelp As String = "no"
  If (HelpButton = True) Then
    sHelp = "yes"
  End If
```

Because a couple of the features for dialog windows are available only in Internet Explorer 5.5 and higher, you next use the `BrowserCapabilities` object (exposed by the ASP.NET `Request.Browser` property) to check the browser type and version (see Listing 7.24). You create a `Decimal` (floating-point) value that contains the major and minor version numbers.

Then, provided that you haven't already done so in a previous instance of the control, you build the client-side script in a `String` variable (as you've done in previous examples). The client-side script here appears to be a bit more complex than that in earlier examples because you have to create the features string as you go along. When the script is complete, you can create the function name and parameters string and attach the whole thing to the target control in exactly the same way as in the two previous examples (the code for this is not repeated here; refer to Listing 7.15).

LISTING 7.24 Sniffing the Browser Type and Creating the Client-Side Script

```
' get browser version, but only if it's Internet Explorer
Dim fVer As Decimal = 0
If Request.Browser.Browser = "IE" Then
  Try
    Dim iMajor As Integer = Request.Browser.MajorVersion
    Dim iMinor As Integer = Request.Browser.MinorVersion
    fVer = Decimal.Parse(iMajor.ToString() & "." _
                       & iMinor.ToString())
  Catch
  End Try
End If

' create client-side script if not already registered
If Not Page.IsClientScriptBlockRegistered("AHHIEDlg") Then
  ' decide whether position is specified or centered
  If (Center = True) Then
    sFeatures = "center:yes;"
  Else
    sFeatures = "dialogTop:" & Top.ToString() _
              & "px;dialogLeft:" & Left.ToString() & "px;"
  End If
```

LISTING 7.24 Continued

```
  sFeatures &= "dialogHeight:" & Height.ToString() _
            & "px;dialogWidth:" & Width.ToString() _
            & "px;edge:" & sBorder & ";scroll:" _
            & sScroll & ";help:" & sHelp & ";"
  'see if it's IE 5.5 or higher
  If fVer >= 5.5 Then
    sFeatures &= "resizable:" & sResize _
              & ";status:" & sStatus & ";"
  End If
  sScript = "<script language='javascript'>" & vbCrlf _
  & "<!--" & vbCrlf _
  & "function IEDlgEvent(sURL, sArgs, sFeatures, sField," _
  & "                    bSubmit) {" & vbCrlf _
  & "  var oHidden = document.getElementById(sField);" & vbCrlf _
  & "  oHidden.value = window.showModalDialog(sURL, sArgs," _
  & "                    sFeatures)" & vbCrlf _
  & "  return bSubmit;" & vbCrlf _
  & "}" & vbCrlf _
  & "//-->" & vbCrlf _
  & "<" & "/script>" & vbCrlf
  Page.RegisterClientScriptBlock("AHHIEDlg", sScript)
End If

' create function name to attach to control
' must escape any single quotes in agruments string
Dim sArgs As String = Arguments.Replace("'", "\'")
Dim sFunctionName As String = "return IEDlgEvent('" _
  & SourceURL & "', '" & sArgs & "', '" _
  & sFeatures & "', '"  & sHidFieldName & "', " _
  & (Not CancelEvent).ToString().ToLower() & ");"

' attach client-side event handler to element
... as in previous examples ...
```

To make it easier to see the result, the client-side script function named `IEDlgEvent` that is generated and injected into the page is shown in Listing 7.25. It takes as parameters the URL of the page to display, the arguments string to pass to the dialog, the features string, the name of the hidden control where the return value will be placed, and a `Boolean` value that specifies whether the underlying control event will be canceled. You can see that the return value from the `showModalDialog` method is simply placed into the hidden control when the dialog is closed, and the value of the `bSubmit` parameter is returned to the underlying control.

LISTING 7.25 The Client-Side `IEDlgEvent` Function That Is Generated by the User Control

```
function IEDlgEvent(sURL, sArgs, sFeatures, sField, bSubmit) {
  var oHidden = document.getElementById(sField);
  oHidden.value = window.showModalDialog(sURL, sArgs, sFeatures)
  return bSubmit;
}
```

The `IEDlgEvent` function, shown in Listing 7.25, is called by the event handler attribute attached to the target control—which, depending on the property settings made in the main page, should look something like this:

```
return IEDlgEvent('dialogpage.aspx', 'S', 'center:yes;
dialogHeight:300px;dialogWidth:500px;edge:Sunken;scroll:yes;
help:yes;resizable:no;status:no;', 'AHHIEDlg$test1', true);
```

Returning a Value from the Modal Dialog

The final issue to consider in the sample page is how to get the value selected in the dialog page back to the main page. In fact, all you need to do is assign it to the `returnValue` property of the `window` object that is hosting the main page and then close the dialog window by calling its `close` method:

```
window.returnValue = sMyReturnVal;
window.close();
```

The value assigned to the `returnValue` property then appears as the return value of the call to the `showModalDialog` method that originally opened the dialog window.

Browser-Adaptive Dialog Windows

As you discovered in the earlier examples in this chapter, it's possible to build user controls that automatically adapt to suit different browsers. The following sections show you how to build a version of the Internet Explorer dialog window example that works in a similar way in other browsers. The sample page that contains the options you can set is shown in Figure 7.13, and you can see that the one extra property is `ModalDialog`, which you can set to `True` or `False`.

When `ModalDialog` is set to `True` and the page is viewed in Internet Explorer, the result is the same as that in the previous example. A modal Internet Explorer dialog window is shown. If you change `ModalDialog` to `False` or view the page in a different browser, it seems at first that the result is the same (see Figure 7.14). However, this is actually a new browser window instance and not a modal dialog. By setting the appropriate features when you call the standard `window.open` method (which all browsers support), you get a similar appearance.

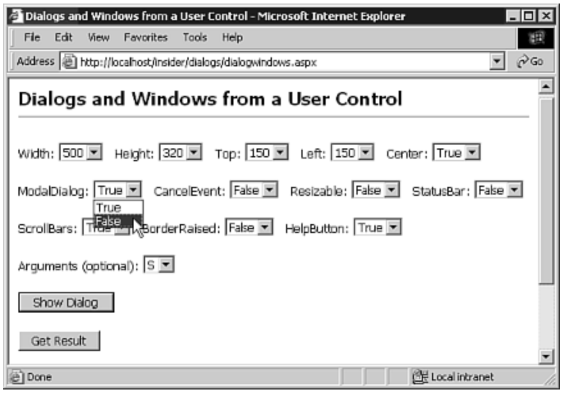


FIGURE 7.13 The browser-adaptive dialog window sample page.

FIGURE 7.14 The nonmodal (new window) dialog page.

However, one major difference in this case is that you can no longer easily provide automatic postback (although it is possible, as you'll see later in this chapter). The new window executes separately from the main window. However, you use script in the new window to insert the value the user selects into the hidden control in the main window, so it can be collected on a postback from the main window (exactly as shown in Figure 7.12). You just click the Get Result button after selecting a value (which closes the new window) to see this occur.

How the Browser-Adaptive Dialog Window Example Works

Much of the code in this example is the same as the code for the previous example. These are the important points where it differs:

- In this example, you have to detect the browser type as before, but this time, you have to determine whether it is Internet Explorer or some other browser.
- If the browser type is not Internet Explorer, you generate a features string that uses the syntax and names specific to the `window.open` method rather than to the `window.showModalDialog` method. Here's an example:

```
"top=150,left=150,height=320,width=500,scrollbars=yes,
resizable=no,status=no,titlebar=yes,menubar=no,location=no,
fullscrceen=no,toolbar=no,directories=no"
```

- You must generate and inject a different client-side script function, which calls the `window.open` method rather than the `window.showModalDialog` method. In addition, when using the `window.open` method, you can't assign the return value to the hidden control.
- There is no arguments parameter for the `window.open` method, but you need to pass the optional argument to the new window. So that the dialog page can work in both versions, you append this value to the URL of the new page as a query string for both the `window.open` method and the `window.showModalDialog` method. You can extract it from the `Request.QueryString` collection within the new page by using the `GetWindowArgument` method (which is described shortly). These are the two functions you generate to open a new browser window and a modal dialog window:

```
function IEDlgEvent(sURL, sArgs, sFeatures, sField, bSubmit) {
  window.open(sURL + '?arg=' + escape(sArgs), '_blank', sFeatures);
  return false;
}

function IEDlgEvent(sURL, sArgs, sFeatures, sField, bSubmit) {
  var oHidden = document.getElementById(sField);
  oHidden.value = window.showModalDialog(sURL + '?arg='
                             + escape(sArgs), '', sFeatures);
  return bSubmit;" & vbCrlf _
```

- You have to use a different technique in a new browser window to get the selected value back to the main window and then close the new window. You'll see how this is achieved in the following section.

As with the modal dialog window example, this chapter doesn't list all the code for the page you see displayed in the new window (the list of customers). However, this example uses *server-side* (ASP.NET) data binding rather than the Internet Explorer–specific client-side data binding approach used in the modal window in the previous example. This means that the dialog page will work on non–Internet Explorer browsers as well as in Internet Explorer. You can use the [view source] link at the bottom of the page in the dialog window to see this code if you wish.

Returning a Value from the New Window

When you open a new browser window to act as a dialog, there is no facility to specify an optional arguments parameter when opening the window or for returning a value to the main window directly (as is possible with the Internet Explorer `showModalDialog` method). Instead, you expose two extra methods from this version of the user control, which are designed to be used in the page that is displayed in the dialog window. Using these methods means that you have to register the user control in the page that you show in the dialog window, as well as in the main page.

Listing 7.26 shows the two methods. The `GetWindowArgument` method takes the ID of the control that the script for opening the dialog or new window was attached to, and it simply extracts the value from the `Request.QueryString` collection where it was placed by the client-side code that opened the dialog or new window. Recall that you pass the value in the query string in all cases, even when using the showModalDialog method because it is the only obvious way to allow the same page to work in the dialog window for all types of browsers.

LISTING 7.26 The GetWindowArgument and SetWindowResult Methods

```
Function GetWindowArgument(ControlID As String) As String
 ' get posted value from Request collection
  Return Server.UrlDecode(Request.QueryString("arg"))
End Function


Sub SetWindowResult(ControlID As String, ReturnValue As String)
  ' build hidden field name
  Dim sHidFieldName As String = "AHHIEDlg$" & ControlID
  ' create client-side script
  Dim sScript As String
  sScript = "<script language='javascript'>" & vbCrlf _
    & "<!--" & vbCrlf _
    & "if (opener != null) {" & vbCrlf _
    & "  var oHidden = window.opener.document.forms[0]" _
    & "  .elements['" & sHidFieldName & "'];" & vbCrlf _
    & "  if (oHidden != null)" & vbCrlf _
    & "    oHidden.value = '" & ReturnValue & "';" & vbCrlf _
    & "  }" & vbCrlf _
    & "else" & vbCrlf _
    & "  window.returnValue = '" & ReturnValue & "';" & vbCrlf _
    & "window.close();" & vbCrlf _
    & "//-->" & vbCrlf _
    & "<" & "/script>" & vbCrlf
  Page.RegisterStartupScript("AHHDlgReturn", sScript)
End Sub
```

The SetWindowResult method, called within the dialog or new window page, accepts the ID of the control that the script to open the dialog or new window was attached to and the value to be returned to the main page. You first check the opener property of the current window to see if it contains a valid reference to the main page window that opened this window. If it does, this provides a reference to the window object where the code that opened the new window was located. You can reference the hidden control in that window and insert the return value into it.

If the opener property is null, you know that the current window is a modal dialog window that was opened with the showModalDialog method. In this case, you can simply set the returnValue property of the current window. This value will automatically be returned to the main window and inserted into the hidden control by the code there that called the showModalDialog method.

Then, in either case, you just have to call the close method of this window. The result is that the new window closes, and the value is available in the main page when the next postback

Implementing `AutoPostback` when the Dialog Window Closes

If you want to provide automatic postback when the new window is closed, you can achieve this by adding code to the script injected by the `SetWindowResult` method. All you need to do is call the `submit` method of the form on the main page before you call the `close` method of the new window.

occurs. As with the earlier examples, it can be extracted at this point, using the same `GetDialogResult` function that is exposed by all the versions of this user control.

The `RegisterStartupScript` Method

Notice that you build the script code as a `String` in the `SetWindowResult` method and then insert it into the page by using the `RegisterStartupScript` method rather than the `RegisterClientScriptBlock` method used in other examples. The `RegisterClientScriptBlock` method is designed to insert complete functions into a page so that they can be called from control event handler attributes (as is done in earlier examples). The script section is inserted into the page at the start of the server-side form section, immediately after the opening `<form>` element.

The `RegisterStartupScript` method is designed to inject into the page client-side code that is *not* a function. If you refer to Listing 7.26, you'll see that you inject *inline* code that will run as the page loads, following the postback. This is how the code inserts the return value into the hidden control on the main page and then closes the new window. This kind of code is often referred to as a *startup script*, and hence the ASP.NET method is called `RegisterStartupScript`.

For the startup script to work properly, the best location is at the end of the page. The `RegisterStartupScript` method actually injects it at the *end* of the server-side form section, just before the closing `<form>` element. Because the controls it references are likely to be on the form, this will work fine in most cases. The corresponding method named `IsStartupScriptRegistered` can be used to check whether this script section has already been registered (that is, already injected into the page).

Summary

This chapter concentrates on user controls and how you can take advantage of many of the features they offer to build reusable content that can implement useful controls or methods in a range of types of browsers.

This chapter starts by looking at how user controls affect the design and implementation of your code and user interface. The main issue here is coping with the possibility that the control may be used more than once in the same page, and there are techniques and features of ASP.NET that help you to manage this. In particular, you can easily prevent duplicate script sections from being injected into a page.

Then, to focus more closely on techniques for building user controls, this chapter shows how you can convert the `MaskedEdit` control you created in Chapter 6 into a user control. Along the way, this chapter looks at issues such as referencing separate script and image files and adding client-side and server-side validation with the ASP.NET validation controls.

Next, this chapter shows how to build a new user control—a `SpinBox` control—from scratch. While many of the techniques are the same as you used for the `MaskedEdit` control, this chapter looks at things like checking property value settings, throwing exceptions, and implementing `AutoPostback` from a composite control.

The remainder of this chapter concentrates on a series of examples that have no visible user interface yet make it easy for you to add useful features to Web applications by taking advantage of client-side dialog boxes and dialog windows. While some of the features are specific to Internet Explorer, this chapter shows how you can quite easily build controls that adapt to different types of browsers.

This last technique described in this chapter—providing graceful fallback for browsers that don't implement features you want to take advantage of—leads neatly in to Chapter 8. You've already learned about and built a couple of these browser-adaptive controls, and you'll see a lot more on this topic in Chapter 8. In particular, you'll extend the `SpinBox` control introduced in this chapter into a full-fledged browser-adaptive server control.

8

Building Adaptive Controls

IN THIS CHAPTER

The previous three chapters discuss different ways to provide useful reusable content for Web sites and Web applications, while taking advantage of the features of more recent browser versions to achieve the best in interactivity and performance. Those chapters concentrate mainly on user controls, which provide an ideal environment to achieve reusability while being relatively quick and easy to build.

This chapter concentrates on an approach mentioned a few times in this book—building *server controls*. This is, in many ways, the ultimate technique for reusable content because it avoids the issues related to user controls that can limit their usefulness.

This chapter looks at two different server controls, both developed from user controls built in previous chapters. You'll see how you can easily convert the `MaskedEdit` control into a server control—effectively a `TextBox` control with extra behavior added.

Then this chapter looks at the `SpinBox` control, again taking it from the user control stage shown in Chapter 7, "Design Issues for User Controls," to a full-fledged server control. The `SpinBox` control is a composite control in that it contains more than one element; it therefore requires some additional implementation. You'll learn how to take this control beyond the first stage of being a basic server control to make it adapt its output for different browsers. You'll also install it in the global assembly cache (GAC) to make it available machinewide to all applications.

The Advantages of Server Controls

Before we dive into implementation of server controls, it's probably a good idea just to reiterate the advantages they provide over other types of reusable content:

- **Server controls hide their implementation from the user in a far more comprehensive manner than user controls**—The source file is compiled into Intermediate Language (IL) code and does not have to be present in order for the control to be used. User controls, on the other hand, are like ordinary ASPX pages in that they have to be present on the machine. They can be opened and the source code viewed, just like an ASPX page. (Code-behind files that are created by Visual Studio .NET are compiled.)
- **Server controls raise events that can be handled in the hosting page**—In fact, this is often a major requirement for a control. Microsoft recommends that event handlers for user controls should only be placed within the user control, which can limit their usefulness in some scenarios.
- **Server controls can be installed in the GAC**—This means that they are available to any application running on the machine. Remember that user controls can be used only within the application where they reside, so they require you to maintain multiple copies if you want to use them in more than one application.

These are three significant features and should convince you that it's worth the extra effort involved in building controls this way. It's certainly not as quick or as easy as building a user control, but you'll find that as you build more, you'll really start to appreciate the advantages.

This book doesn't have room for a full reference or tutorial on building server controls. Besides, you might have already started building your own controls. Therefore, the aim of this chapter is to demonstrate how you can get the most from the techniques involved in building server controls. However, this chapter shows how to get started, the basic features you need to implement for a server control, and how you can achieve the appearance and behavior you want.

> **The ASP.NET QuickStart Server Control Tutorial**
>
> A useful guide for starting to create server controls is included in the QuickStart samples provided with ASP.NET and is available online at `www.dotnetjunkies.com/quickstart/aspplus/doc/webctrlauthoring.aspx`.

The Basics of Building Server Controls

The first step in building a control of any kind is to decide exactly what you want from it—just as you have done with user controls in earlier chapters. You can even take advantage of the same technique you sometimes use when building user controls. It's not unusual to identify sections of code or user interface in ASPX pages that you want to reuse, so you pull them out and package them up as a user control. From there, you develop the code interface and the user interface, often adapting the content as you go to achieve the result you want.

When you do this, you actually complete much of the design process for the equivalent server control. For example, you already know how the user interface should be constructed (which elements and attributes are required), which properties and methods must be exposed, and the implementation of the code within the control.

Of course, you still need to consider how the move to a server control might change things. For example, the ability to expose events might make some tasks much easier to perform in a server control. You might want to raise an event when some values change, and you can then pass those values to the hosting page as properties of an event object—in the same way that many of the built-in ASP.NET controls do.

The Process of Building a Server Control

The following is a list of steps involved in building a server control:

- Design the user interface that the control will implement. This might be as simple as a single control (such as the `MaskedEdit` control you'll see shortly), or it might be a compound control involving multiple elements (such as the `SpinBox` control covered later in this chapter).
- Design the code interface that will be exposed by the control, including the properties, methods, and events that you want the control to provide.
- Figure out which existing class to inherit from. This class can provide many of the features and behavior that a server control must exhibit, and it saves you from having to implement all the basic features yourself. You just override existing features that you don't want, in order to remove them or change their behavior, and add any extra features you need.
- Plan where and how the control must handle the events raised by the ASP.NET page framework so that you know when and where you need to interact with the framework and the base class to create the required output in the page.
- Create the class file to implement the control, compile it, test it, and then deploy it.

In this chapter you'll work through all these steps for two server controls. However, because you've already built them both as user controls, you already roughly know what the code interface, user interface, and implementation should look like.

The Life Cycle of ASP.NET Controls

When you build server controls, the life cycle (that is, the way that the controls are instantiated, the events that they react to, and the point at which they are destroyed) is relatively simple. As you have seen in earlier chapters, the ASP.NET page framework creates an instance of the user control and inserts it into the control tree of the page. It raises the `Init` event for every user control on the page before it raises the `Init` event for the page itself.

Then, after the complete control tree for the page and the controls it contains has been constructed, ASP.NET retrieves the viewstate and any posted data from the form (if this is a post-back) for all the controls, and it sets their values. Finally, when all the information is available, it raises the `Load` event for the page, followed by the `Load` event for each user control.

In almost all cases, you only need to react to the `Load` event in a user control (through an event handler for its `Page_Load` event). At that point, you know that the control tree for the complete page is available, so you can access other controls and their values, both within the user control and in the hosting page.

> **The Events for a User Control**
>
> **The events that a user control can handle mirror those that are available for an ASPX page, such as `DataBinding` (which occurs when the server control binds to a data source), `AbortTransaction` (which occurs when a user aborts a transaction), and `CommitTransaction` (which occurs when a transaction completes).**

Of course, user controls themselves are server controls as well—in the sense that they inherit from the base class `System.Web.UI.UserControl`. Therefore, they receive several other events, such as `PreRender`, `Unload`, and `Disposed`. However, these events are rarely useful for the common kinds of user controls you will create.

The Life Cycle of a Server Control

An ASP.NET server control has a life cycle similar to that of user controls, which is to be expected because both types of controls inherit from the base class for all ASP.NET controls—`System.Web.UI.Control`. The `Control` class handles just six events, which are shown in Table 8.1 in the order in which they occur.

TABLE 8.1

The Events Handled by the `Control` Class, in the Order in Which They Occur

Event	Description
`Init`	Occurs when the control instance is created and initialized.
`Load`	Occurs when the control is loaded into the ASP.NET page as part of the control tree.
`DataBinding`	Occurs when the control binds to a data source.
`PreRender`	Occurs just before the control creates its output into the containing page.
`Unload`	Occurs when the control is unloaded from memory.
`Disposed`	Occurs when the control is released from memory.

As you can see from Table 8.1, there is little difference between the series of events in the life cycle of a server control and that in the life cycle of a user control. However, the .NET Framework provides another two base classes from which you can inherit; they provide far more comprehensive support for building custom server controls.

Recall that there are two types of server controls provided with ASP.NET: the HTML controls in the `System.Web.UI.HtmlControls` namespace and the Web Forms controls in the `System.Web.UI.WebControls` namespace. The latter type of control provides much more in the way of features than the former, including automatic adaptability for different browsers ("up-level" and "down-level"), provision of the `AutoPostback` feature, and a wide range of useful list controls.

The two types of controls are descended from two different base classes, `HtmlControl` and `WebControl`. These classes provide the default behavior that is required by all the server controls that are descended from them. For example, they provide support for viewstate by implementing the `IStateManager` interface and for handling postback values through the `IPostBackDataHandler` and `IPostBackEventHandler` interfaces.

You can override methods and handle events that these interfaces expose, together with the methods and events of the base classes, to build server controls that plug into an ASPX page and behave just like the "native" server controls provided with ASP.NET.

> **Determining the Control Base Type**
>
> **In the section "The `AttachDialog` Method" in Chapter 7, you saw the use of the two different base classes when you were binding events to a control. In that example, you had to determine whether a reference returned from the `FindControl` method was to a control that inherits from `HtmlControl` or `WebControl` by using the statement `If TypeOf oCtrl Is HtmlControl`.**

Creating a Class for a Server Control

A *server control* is simply a .NET `Class` file that is compiled into an assembly and instantiated within an ASP.NET page. Depending on which base class you inherit from, you must import the namespaces that contain that base class and any other classes you use. For example, Listing 8.1 shows the minimum definition of a server control that inherits from the `Control` class.

LISTING 8.1 The Minimum Definition of a Server Control

```
Imports System
Imports System.Web
Imports System.Web.UI

Namespace Stonebroom

  Public Class MyClassName

    ' specify base class to extend
    Inherits Control

    Overrides Protected Sub Render (oWriter As HtmlTextWriter)
      'generate the output required from the control
    End Sub

  End Class

End Namespace
```

The `System.Web.UI` namespace is required because this is where the `Control` and `HtmlTextWriter` classes are defined. As you can see from Listing 8.1, you declare a namespace for the new class, and it should be something specific to you or your organization because it will form part of the fully qualified name of the class. You should not be tempted to use `System`, which is reserved for the classes that are part of the .NET Framework.

You also specify the class you are inheriting from—in this case, `Control`. Then, to generate the output from the control, you override the `Render` method of the base `Control` class. The output you create here will be injected into the ASP.NET page that uses the control.

In some cases, you might need to import other namespaces as well. For example, if you decide to inherit from `HtmlControl` or `WebControl` instead of `Control`, you must import the appropriate namespace—either this:

```
Imports System.Web.UI.HtmlControls
```

or this:

```
Imports System.Web.UI.WebControls
```

And, as you'll see later in this chapter, you often need to import other namespaces. For example, if you want to work with the values sent in a postback to the current page, you need to use the `NameValueCollection` class. This is defined in the namespace `System.Collections.Specialized`, so you would have to import it, as shown here:

```
Imports System.Collections.Specialized
```

Choosing and Extending a Base Class

As discussed earlier in this chapter, one of the most important decisions when building a server control involves which existing class to inherit from. Obviously, you want to get as much functionality as you can for free, by inheriting a control that already does most of the things you want, so you can just add to it the few extra features you require. On the other hand, you can go too far and end up spending more time modifying the existing rich behavior of a class to prevent it from doing things you don't want or need.

In most cases, the obvious choice of base class is `WebControl`. This class implements features that make it easy to hook into the viewstate and postback data architecture, while leaving you free to implement the remainder of the public interface you want to provide to uses of the control. This is what is done in the two examples described in this chapter.

However, if you don't want to access viewstate and postback data—perhaps to provide a control that is not interactive or just exposes methods and no user interface—you might decide to inherit from the base class `Control` instead of `WebControl`.

Inheriting from the `Control` Class

When you inherit from the `Control` class, the common approach is to handle just two events:

- **`Init`**—In this event, you initialize any variables you'll need and possibly generate instances of any other classes you want to use within the control.
- **`Render`**—In this event, you generate the complete output for the control. The `Render` event handler receives a reference to the `HtmlTextWriter` class that will be used to generate the output for the control, and you can call various methods of the `HtmlTextWriter` class to create the output you want for the control. Some of the most commonly used methods are shown in Table 8.2.

TABLE 8.2

Commonly Used Methods of the `HtmlTextWriter` Class for Generating Output from a Control

Method	Description
`Write`, `WriteLine`, `WriteLineNoTabs`	Write the string representation of a variable to output, with or without a carriage return. The `WriteLineNoTabs` method does not inject any prefix tab characters into the output.
`WriteBeginTag`, `WriteFullBeginTag`, `WriteEndTag`	Write the opening or closing tag of the element to output and prefix the output with tabs to maintain output layout. The `WriteFullBeginTag` method adds the closing > character of the opening tag, and the `WriteBeginTag` method omits it so that attributes can be added.
`RenderBeginTag`, `RenderEndTag`	Write the opening or closing tag of the element to output without prefixing them with tabs.
`AddAttribute`, `WriteAttribute`	Add an HTML attribute and its value to the output.
`AddStyleAttribute`, `WriteStyleAttribute`	Add an HTML `style` attribute and its value to the output stream.

As an example of the use of the methods listed in Table 8.2, you can use the code shown in Listing 8.2 to override the `Render` method and generate a `<span>` element from a server control. It creates the opening `<span>` tag, but without the closing >, and then it adds `class` attributes to this tag. Next, it closes the opening tag by using one of the predefined constant values exposed by the `HtmlTextWriter` class. Then it outputs the content of the `<span>` element. The last line completes the `<span>` element by adding the closing tag.

LISTING 8.2 Using the `Render` Method to Generate Output from a Server Control

```
Overrides Protected Sub Render (oWriter As HtmlTextWriter)
  oWriter.WriteBeginTag("span")
  oWriter.WriteAttribute("class", "large-text")
  oWriter.Write(HtmlTextWriter.TagRightChar)
  oWriter.Write("Welcome to my Web page")
  oWriter.WriteEndTag("span")
End Sub
```

Inheriting from the WebControl Class

For anything other than the most basic server control, it makes sense to inherit from the WebControl or HtmlControl class. Usually the WebControl class is the choice because it supports extra features that you might find useful.

The HtmlControl class provides very few properties that define the style or appearance of the controls that descend from it because each one uses control-specific properties to define the behavior of that control. On the other hand, the WebControl class has a host of properties for the border, font, size, and color that are standard across all controls created from this base class.

One advantage of inheriting from the WebControl and HtmlControl base classes is that you can override the various methods they expose to add specific sections of content to the output generated by the control. For example, the CreateChildControls method of each WebControl instance in a page is called when it's time for the control to create any child control that it requires. You just create the child controls you need and add them to the Controls collection of the server control.

Then, when the Render method is called for the control, the child controls that are now part of the control tree create their own output and inject it into the output of the control, in the appropriate location. This approach is the one used to create the sample SpinBox server control you'll see later in this chapter.

The methods of the WebControl class that you commonly override to create custom output are shown in Table 8.3. You'll see several of these methods used in the examples in this chapter.

TABLE 8.3

Commonly Used Methods of the WebControl Class for Generating Output from a Control

Method	Description
AddAttributesToRender	Called when it's time to add HTML attributes and styles for this control to the HtmlTextWriter instance that is creating the output. The output to create these attributes will be generated during the Render method.
CreateChildControls	Called when it's time for the control to create any child controls or other content that is required to implement the control. The output to create these controls will be generated during the Render method. This method is inherited from Control.
Render	Called when it's time to generate the output of the control. A reference to the HtmlTextWriter instance that will create the output is passed to this method.
RenderChildren	Called when it's time for each child control to generate its output. This method is inherited from Control.
RenderContents	Called when it's time for the control to render its content (the text between the opening and closing tags).

Inheriting from Specific Control Classes

Besides inheriting from the WebControl class or the Control class, a third option for creating custom controls is to inherit from an existing control that already provides most of the behavior and appearance you need and simply add or override methods and properties to get the final result you want.

For example, if you want to implement a control that is basically just a text box but with a few added features, you can inherit from the ASP.NET TextBox control. In this case, you don't have to do anything to implement the features that the TextBox control already provides, such as generating an <input type="text"> element, maintaining viewstate, handling postbacks to update the value, or worrying about how to expose style properties.

You can override the methods of the TextBox control to modify the output that is generated, as long as you call the equivalent method on the base class as well. For example, you could override the AddAttributesToRender method to add your own attributes to the <input> element that the TextBox control generates. Then you just call the AddAttributesToRender method of the TextBox control that you're inheriting from to add the "standard" attributes, such as type="text", id="*id-value*", and name="*control-name*". This is exactly what you'll be doing next to create the sample MaskedEdit server control.

Building a MaskedEdit Server Control

The MaskedEdit user control you created in Chapter 6, "Client-Side Script Integration," is basically an ASP.NET TextBox control with extra features added. These extra features consist of attributes you add to the <input type="text"> element that ASP.NET generates for you when you use a TextBox control. The following are the extra attributes and features:

- A title attribute that displays the current mask and an explanation of the mask characters.
- Four event handler attributes that connect the events in the control to the client-side script. These event attributes are onkeydown, onkeypress, onkeyup, and onfocus.
- A client-side script file that you reference in the page through a <script> element and that implements the event handlers for the four events of the text box. This script file is located in the /aspnet_client/custom/ folder of the server.
- A style attribute that defines the font family, the font size, and the URL of the image that is used to create the light gray representation of the mask for the text box background.

Figure 8.1 shows the MaskedEdit control demonstration page that uses the server control built in this chapter. You can see the pop-up ToolTip that the title attribute generates, as well as the underline characters of the light gray background mask.

The MaskedEdit Control Class File

Listing 8.3 shows the skeleton of the Class file that implements the MaskedEdit control. You have to import the System, System.Web, and System.Web.UI namespaces to provide access to the HtmlTextWriter class that appears as a parameter to the methods you are overriding and to provide access to other features of the ASP.NET page architecture that you reference in the code. You also need the System.Web.UI.WebControls namespace so that you can reference the TextBox class you want to inherit from.

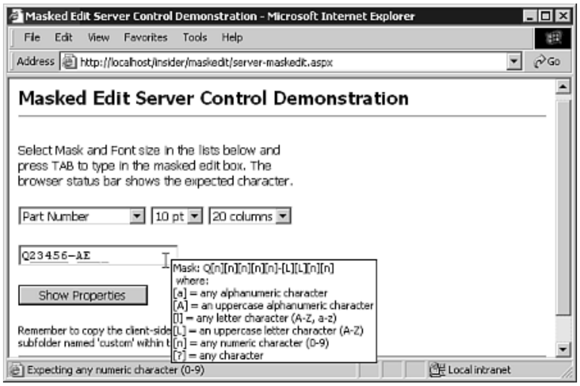


FIGURE 8.1
The MaskedEdit server control demonstration page.

LISTING 8.3 The MaskedEdit Control Class File

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports System.Web.UI.WebControls

Namespace Stonebroom

  Public Class MaskEdit

    ' specify base class to extend
    Inherits Textbox

    ' private internal member variables
      ...

    ' public constructor
      ...

    ' public property accessor declarations
      ...

    ' override AddAttributesToRender method
    ' called when its time to add attributes to control
    OverRides Protected Sub AddAttributesToRender _
                            (writer As HtmlTextWriter)
      ...
    End Sub

    ' override CreateChildControls method
    ' called when its time to create any child controls
```

LISTING 8.3 Continued

```
    OverRides Protected Sub CreateChildControls()
      ...
    End Sub

  End Class

End Namespace
```

After this come the namespace declaration and the class declaration. The first line of code within the class declaration defines the class you want to inherit from—the standard ASP.NET `TextBox` control. This is followed by the declaration of the private "internal" variables you'll be using in the code, the public constructor for the new class, and the property accessor declarations.

Next come the overridden method declarations for the `AddAttributesToRender` method, where you'll add the extra attributes you need to the `TextBox` control, and the `CreateChildControls` method, where you'll generate the client-side script reference.

The Internal Variable Declarations for the `MaskedEdit` Control

There are three variables that you'll need to access in more than one subroutine of the sample class. Two of these represent the values of `Public` properties that are exposed from the class—`Mask` and `FontSize`. You specify default values for all three of these internal variables. Remember that you require a monospaced (fixed-pitch) font for the text box to maintain the alignment between the text and the background mask image:

```
Private _font As String = "Courier New"
Private _mask As String = ""
Private _fontsize As Integer = 10
```

BEST PRACTICE

Providing a Default Constructor for a Class

In order for a class to be created, it must expose a `Public` constructor. In fact, the compiler will automatically add one to the class if you don't provide any constructor routines; however, it's always good practice to include one. You can use the constructor to initialize variables and accept values passed as parameters if required. However, there is no need for users of this component to provide any values when they create an instance of the `MaskedEdit` control, so no parameters are required in this example.

The Public Constructor for the `MaskedEdit` Control

Listing 8.4 shows the constructor for the sample class. Because it inherits from a base class that implements its own functionality (as opposed to, say, inheriting from the root `Object` class), you should consider calling the constructor of the base class first. Although the compiler will look after this for you if you don't call it explicitly, it will only be a call to the default constructor of the base class. If you need to call any other constructor in the base class, perhaps to pass in values, you must do so explicitly.

LISTING 8.4 The Constructor for the `MaskEdit` Class

```
Public Sub New()
  ' call constructor of base class
   MyBase.New()
End Sub
```

The Public Property Declarations for the `MaskedEdit` Control

You need to expose just two properties from the `MaskedEdit` control: `Mask` and `FontSize`. You aren't validating the values applied to these properties in this example, so the property accessor routines just update or return the value of the corresponding internal variables—as shown in Listing 8.5. As you can see, the syntax and techniques for declaring properties is identical to what you used in user controls in Chapters 5, 6, and 7.

> **Exposing Properties and Fields**
>
> **Because this example does not validate or process the values, they could be exposed as fields or `Public` variables instead of using accessor routines, as demonstrated in Chapter 5.**

LISTING 8.5 The `Public` Properties of the `MaskEdit` Class

```
Public Property Mask As String
  Get
    Return _mask
  End Get
  Set
    _mask = value
  End Set
End Property

Public Property FontSize As Integer
  Get
    Return _fontsize
  End Get
  Set
    _fontsize = value
  End Set
End Property
```

The AddAttributesToRender Method for the MaskedEdit Control

The TextBox control exposes two methods that you need to override to add the specific behavior you want for the MaskedEdit control. The AddAttributesToRender method of a control is called by the ASP.NET page framework when it is time for the control to generate the attributes that will be added to the element that implements the control.

Of course, the TextBox control generates an <input type="text"> element and automatically adds all the other attributes required to create a text box in the browser. The attributes it always adds include id and name (which is the same as id). If there is a value in the Text property, the control also adds the appropriate value attribute so that the text box displays the specified text content.

Of course, the control adds attributes for whatever other properties you specify values for—for example, size (from the Columns property), style, maxlength, disabled, and so on. If you override the AddAttributesToRender method in the custom control class, none of these attributes will appear unless you call the AddAttributesToRender method of the TextBox control from which you're inheriting.

Listing 8.6 shows the AddAttributesToRender method in the MaskEdit control class. You have to declare it by using the Overrides keyword to indicate to the compiler that you want to override an existing method of the base class. Notice that the method passes a reference to the HtmlTextWriter class that will be used to generate the output when the .NET Framework calls the Render method of the base class.

Within the AddAttributesToRender method, you first call the AddAttributesToRender method of the base class (TextBox) to force it to generate the standard attributes it requires, and then you add your own custom attributes by using the AddAttribute method of the HtmlTextWriter instance passed to the method.

> **Non-overridden Methods of the TextBox Class**
>
> **Because you aren't overriding the Render method, the class will inherit it from the TextBox class. So when the .NET Framework calls Render on the class, the existing Render method in the base class will be executed. Because you don't need to change the way this behaves, you don't need to override it. And, of course, the same applies to all the other methods of the TextBox class that you aren't overriding in the sample class.**

LISTING 8.6 The AddAttributesToRender Method in the MaskEdit Class

```
OverRides Protected Sub AddAttributesToRender _
                        (writer As HtmlTextWriter)
  ' called when its time to add attributes to control

  ' call base class method to add standard attributes
  MyBase.AddAttributesToRender(writer)

  ' create mask for display as Textbox background
  Dim sQuery As String = _mask
  sQuery = sQuery.Replace("a", "_")
  sQuery = sQuery.Replace("A", "_")
  sQuery = sQuery.Replace("l", "_")
```

LISTING 8.6 Continued

```
sQuery = sQuery.Replace("L", "_")
sQuery = sQuery.Replace("n", "_")
sQuery = sQuery.Replace("?", "_")

' encode it for query string to pass to page
' mask-image.aspx that generates the image
sQuery = Context.Server.UrlEncode(sQuery)

' create Style attribute value string
Dim sStyle As String = "font-family:" & _font _
  & ";font-size:" & _fontsize _
  & "pt;background-image:url(mask-image.aspx?mask=" _
  & sQuery & "&font=" & Context.Server.UrlEncode(_font) _
  & "&size=" & _fontsize.ToString() _
  & "&cols=" & Columns.ToString() & ")"
writer.AddAttribute(HtmlTextWriterAttribute.Style, sStyle)

' declare a carriage return character string
Dim vbCrlf As String = Convert.ToChar(13) _
                     & Convert.ToChar(10)

' create string to use as Tooltip for control
Dim sTip As String = Mask
sTip = sTip.Replace("a", "[a]")
sTip = sTip.Replace("A", "[A]")
sTip = sTip.Replace("l", "[l]")
sTip = sTip.Replace("L", "[L]")
sTip = sTip.Replace("n", "[n]")
sTip = sTip.Replace("?", "[?]")
sTip = "Mask: " & sTip & vbCrlf & " where:" _
  & vbCrlf & "[a] = any alphanumeric character" _
  & vbCrlf & "[A] = an uppercase alphanumeric character" _
  & vbCrlf & "[l] = any letter character (A-Z, a-z)" _
  & vbCrlf & "[L] = an uppercase letter character (A-Z)" _
  & vbCrlf & "[n] = any numeric character (0-9)" _
  & vbCrlf & "[?] = any character"
writer.AddAttribute(HtmlTextWriterAttribute.Title, sTip)

' add client-side event handler attributes
Dim sParams As String = "(event, this, '" & _mask & "')"
writer.AddAttribute("onkeydown", _
                    "return doKeyDown" & sParams)
writer.AddAttribute("onkeypress", _
                    "return doKeyPress" & sParams)
```

LISTING 8.6 Continued

```
  writer.AddAttribute("onkeyup", "return doKeyUp" & sParams)
  writer.AddAttribute("onfocus", "return doFocus" & sParams)

End Sub
```

You can see from Listing 8.6 that the implementation is virtually identical to what you used in the user control version of the `MaskedEdit` control in Chapters 6 and 7. The major difference is that you have to reference the built-in ASP.NET objects (such as `Server`, `Request`, and `Session`) via the static `Context` object. For example, to URL-encode the query string for the page that creates the background image for the text box, you use `Context.Server.UrlEncode(value)`.

The other point to watch is that several namespaces are imported by default into ASP.NET pages, whereas none are imported by default into a class file. This is why you have to import the `System`, `System.Web`, and other namespaces into the class. It means that what you think are obvious object types and constants—ones you use in ASP.NET pages all the time—are often not available in a class file. An example of this is the predefined `vbCrlf` constant that represents a carriage return. It lives in the `Microsoft.VisualBasic` namespace, which is imported by default into ASP.NET pages written in Visual Basic.NET but not into a class file. You therefore have to either import this namespace, which is useful if you want to use other Visual Basic .NET–specific methods, or declare your own equivalent by using the following:

```
' declare a carriage return character string
Dim vbCrlf As String = Convert.ToChar(13) _
                     & Convert.ToChar(10)
```

The `CreateChildControls` Method for the `MaskedEdit` Control

The second method you override in the `MaskEdit` class is the `CreateChildControls` method. You might wonder why, because you don't need any child controls. This method is usually used when building composite controls, and it is called at the point where the control should construct the tree of child controls and add them to its `Controls` collection.

However, you still have to generate the `<script>` reference that will load the client-side script file you need to make the control interactive. You can do it any time between when the control is initialized and when the `Render` method is called. However, the call to `CreateChildControls` indicates that ASP.NET is in the process of deciding exactly what the page will contain in the way of controls and literal content, so it's as good a place as any to register the client-side script.

Listing 8.7 shows the `CreateChildControls` method in the `MaskEdit` class, which overrides the method in the `TextBox` class from which you're inheriting. All you have to do is build the `<script>` element and call the `RegisterClientScriptBlock` method of the hosting page to indicate that you want this script injected into the output directly after

> **The `Page` Property of a Server Control**
>
> **All server controls expose the `Page` property. This returns a reference to the page that is hosting the control, regardless of the hierarchy of the page and the container (in this case, the `Controls` collection) in which the control resides. This property is inherited from the base class `Control`.**

the opening server-side `<form>` tag. So, again, the code in this method is exactly the same as what is used in the user control version of the `MaskedEdit` control in Chapter 7.

LISTING 8.7 The `CreateChildControls` Method in the `MaskEdit` Class

```
OverRides Protected Sub CreateChildControls()
  ' called when its time to create any child controls
  ' just used here to add client-side script section

  ' see if previous instance of this control has already
  ' added the required JavaScript code reference to the page
  If Not Page.IsClientScriptBlockRegistered("StonebroomMaskEdit") Then
    Dim sPath As String = "/aspnet_client/custom/"
    Dim sScript As String = "<script language='javascript' " _
      & "src='" & sPath & "maskedit.js'><" & "/script>"
    ' add this JavaScript code to the page
    Page.RegisterClientScriptBlock("StonebroomMaskEdit", sScript)
  End If

End Sub
```

Compiling and Testing the `MaskedEdit` Control

After you create the class file and save it with the `.vb` file extension, you can compile it and make sure it works. We provide a batch file named `make.bat` with the examples for this chapter (see `www.daveandal.net/books/6744/`), which saves you from typing the complete command for the compiler each time. This file contains just the following:

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322\vbc /t:library
➥/out:..\..\bin\maskedit.dll /r:System.dll,System.Web.dll maskedit.vb
```

BEST PRACTICE

Specifying the Versions of Command-Line Tools

When you use a command-line compiler (or any other command-line tools in the .NET Framework), it's a good idea to explicitly specify the path to the Visual Basic .NET compiler version to be used if you have more than one version of the .NET Framework installed. If you just type `vbc` (or `vbc.exe`), the version that comes first in your current `Path` environment variable will be used. This might not be the correct version if you have multiple versions of the .NET Framework installed, so it's a good idea to include the full path.

An alternative to specifying the full path to the `vbc.exe` program (and, as you'll see later, other .NET Framework utilities) is to edit the `Path` variable for your machine to point to the version you want to use. To do so, you select Start, Control Panel, System and open the Advanced page in the System Properties dialog. Then you Click Environment Variables and then find the Path entry in the System Variables portion of the Environment Variables dialog and click Edit (see Figure 8.2).

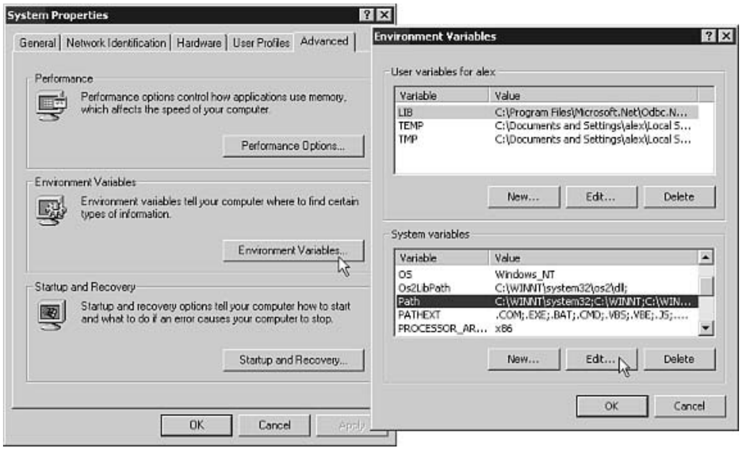


FIGURE 8.2
Editing the `Path` environment variable to point to the correct .NET Framework version.

The Parameters Required by the VBC Compiler

In this chapter you're creating a .NET assembly, so the `t(arget)` parameter specifies that you want a `library` (a DLL), and the `out` parameter specifies the path of the `bin` folder within the root folder of the sample files and names the DLL `maskedit.dll`.

You also have to provide the names of all the .NET Framework files that implement namespaces from which you use classes in the file. This means that you must provide the values `System.dll` and `System.Web.dll` for the `r(eferences)` parameter. `System.Web.dll` implements the namespaces `System.Web`, `System.Web.UI`, `System.Web.UI.HtmlControls`, and `System.Web.UI.WebControls`, as well as others based on `System.Web`. The final parameter is the name of the source file (in this case, `maskedit.vb`) in the current folder.

Of course, if you have Visual Studio .NET, you can write your class files within the IDE and compile them directly by using the Visual Studio .NET menu commands.

> **Using the "Command Prompt Here" Utility**
>
> **One of the easiest ways to use the command-line compilers (both for Visual Basic .NET and C#) is to install the TweakUI add-in or the Power Toys add-in, which install a "Command Prompt Here" link on the right-click menu for a folder in Windows Explorer. Selecting this link opens a command window on the current folder, making it easy to use the** `make.bat` **files provided with the examples for this book or any that you create yourself. For more information, see** `www.microsoft.com/ntworkstation/downloads/PowerToys/Networking/NTTweakUI.asp` **or** `www.microsoft.com/technet/ScriptCenter/other/`

Testing and Deploying the `MaskedEdit` Control

After you compile the class file into an assembly and place it into the `bin` folder, you can test it in an ASP.NET page. The first step is to register the assembly with the page. You use a `Register` directive, as you did with the user control version in Chapter 7, but this time you use the

`Assembly` attribute to specify the name of the assembly without the `.dll` file extension (when registering a user control, you use the `Src` attribute to specify the location of the `.ascx` file instead):

```
<%@ Register TagPrefix="ahh" Namespace="Stonebroom"
             Assembly="maskedit" %>
```

Remember the Client-Side Script Files!

Remember to copy the client-side script file `maskedit.js` provided with the examples into a new subfolder named `custom` within the `aspnet_client` folder of your Web site.

Then you can declare an instance of the server control, just as you did with the user control. However, this time you have an event that you can handle in the hosting page. You didn't declare or implement this event in the class file, but because you inherited the `TextBox` class, all its events are exposed from the custom control as well. You'll react to the `OnTextChanged` event and handle it with a routine named `TextValueChanged`:

```
<ahh:MaskEdit id="oCtrl" runat="server"
              OnTextChanged="TextValueChanged" />
```

Now you can add the server-side code that implements the event handlers to the ASPX page that will use the `MaskedEdit` control. In the `Page_Load` event, you set three properties of the control, including the two custom properties you added to it—the `Mask` and `FontSize` properties. All three values come from the drop-down lists declared within the HTML section of the page (as shown in Figure 8.3).

FIGURE 8.3 Displaying the property values and detecting events in the `MaskedEdit` server control demonstration page.

There's also a button on the page that has the caption Show Properties. When clicked, it executes the `ShowProperties` event handler shown in Listing 8.8. This simply extracts the current value of some properties—including the values of our two custom properties `Mask` and `FontSize`—from the `MaskedEdit` control and displays them in a `Label` control on the page.

LISTING 8.8 The Event Handlers in the MaskedEdit Control Demonstration Page

```
Sub Page_Load()
  oCtrl.Mask = selMask.SelectedValue
  oCtrl.FontSize = selSize.SelectedValue
  oCtrl.Columns = selCols.SelectedValue
End Sub

Sub ShowProperties(Sender As Object, Args As EventArgs)
  lblResult.Text &= "Property values:" _
    & "<br />  Mask: '" & Server.HtmlEncode(oCtrl.Mask) _
    & "<br />  FontSize: " & oCtrl.FontSize _
    & "<br />  Columns: " & oCtrl.Columns _
    & "<br />  Text: '" & oCtrl.Text & "'<br />"
End Sub

Sub TextValueChanged(Sender As Object, Args As EventArgs)
  lblResult.Text &= "Detected TextChanged event for control " _
    & Sender.ID & ".<br />"
End Sub
```

The third event handler shown in Listing 8.8 displays a message whenever the TextChanged event is raised by the MaskedEdit control. You'll see this appear whenever you change the text in the MaskedEdit control and post the page back to the server by clicking the Show Properties button. Figure 8.3 shows the results.

Clearing the Contents of the TextBox Control

Not shown in Listing 8.8 but included in the sample page is an event handler that simply clears the contents of the text box whenever you select a different mask in the first drop-down list. The MaskedEdit control, because it is based on the TextBox control, automatically maintains its value through the viewstate of the page, so it must be cleared whenever a different mask is selected.

Building a SpinBox Server Control

In Chapter 7 you built a composite control that generates a SpinBox control in the browser. This neat and useful control allows users to easily enter or select a numeric value, and you built it so that the up and down buttons could be used to change the value without requiring postback to the server.

In this chapter, you'll convert that control into a server control, to demonstrate how easy it is to build even quite complex composite controls that react just like the standard Web Forms controls provided with ASP.NET. Figure 8.4 shows the SpinBox control demonstration page, and you can experiment with it to see how it works and the properties it exposes.

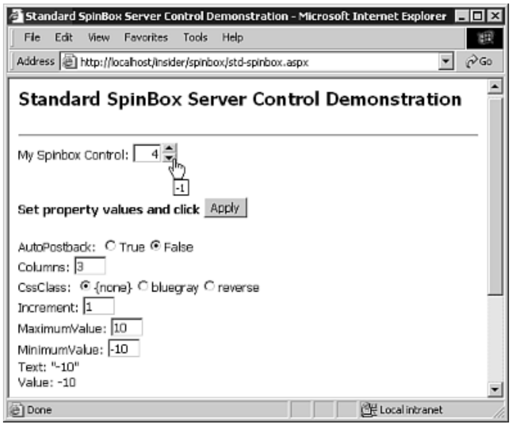


FIGURE 8.4 The `SpinBox` server control demonstration page.

The Standard `SpinBox` Control `Class` File

The `SpinBox` control shown in Figure 8.4 is implemented as a `Class` file, just like the `MaskedEdit` control examined in the preceding sections of this chapter. Listing 8.9 shows the `Imports` statements for the namespaces you'll need and the declaration of the namespace and class for the control.

Notice that you need to import an extra namespace that is not in the `MaskedEdit` example. You need to be able to reference instances of the `NameValueCollection` type in one of the routines, as you'll see later, and that type is defined in the `System.Collections.Specialized` namespace.

LISTING 8.9 The Namespace and Class Declarations for the `StandardSpinBox` Class File

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports System.Web.UI.WebControls
Imports System.Collections.Specialized

Namespace Stonebroom

  Public Class StandardSpinBox

    ' specify base class to extend
    Inherits WebControl

    ' need to be able to handle postbacks
    Implements IPostBackDataHandler

    ...
    ... code here to implement SpinBox control
    ...
```

LISTING 8.9 Continued

```
  End Class
End Namespace
```

In addition, in this example, you're inheriting the `WebControl` base class. The `MaskedEdit` control you looked at previously in this chapter is basically just a text box, so it makes sense for it to inherit from `TextBox`; then you can add the extra features you require. However, the `SpinBox` control is a composite control, and it is in fact based on a `<span>` element into which you insert child controls. Figure 8.5 shows the components of the `SpinBox` control (this is the same structure you created in the user control example in Chapter 7).

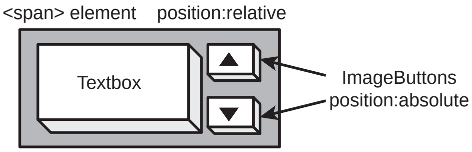


FIGURE 8.5 The structure of the `SpinBox` control.

The final part of Listing 8.9 also indicates that you have to do more work within the composite control than you did in the simple `MaskedEdit` control example. Because you're inheriting from `WebControl`, you will have to add to the class code to handle postbacks and viewstate so that the text box can maintain its value when the page is posted back to the server each time. You'll have to implement methods to handle all this, although it's not actually that difficult to do. To make it all work, the class must implement the `IPostBackDataHandler` interface that declares the methods you'll use to read and set values in the viewstate and the `Request` collections following a postback. You'll see the ramifications of this, and what the methods must accomplish, later in this chapter.

The `Public` Property and `Private` Variable Declarations

The `SpinBox` control exposes eight `Public` properties, as shown in Table 8.4. These are the same properties you implemented in the user control version of the `SpinBox` control in Chapter 7.

TABLE 8.4
The Properties Exposed by the Standard `SpinBox` Server Control

Property	Description
`AutoPostback`	A `Boolean` value that indicates whether clicking the up or down button will cause a postback to the server.
`Columns`	An `Integer` value that determines how wide the text box will be (approximately the number of characters it will hold).
`CssClass`	A `String` value that is the CSS style class to apply to the text box within the control.
`Increment`	An `Integer` value that determines the increase or decrease in the value for the up and down buttons and the up- and down-arrow keys.

TABLE 8.4
Continued

Property	Description
`MaximumValue`	An `Integer` value that indicates the maximum value the `SpinBox` control can be set to.
`MinimumValue`	An `Integer` value that indicates the minimum value the `SpinBox` control can be set to.
`Text`	A `String` value that represents the value displayed in the control.
`Value`	An `Integer` that is equivalent to the value displayed in the control.

To support these `Public` properties, you also declare a series of `Private` internal variables that will hold the values across the various routines in the class. Listing 8.10 shows the internal variable and property accessor declarations in the class file. You can see that the property declarations are identical to those in the `SpinBox` user control example in Chapter 7.

LISTING 8.10 The `Private` Internal Variables and `Public` Property Declarations of the `SpinBox` Control

```
' private internal member variables
Private _autopostback As Boolean = False
Private _columns As Integer = 3
Private _cssclass As String = ""
Private _increment As Integer = 1
Private _maxvalue As Integer = 99
Private _minvalue As Integer = 0
Private _text As String = ""

' public property accessor declarations
Public Property AutoPostback As Boolean
  Get
    Return _autopostback
  End Get
  Set
    _autopostback = value
  End Set
End Property

Public Property Columns As Integer
  Get
    Return _columns
  End Get
  Set
    If (value > 0) And (value < 1000) Then
      _columns = value
    Else
      Throw New Exception("Columns must be between 1 and 999")
    End If
```

LISTING 8.10 Continued

```
  End Set
End Property

Public OverRides Property CssClass As String
  Get
    Return _cssclass
  End Get
  Set
    _cssclass = value
  End Set
End Property

Public Property Increment As Integer
  Get
    Return _increment
  End Get
  Set
    If value > 0 Then
      _increment = value
    Else
      Throw New Exception("Increment must be greater than zero")
    End If
  End Set
End Property

Public Property MaximumValue As Integer
  Get
    Return _maxvalue
  End Get
  Set
    If value > _minvalue Then
      _maxvalue = value
    Else
      Throw New Exception("MaximumValue must be greater than " _
                & "the current MinimumValue")
    End If
  End Set
End Property

Public Property MinimumValue As Integer
  Get
    Return _minvalue
  End Get
  Set
```

LISTING 8.10 Continued

```
    If value < _maxvalue Then
      _minvalue = value
    Else
      Throw New Exception("MinimumValue must be less than " _
                & "the current MaximumValue")
    End If
  End Set
End Property

Public Property Text As String
  Get
    Return _text
  End Get
  Set
    Dim iValue As Integer
    Try
      iValue = Int32.Parse(value)
    Catch
      Throw New Exception("Text property must represent " _
                & "a valid Integer value")
    End Try
    If (value >= _minvalue) And (value <= _maxvalue)
      _text = value
      SetMaxMinValues()
    Else
      Throw New Exception("Text property must be within" _
                & "the current MinimumValue and MaximumValue")
    End If
  End Set
End Property

Public Property Value As Integer
  Get
    Try
      Return Int32.Parse(_text)
    Catch
    End Try
  End Get
  Set
    If (value >= _minvalue) And (value <= _maxvalue)
      _text = value.ToString()
    Else
      Throw New Exception("Value property must be within the " _
                & "current MinimumValue and MaximumValue")
```

LISTING 8.10 Continued

```
    End If
  End Set
End Property
```

You also need a few variables to reference child controls within the class file, and you can declare these outside any of the routines to make them available across the whole class file:

```
' to hold child control references
Private oTextBox As TextBox
Private oImageUp, oImageDown As ImageButton
```

You might recall from Chapter 7 that you used a separate routine named `SetMaxMinValues` in the user control version of the `SpinBox` control to check whether the current value in the text box (the `Text` property) is within the currently defined maximum and minimum values. You do the same in the server control version, calling it from the `Set` section of the `Text` property accessor (see Listing 8.10). The `SetMaxMinValues` routine is shown in Listing 8.11.

LISTING 8.11 The `SetMaxMinValues` Routine

```
' check if current value of Textbox (in _text member variable)
' is within current max and min limits, and reset if not
Private Sub SetMaxMinValues()
  Dim iValue As Integer
  Try
    iValue = Int32.Parse(_text)
  Catch
    iValue = _minvalue
  End Try
  If iValue < _minvalue Then
    iValue = _minvalue
  End If
  If iValue > _maxvalue Then
    iValue = _maxvalue
  End If
  _text = iValue.ToString()
End Sub
```

The Public Constructor for the `SpinBox` Control

As with the `MaskedEdit` control, you include a default constructor (that is, a constructor that does not accept any parameters) within the `SpinBox` control class file. However, this time there is one important difference in the implementation. When you inherited from `TextBox` in the `MaskedEdit` control example, you called the constructor of the base class with no parameters. This is because the default and only constructor for the `TextBox` class does not accept any parameters.

In the SpinBox control, you are inheriting from WebControl, which has no predefined element type as a TextBox control does. You saw earlier in this chapter that you want to implement the SpinBox control as a <span> element that contains the text box and up and down buttons, so you call the constructor of the WebControl base class with the name of the element (the tag name) that you want to create.

Listing 8.12 shows the constructor for the server control, and as you can see, you specify "span" as the parameter to the WebControl constructor.

LISTING 8.12 The Constructor for the Standard SpinBox Control

```
' public constructor
Public Sub New()
  ' call base method first with element type
  ' root element for control will be a SPAN
  MyBase.New("span")
End Sub
```

The parameter to the base constructor is used to set the TagName property of the control. As an alternative, you could use one of the predefined values in the HtmlTextWriterTag enumeration, as in this example:

```
MyBase.New(HtmlTextWriterTag.Span)
```

You could even call the constructor of the WebControl class with no parameters. In that case, the constructor defaults to creating a <span> element anyway, so you could actually omit the parameter after all!

Overriding the CreateChildControls Method

The SpinBox control is a composite control, so you have to create the child controls it requires and add them to the Controls collection of the <span> element that forms the root of the control. You do this by overriding the CreateChildControls method of the <span> control. Because the <span> control (the base class) does not itself create any child controls, you don't need to call its CreateChildControls method.

The next several listings show the CreateChildControls method. We'll look at the parts of the method in the following sections to make it easier to see what's going on. There is a lot of code, but much of it is repetitive in that you need to generate the child controls, add all the attributes you require to each one, and then add the attributes to the control tree of the root <span> control.

Listing 8.13 shows how you first build up a string for the prefix of the controls, taking into account the ID of the root <span> element that implements the control (as exposed by the UniqueID property of the control). This value is set by the user when the control is inserted into a page, or it is allocated automatically by ASP.NET if no ID is specified. You'll use the control ID to build up the unique IDs for the child controls you create so that, when the element is created and inserted into the page, the child control IDs will be a combination of the root control ID, an underscore, and the ID of the child control. This key will be required when you create the client-side event handlers later in the code.

LISTING 8.13 Setting Attributes for the Base Control in the CreateChildControls Method

```
OverRides Protected Sub CreateChildControls()
  ' called when its time to create the child controls
  ' create HTML elements and ASP.NET server controls
  ' set properties and add to Controls collection

  ' control ID prefix for contained controls
  Dim sCID As String = Me.UniqueID & "_"

  ' check if value is within max and min limits
  SetMaxMinValues()

  ' set properties (attributes) of root SPAN element
  Me.Style("position") = "relative"

  ' save current value of Textbox in viewstate
  ViewState(sCID & "textbox") = _text
  Context.Trace.Write("CreateChildControls:" & Me.UniqueID, _
                    "Saved value '" & _text & "' in viewstate")
  ...
```

You also take this opportunity to call the SetMaxMinValues method (shown in Listing 8.11) to ensure that the current value of the Text property is within the current maximum and minimum values. Then you can add the position:relative style selector you need to the root <span> element.

The final action in this part of the CreateChildControls method is to save the current value of the text box in the viewstate of the page. Normally a TextBox control does this automatically (in the MaskedEdit control, it does so when you call the CreateChildControls method of the base TextBox class). However, you're inheriting from WebControl this time to create the root <span> element—not calling its CreateChildControls method.

You also add a feature here that can help you debug controls. This feature also makes it easier to understand what's happening inside controls when they are instantiated and used. You reference the Trace object of the hosting ASP.NET page through the static Context object and write a message to it—including the current value of the control. As you'll see later, this appears in the output generated by ASP.NET when tracing is enabled in the hosting page.

Saving Control Values in the Viewstate

The sample control uses the complete ID of the text box, including the prefix (made up of the root element ID and an underscore), when storing the value in the viewstate. This isn't actually required because the page framework automatically looks after storing values for multiple instances of a control. However, the full ID is required for use when you create the client-side event handlers, so it is used here as well.

Building a Tree of Child Controls

The section of the `CreateChildControls` method shown in Listing 8.14 creates the child controls and adds them to the `Controls` collection of the root `<span>` element. It's simply a matter of instantiating an instance of a `TextBox` control and two `ImageButton` controls and then setting the appropriate attributes for them, including the unique IDs for the controls, the CSS position and size selectors, and any other `style` selectors you need.

LISTING 8.14 Creating the Child Controls in the `CreateChildControls` Method

```
...
' create Textbox control, set properties
' and add to Controls collection
oTextBox = New TextBox()
With oTextBox
  .id = sCID & "textbox"
  If _cssclass <> "" Then
    .CssClass = _cssclass
  End If
  .Columns = _columns
  .Style("top") = "0"
  .Style("left") = "0"
  .Style("width") = _columns * 10
  .Style("text-align") = "right"
  .Text = _text
End With
Controls.Add(oTextBox)

' create "up" ImageButton control, set
' properties and add to Controls collection
oImageUp = New ImageButton()
With oImageUp
  .id = sCID & "imageup"
  .Style("position") = "absolute"
  .Style("top") = "0"
  .Style("left") = oTextBox.Style("width")
  .Width = New Unit(16)
  .Height = New Unit(10)
  .ImageUrl = "~/images/spin-up.gif"
  .AlternateText = "+" & _increment.ToString()
  .BorderStyle = BorderStyle.None
  .BorderWidth = New Unit(0)
  .Attributes.Add("border", "0")
End With
Controls.Add(oImageUp)

' create "down" ImageButton control, set
```

LISTING 8.14 Continued

```
' properties and add to Controls collection
oImageDown = New ImageButton()
With oImageDown
  .id = sCID & "imagedown"
  .Style("position") = "absolute"
  .Style("top") = "10"
  .Style("left") = oTextBox.Style("width")
  .Width = New Unit(16)
  .Height = New Unit(10)
  .ImageUrl = "~/images/spin-down.gif"
  .AlternateText = "-" & _increment.ToString()
  .BorderStyle = BorderStyle.None
  .BorderWidth = New Unit(0)
  .Attributes.Add("border", "0")
End With
Controls.Add(oImageDown)
...
```

For the `TextBox` control, you also set the `Columns` and `Value` properties to the current values of the internal `_columns` and `_text` variables. For the `ImageButton` controls, you set the CSS absolute positions of each one, using the width of the `TextBox` control. Note that if you don't set the value of a CSS style selector, such as `width` or `top`, the selector returns `null` when you try to read it.

You also set the `Width` and `Height` properties of the two `ImageButton` controls. In browsers other than Internet Explorer, the output generated by an `ImageButton` control will contain the HTML `width` and `height` attributes rather than the equivalent style selectors. You have to create new instances of the `Unit` class to set these properties programmatically because these (and some other) properties expect a `Unit` type and not just simple integer values. This allows them to be set, for example, to values such as 20px or 15%.

Other features you set for the two `ImageButton` controls are the URL of the up and down button images (in the `images` subfolder of the

Setting the Size and Position of the Contained Controls

Notice that the code sets the top and left positions of the text box, even though it is not absolutely positioned, as well as the width. The width is calculated by multiplying the number of columns required by 10. As discussed when looking at the `MaskedEdit` control in Chapter 7, it's extremely difficult to equate the actual width with the number of columns. The method used here gives a reasonable result with the average font sizes that are used in ASP.NET pages.

Removing Image Borders in All Browsers

Notice that you remove the border by setting the `BorderStyle` and `BorderWidth` properties of the `ImageButton` controls, as well as by specifically adding the `border="0"` attribute to them. Some older browsers (in particular, Navigator 4.x) require this to be present to prevent the border from appearing, and the `ImageButton` control does not add it automatically.

current virtual application) and the values for the alt attributes (to indicate what the button does in the pop-up ToolTip, as is visible in Figure 8.4). You also remove the border from the images.

All these property settings equate to those you applied in the user control version of the SpinBox control in Chapter 7, although in that case you created the child controls declaratively within the user interface section of the .ascx file and set the properties by using attributes. This time, of course, you've had to do it all programmatically. And, as you create each child control, you add it to the Controls collection of the root <span> element by using the Add method.

Adding Client-Side Script and Event Attributes to the Control

Listing 8.15 shows how you generate the client-side event handler attributes needed for the TextBox and ImageButton controls, and the reference to the client-side script file in the /aspnet_client/custom/ folder that implements these event handlers. This is the same as you did in the user control version of the SpinBox control in Chapter 7, and it follows the same logic as the MaskedEdit server control you built earlier in this chapter. And, of course, the client-side code file you use is the same as you used for the SpinBox control user control.

LISTING 8.15 Adding the Client-Side Code and Event Handlers in the CreateChildControls Method

```
...
' create true/false string for JavaScript code
Dim sAutoPostback As String = "false"
If _autopostback Then
  sAutoPostback = "true"
End If

' create JavaScript parameter string - used to set
' parameters for client-side control event handlers
Dim sParams As String = "'" & sCID & "textbox', " _
  & _minvalue.ToString() & ", " _
  & _maxvalue.ToString() & ", " _
  & _increment.ToString() & ", " _
  & sAutoPostback

' see if previous instance of this control has already
' added the required JavaScript code reference to the page
If Not Page.IsClientScriptBlockRegistered("StnbrmSpinBox") Then
  Dim sPath As String = "/aspnet_client/custom/"
  Dim sScript As String = "<script language='javascript' " _
    & "src='" & sPath & "spinbox.js'><" & "/script>"
  ' add this JavaScript code to the page
  Page.RegisterClientScriptBlock("StnbrmSpinBox", sScript)
End If
```

LISTING 8.15 Continued

```
' set client-side event handlers for controls
oImageUp.Attributes.Add("onclick", _
                        "return incrementValue(" & sParams & ")")
oImageDown.Attributes.Add("onclick", _
                        "return decrementValue(" & sParams & ")")
oTextBox.Attributes.Add("onblur", _
                        "return checkValue(" & sParams & ")")
oTextBox.Attributes.Add("onkeydown", _
                        "return keyDown(event, " & sParams & ")")
...
```

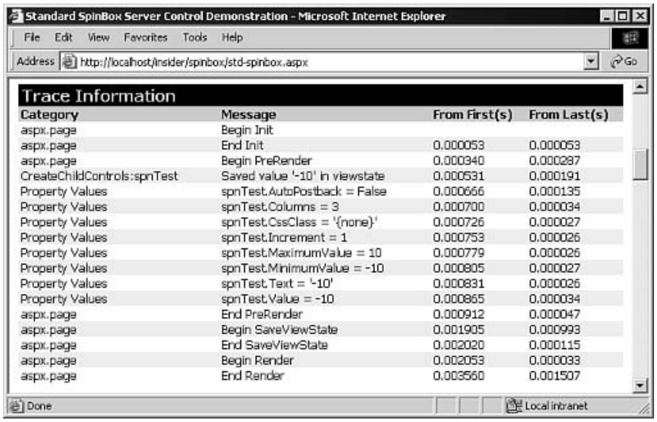
Writing Trace Information in an ASP.NET Page

The final code in the `CreateChildControls` method, shown in Listing 8.16, is there simply to help you understand and debug the control. It writes messages to the `Trace` object for display in the hosting ASP.NET page. You did the same thing earlier, to display the value you save in the viewstate of the current page, and you use exactly the same approach here to display the values of the eight `Public` properties of the `SpinBox` control.

LISTING 8.16 Displaying the Property Values in the `CreateChildControls` Method

```
...
' display control property values in Trace
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".AutoPostback = " & Me.AutoPostback.ToString())
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".Columns = " & Me.Columns.ToString())
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".CssClass = '" & Me.CssClass & "'")
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".Increment = " & Me.Increment.ToString())
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".MaximumValue = " & Me.MaximumValue.ToString())
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".MinimumValue = " & Me.MinimumValue.ToString())
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".Text = '" & Me.Text & "'")
Context.Trace.Write("Property Values", Me.UniqueID _
        & ".Value = " & Me.Value.ToString())

End Sub
```

Figure 8.6 shows the result when the page contains the `Trace="True"` attribute in the `Page` directive. You can see the values for the CreateChildControls and Property Values categories in the list. (The control ID used when you declared the control in the page is `spnTest`.)



Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000053	0.000053
aspx.page	Begin PreRender	0.000340	0.000287
CreateChildControls:spnTest	Saved value '-10' in viewstate	0.000531	0.000191
Property Values	spnTest.AutoPostback = False	0.000666	0.000135
Property Values	spnTest.Columns = 3	0.000700	0.000034
Property Values	spnTest.CssClass = '{none}'	0.000726	0.000027
Property Values	spnTest.Increment = 1	0.000753	0.000026
Property Values	spnTest.MaximumValue = 10	0.000779	0.000026
Property Values	spnTest.MinimumValue = -10	0.000805	0.000027
Property Values	spnTest.Text = '-10'	0.000831	0.000026
Property Values	spnTest.Value = -10	0.000865	0.000034
aspx.page	End PreRender	0.000912	0.000047
aspx.page	Begin SaveViewState	0.001905	0.000993
aspx.page	End SaveViewState	0.002020	0.000115
aspx.page	Begin Render	0.002053	0.000033
aspx.page	End Render	0.003560	0.001507

FIGURE 8.6
The trace information displayed in the hosting ASP.NET page.

The other entries in the list (the category named `aspx.page`) are generated automatically by ASP.NET. You can see from this the ordering of the events and the calls to the standard methods of the page—such as the timing of the `Init`, `PreRender`, `SaveViewState`, and `Render` events—and where the execution of the `CreateChildControls` method occurs.

Declaring and Implementing the `ValueChanged` Event

So far, you've implemented all the features of the `SpinBox` control, with one major exception. You want to expose an event that is raised when the value of the control has changed between the page being served and the subsequent postback. In effect, you want to provide the equivalent of the `TextChanged` event that is exposed by a `TextBox` control. However, because the control is aimed at handling numbers rather than text values, you'll expose an event named `ValueChanged`.

Exposing events is probably the most complex topic related to building server controls, and it usually involves writing several separate routines that interact with the page framework to accomplish this. And although it's relatively easy to expose an event that is raised when something like a button click occurs, it's a little more difficult to expose change events because you have to compare the current and previous values of the control to detect the change. However, this section walks through the whole process.

To expose an event, you have to consider four aspects:

- You have to expose the event as a `Public` event and define the event type. For most events, you can use an existing event (delegate) type such as the standard `EventHandler` type that is exposed by most server controls for click and change events.
- If you want to be able to read the current values of the constituent controls during a postback or detect changes to the values, you must implement the `IPostBackDataHandler` interface. (The `Implements` statement for this is shown in the declaration of the `SpinBox` control class in Listing 8.9.)

- You must provide code that executes the RaiseEvent statement in Visual Basic .NET (in C# you just use the event name) at the appropriate point and pass the parameters required for the event. Usually these parameters are a reference to the current control (Me in Visual Basic .NET or this in C#) and an instance of the appropriate EventArgs class.
- In some cases, you might also have to register for specific page framework events to be passed to the control if you want to react to them. This depends on the type of control you inherit and the event you want to raise.

In the following sections you'll look at each stage of the process by adding code to the SpinBox control class to detect changes to the value in the text box and raise a ValueChanged event.

Exposing the ValueChanged Event

The first step in declaring and implementing the ValueChanged event is to declare the event as a Public event and specify the event object type (actually called a *delegate*). The ValueChanged event is defined using the basic EventHandler type, which is the most standard ASP.NET control used for click and change events:

```
Public Event ValueChanged As EventHandler
```

This allows users of the control to write standard event handler routines, such as the following:

```
Sub MyEventCode(ByVal sender As Object, ByVal e As EventArgs)
  ... code to handle event here ...
End Sub
```

However, you could equally well use a more complex event type, such as the ImageClickEventHandler event that exposes the position of the mouse pointer as the X and Y properties of the corresponding ImageClickEventArgs object. It all depends on what data you want to pass to the user's event handler routine when the event is raised.

Next, you must include code that raises the event at the appropriate time. A common approach is to provide a separate, Protected routine that can be overridden, as shown in Listing 8.17. This means that other developers can override the event themselves when they use the control as a base class for their own controls.

LISTING 8.17 The OnValueChanged Routine for the SpinBox Control

```
Protected OverRidable Sub OnValueChanged(e As EventArgs)

  ' write message to Trace and raise the public ValueChanged
  ' event with appropriate EventArgs values
  Context.Trace.Write("OnValueChanged:" & Me.UniqueID, _
                      "Raising ValueChanged event")
  RaiseEvent ValueChanged(Me, e)

End Sub
```

Notice that you include code that writes to the `Trace` object in this example, as you did in previous routines. Then you simply raise the event and pass a reference to this control (`Me`) and whatever event object was passed to this routine. You can call the `OnValueChanged` routine yourself from elsewhere in the code, when you want to raise the event.

Detecting when the Value of a Control Changes

You now have a `Public` event and a routine that will raise that event. All you have to do is call the `OnValueChanged` routine at the appropriate time. So, next, you need to figure out when the appropriate time is. Usually, you'll want to raise an event when the value of a control within the composite control changes. If the user clicks a button within the control to submit the page, the collection of values posted back to the server will include the values of that element's `name` and `value` attributes (equivalent in Web Forms controls to the `ID` and `Text` property values).

> **The Value Could Be in the Form or the `QueryString` Collection**
>
> **In ASP.NET pages, values are usually posted to the server and appear in the `Request.Form` collection. However, it's possible for the user to set the `action` attribute (property) of the form to `"get"` so that the values appear in the `Request.QueryString` collection.**

> **What Is an Interface?**
>
> **You can think of an interface as being just a list of properties, methods, and events. When your class advertises that it implements a specific interface, other classes can be sure that you are exposing all of these properties, methods, and events, without exception. Interfaces also allow methods to define their parameters in terms of interfaces, so that different classes can be passed to a method, as long as they implement the appropriate interface(s).**

By implementing the `IPostBackDataHandler` interface, you can get access to the values of the control and its child controls within the `Request` collections. You can also read (and write) the viewstate for these controls. This means that you can detect a change event by comparing the value in the viewstate with the value in the `Request` collections for this or any of the constituent child controls. If the user has changed the value in a control, these two values will differ and you can raise the `ValueChanged` event.

Implementing the `IPostBackDataHandler` Interface

To be able to access the posted data, you must implement the `IPostBackDataHandler` interface. Implementing an interface in a class really just means that you must fulfill the "contract" that the interface defines. In other words, you must expose *all* the properties, methods, and events defined for that interface. If you do not, the compiler will refuse to compile the class.

The `IPostBackDataHandler` interface defines just two methods. The first, `LoadPostData`, is executed by the page framework when the data posted from a form is retrieved and made available to the controls on the page. This is the signature:

```
Overridable Function LoadPostData(key As String, _
                     vals As NameValueCollection) _
                     As Boolean
```

A class that handles this event can use the key passed to it (which contains the equivalent of the `name` attribute of the control on the page) to extract the value of this control and any constituent child controls. For example, you can get the value of the child TextBox control within the SpinBox control (which has the ID "textbox") by using the following:

```
Dim NewValue As String = vals(key & "_textbox")
```

You cannot successfully raise an event from the control to indicate that the value has changed during execution of the LoadPostData method. Instead, you must do it at the point when the page framework calls the second method of the IPostBackDataHandler interface—the RaisePostBackDataChangedEvent method. The signature of this method is as follows:

```
Overridable Sub RaisePostBackDataChangedEvent()
```

However, the page framework will not execute this method of the control by default. You have to indicate that you want it to be called by returning the value True from the LoadPostData method.

The SpinBox control class already contains the following statement:

```
Implements IPostBackDataHandler
```

Therefore, now you can add the two methods that are defined in this interface. Listing 8.18 shows these methods. Notice that you have to include an Implements statement in each of the methods to define which method of the IPostBackDataHandler interface the methods are implementing.

LISTING 8.18 Implementing the IPostBackDataHandler Interface

```
Overridable Function LoadPostData(key As String, _
                     vals As NameValueCollection) _
                     As Boolean _
  Implements IPostBackDataHandler.LoadPostData

  ' occurs when data in postback is available to control
  ' get value from postback collection
  Dim NewValue As String = vals(key & "_textbox")

  ' get value from viewstate - i.e. when page was last created
  Dim ExistingValue As String = ViewState(key & "_textbox")
  If NewValue <> ExistingValue Then

    ' value in control has been changed by user
    ' set internal member to posted value and write message
    ' return True so PostDataChangedEvent will be raised
    _text = NewValue
    Context.Trace.Write("LoadPostData:" & key, _
                     "Loaded new value '" & NewValue _
                   & "' from postback data")
    Return True
```

LISTING 8.18 Continued

```
  Else

    ' value in control has not changed
    ' set internal member to viewstate value and write message
    ' return False because no need to raise ValueChanged event
    _text = ExistingValue
    Context.Trace.Write("LoadPostData:" & key, _
                        "Loaded existing value '" & ExistingValue _
                      & "' from viewstate")
    Return False

  End If
End Function

'----------------------------------------------------------

Overridable Sub RaisePostBackDataChangedEvent() _
       Implements IPostBackDataHandler.RaisePostDataChangedEvent

  ' called after all controls have loaded postback data,
  ' but only if LoadPostData handler (above) returned True
  ' call event handler for ValueChanged event
  OnValueChanged(EventArgs.Empty)

End Sub
```

In the `LoadPostData` method, you extract the current value of the `TextBox` control from the posted values. You can see that this is exposed as a `NameValueCollection` type, which is why you had to import the `System.Collections.Specialized` namespace into the class. Then you compare this value with the value for the `TextBox` control in the viewstate of the page. (Recall that you save it into the viewstate each time in the `CreateChildControls` method.)

If the two values are not the same, you want to raise the `ValueChanged` event, so you return `True` from the `LoadPostData` routine to indicate that the page framework should call the `RaisePostBackDataChangedEvent` routine when it's time to raise events from the control. You also set the value of the internal `_text` property to the posted value (it will be used later, in the `CreateChildControls` method, to set the `Text` property of the `TextBox` control) and write a message to the `Trace` object to indicate what's happening within the control. If the values are the same, you don't want to raise the `ValueChanged` event—so you can return `False`. This time you set the `_text` variable to the value held in the viewstate of the page (so that the `TextBox` control maintains its value between postbacks) and write a corresponding message to the `Trace` object as well.

Then, if the page framework does call the RaisePostBackDataChangedEvent routine, all you have to do is raise the ValueChanged event by calling the OnValueChanged routine. Because you don't want to pass back any information about the event, you specify the special value EventArgs.Empty for the event argument parameter. You don't need to provide any information because the value of the control can always be obtained by code in the hosting page from the Text or Value property of the control.

Using a Custom Event Type

If you want to pass information back from an event, you can create a custom event (delegate) type or use one of the other existing event types. Then, when you raise the event, you can create an instance of the appropriate argument's class, fill in the properties, and pass this as the second parameter when you call RaiseEvent.

Registering for Postbacks in a Control

The final issue you might face when raising events is a situation in which the LoadPostData method is not actually called for your control by default. The page framework is clever enough to realize that many control types do not usually allow users to post back changes to the data they contain from a browser. For example, you can't change the value of a <td> element in the browser and have that value sent back in the Request.Form or Request.QueryString collections unless you specifically write custom client-side code to do so.

For these noninteractive controls, the page framework does not call the LoadPostData method automatically. This is the case with the sample SpinBox control because it is based on a <span> element. Therefore, you have to tell the framework that you do want it to call LoadPostData (and subsequently the RaisePostBackDataChangedEvent method). You do this by registering for postbacks.

Listing 8.19 shows the code you use. Notice that you override the Init event of the base class because you have to register for events right at the start of the life cycle of the control. It's no good registering for events in the CreateChildControls method, for example, because the point where LoadPostData is called will already have passed.

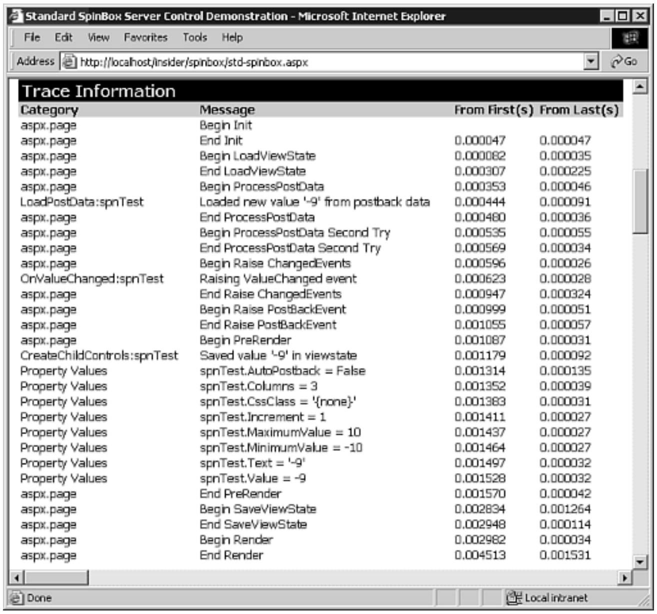
LISTING 8.19 Registering for Postbacks in the Init Event

```
OverRides Protected Sub OnInit(e As EventArgs)

  ' first event that control can handle
  ' must always call base method first
  MyBase.OnInit(e)

  ' must register to receive postback events
  ' required because "root" control is a SPAN
  ' does not receive postback events by default
  Page.RegisterRequiresPostBack(Me)

End Sub
```

All you have to do is call the `RegisterRequiresPostBack` method of the hosting page and pass to it a reference to the control you want to register—in this case, the current control (`Me`). And it's important that you remember to call the `OnInit` method of the base class first.

The Trace Information After a Postback

Putting all the preceding code together, Figure 8.7 shows the trace information that is displayed in the hosting page following a postback where the value of the control has been changed (and when the `Page` directive contains the attribute `Trace="True"`).



Standard SpinBox Server Control Demonstration - Microsoft Internet Explorer

Address http://localhost/insider/spinbox/std-spinbox.aspx

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000047	0.000047
aspx.page	Begin LoadViewState	0.000082	0.000035
aspx.page	End LoadViewState	0.000307	0.000225
aspx.page	Begin ProcessPostData	0.000353	0.000046
LoadPostData:spnTest	Loaded new value '-9' from postback data	0.000444	0.000091
aspx.page	End ProcessPostData	0.000480	0.000036
aspx.page	Begin ProcessPostData Second Try	0.000535	0.000055
aspx.page	End ProcessPostData Second Try	0.000569	0.000034
aspx.page	Begin Raise ChangedEvents	0.000596	0.000026
OnValueChanged:spnTest	Raising ValueChanged event	0.000623	0.000028
aspx.page	End Raise ChangedEvents	0.000947	0.000324
aspx.page	Begin Raise PostBackEvent	0.000999	0.000051
aspx.page	End Raise PostBackEvent	0.001055	0.000057
aspx.page	Begin PreRender	0.001087	0.000031
CreateChildControls:spnTest	Saved value '-9' in viewstate	0.001179	0.000092
Property Values	spnTest.AutoPostback = False	0.001314	0.000135
Property Values	spnTest.Columns = 3	0.001352	0.000039
Property Values	spnTest.CssClass = '{none}'	0.001383	0.000031
Property Values	spnTest.Increment = 1	0.001411	0.000027
Property Values	spnTest.MaximumValue = 10	0.001437	0.000027
Property Values	spnTest.MinimumValue = -10	0.001464	0.000027
Property Values	spnTest.Text = '-9'	0.001497	0.000032
Property Values	spnTest.Value = -9	0.001528	0.000032
aspx.page	End PreRender	0.001570	0.000042
aspx.page	Begin SaveViewState	0.002834	0.001264
aspx.page	End SaveViewState	0.002948	0.000114
aspx.page	Begin Render	0.002982	0.000034
aspx.page	End Render	0.004513	0.001531

Done Local intranet

FIGURE 8.7
The trace information from the `SpinBox` control after a postback when the value has changed.

As well as the CreateChildControls and Property Values category entries you saw earlier (refer to Figure 8.6), you can see entries for categories named LoadPostData and OnValueChanged. These show the value that was extracted from the posted data sent back in the `Request` collection and the `ValueChanged` event being raised. If you force a postback without changing the value of the `SpinBox` control (by just clicking the Apply button in the page), you'll see that the `LoadPostData` category entry is "Loaded existing value '-9' from viewstate."

Using an Adaptive `SpinBox` Control

Using the `SpinBox` control simply involves registering it in the page and then declaring an instance of it—just as you did with the `MaskedEdit` control earlier in this chapter. This is the `Register` directive:

```
<%@ Register TagPrefix="ahh" Namespace="Stonebroom"
             Assembly="std-spinbox" %>
```

You can declare an instance of the control by using something like this:

```
My SpinBox Control:
<ahh:StandardSpinBox id="spnTest" OnValueChanged="SpinValueChanged"
     AutoPostback="False" Columns="3" Increment="10"
     MinimumValue="0" MaximumValue="100" runat="server" />
```

And, of course, you can handle the ValueChanged event as you would any other control. The routine SpinValueChanged that is defined for the ValueChange event in the preceding code might look like this:

```
Sub SpinValueChanged(sender As Object, e As EventArgs)
  lblResult.Text &= "Detected ValueChanged event for control " _
                 & sender.ID & ". New value is " _
                 & sender.Text & "<br />"
End Sub
```

Making the SpinBox Control Adaptive

The SpinBox control works well in Internet Explorer, but what about in other browsers? Often, you get a shock when you develop and test with just one browser and then view the results in a different browser—and the SpinBox control is no exception. Figure 8.8 shows the result of using this control in Opera 7.21, and you can see that it looks and works just as it does in Internet Explorer (other than the lack of the pop-up ToolTips for the up and down buttons).

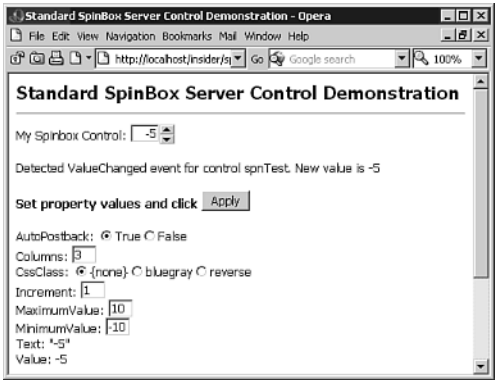


FIGURE 8.8 The SpinBox control demonstration page in Opera 7.21.

However, in Mozilla 1.5 you have a problem. The display looks fine, but the up and down buttons don't respond to mouse clicks at all. The cursor doesn't even change to a hand when it's over these buttons (as shown in Figure 8.9). The text box works fine, and it reacts to keypress events. However, there is obviously some major problem.

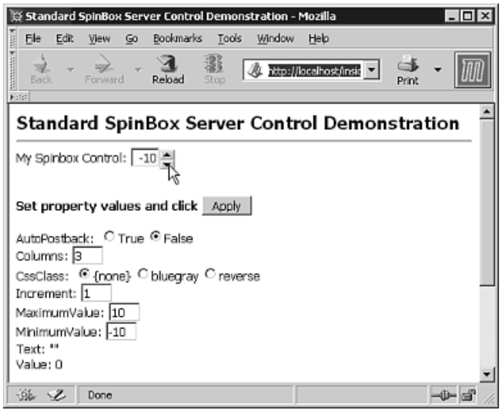


FIGURE 8.9 The `SpinBox` control demonstration page in Mozilla 1.5.

Trying an older browser is even worse. In Netscape Navigator 4.5, the text box doesn't even appear, and the up and down buttons aren't properly aligned and cannot be clicked (see Figure 8.10). The control is useless here.

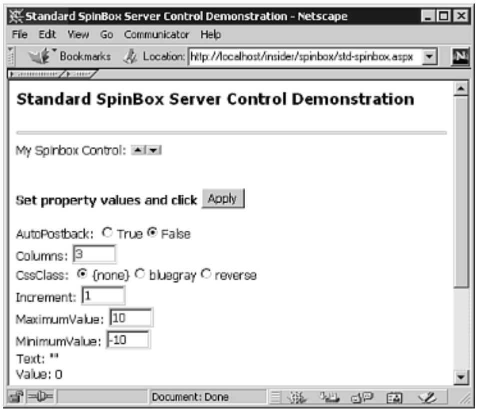


FIGURE 8.10 The `SpinBox` control demonstration page in Netscape Navigator 4.5.

As a final example, you can try the page in the W3C reference browser called Amaya. This is a great way to see if your pages comply with the rules and recommendations of HTML and CSS. In fact, Amaya reports only a couple minor errors in the page—for example, that the `<input type="image">` element you use for the up and down buttons does not support the `border="0"` attribute. Remember that you had to force ASP.NET to add this to the `ImageButton` controls to prevent some older browsers from displaying borders for them.

Figure 8.11 shows the page in Amaya. The font in the text boxes is wrong, causing them to wrap, but everything seems to be there. However, because Amaya doesn't support client-side scripting, the up and down buttons and keypresses don't work. However, you can change the control's value can if you type into the text box and click the Apply button.

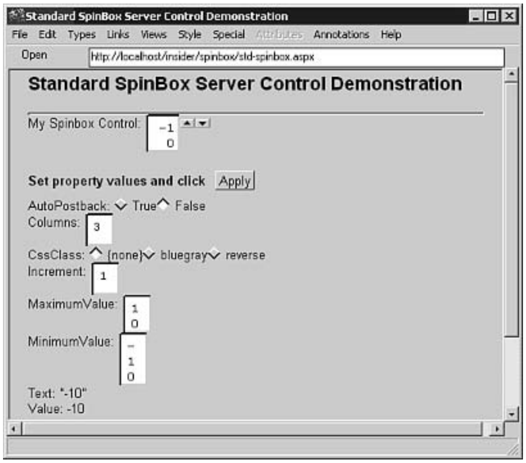


FIGURE 8.11 The SpinBox control demonstration page in Amaya.

Coping with Older and Nonstandard Browsers

Before you can fix the problems that arise when the sample page is used in browsers other than Internet Explorer, you need to figure out why the control doesn't work in these problem browsers. The main issue is that you've used CSS2 absolute positioning and DOM 2.0 scripting techniques (such as the `getElementById` method) within the control. This means that it won't work properly on older browsers, such as Netscape 4.5, and browsers that don't support client-side scripting, such as Amaya.

The secondary issue concerns Mozilla (and later versions of Netscape that use the Mozilla rendering engine). Some features of CSS2 are not fully defined in the W3C recommendations for all possible scenarios. This means things might not work in one browser that work in another, even though both browsers supposedly support current standards.

If you experimented with the `ComboBox` control built in Chapter 5, you might have found that it doesn't work quite as expected in Mozilla. The W3C CSS2 recommendations don't define exactly what should happen to the z-order of controls that are absolutely positioned when they are shown or hidden dynamically. In Mozilla, this results in the drop-down list appearing behind the existing text boxes (see Figure 8.12), whereas it appears in front of them in Internet Explorer 6.0 (and they appear in front of each other, depending on the order in which they are opened).

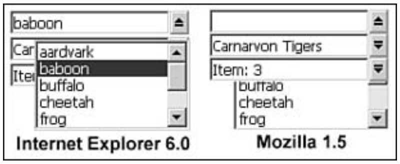


FIGURE 8.12 Problems with the ComboBox control in Mozilla 1.5.

> **The Drop-Down Lists in Internet Explorer**
> Interestingly, in Internet Explorer 5.5, the drop-down lists open on top of the text boxes, but not always on top of each other in the correct order—depending on the order in which they are opened.

The problem you have with Mozilla 1.5 and the SpinBox control is also related to the specifications of CSS2 not being totally comprehensive. Recall that the structure generated for the SpinBox control (see Figure 8.13) is a root `<span>` element that is relatively positioned. The contained TextBox control is not positioned (it simply appears in the flow of the page within the `<span>` control). However, the two ImageButton controls carry the `position:absolute` selectors so that they will be located at the right side of the TextBox control.

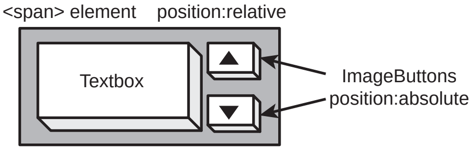


FIGURE 8.13 The structure of the standard SpinBox control.

What happens is that the ImageButton controls (which are rendered as `<input type="image">` elements in the page) are removed from the flow of the page by the `position:absolute` selectors. This means that the `<span>` element is only sized to contain the TextBox control, so the two ImageButton controls lie outside the `<span>` element in terms of location—even though they are still child controls.

Internet Explorer and Opera take into account the control hierarchy, and the buttons work fine. However, Mozilla does not consider the buttons to be part of the rendered page as far as the mouse pointer is concerned, and it ignores mouse clicks on them. But if you place the cursor on the text box and press the Tab key, you do in fact move the focus to them and can click them by pressing the Spacebar.

Creating an Alternative Structure for the SpinBox Control

One solution for the various problems with the SpinBox control is to offer an alternative structure for the controls that provides wider support for older browsers. The obvious approach is to use an HTML table to locate the TextBox and ImageButton controls. But this leads to another problem.

The reason you used a `<span>` element in the first place was so that the control could be used like a TextBox control or other standard controls within the flow layout of the page. For example, the user should be able to place a text caption to the left and more content after it, without causing the caption or the following content to wrap to a new line. If you use an HTML table to locate the constituent controls, it will cause preceding and following content to wrap, forcing the user to insert the whole lot into an HTML table (or use absolute positioning) to get the layout desired.

Another possibility is to use a `<div>` element as the root control for the SpinBox control, but this has the same problem as using an HTML table. In the end, this example uses the HTML table but adds an extra cell to the left, where you insert a user-supplied value for the caption (see Figure 8.14). It's not ideal because preceding and following content will still wrap, but at least the caption will appear in the correct position. And it seems to be the only solution for older browsers.

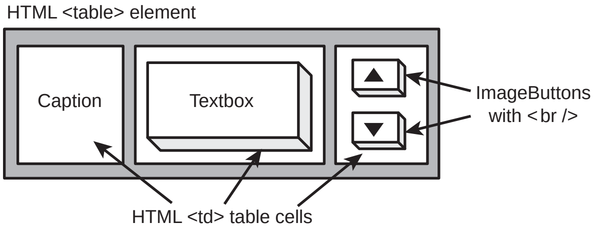


FIGURE 8.14
The structure of the adaptive `SpinBox` control for older browsers.

To maintain the interface and behavior of the control across all browser types, you need to support the caption in more recent browsers that work with the up-level version of the control. You can expose the caption as a property of the control, and if the user sets this property, he or she will expect to see it appear in all browsers. Figure 8.15 shows the updated structure of the `SpinBox` control for these newer browser types.

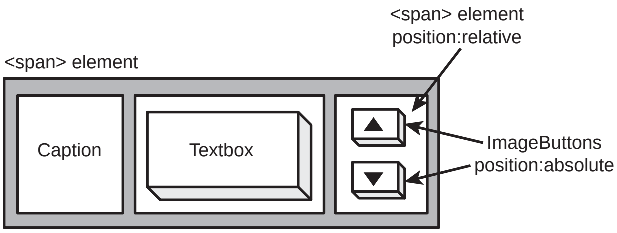


FIGURE 8.15
The structure of the adaptive `SpinBox` control for more recent browsers.

Adaptability Changes to the `SpinBox` Control Class

The following sections briefly review the changes required in the `SpinBox` control to implement the dual behavior for up-level and down-level clients. When you look at the `CreateChildControls` method, you'll see how you decide what output to send to each type of browser.

Changes to the `Private` and `Public` Declarations

You need to make a couple minor changes to the variable and property declarations of the `SpinBox` control. You must import the `System.Web.UI.HtmlControls` namespace because you're using the `HtmlGenericControl` class that it defines to create the nested `<span>` element for the up-level version of the control. You also use a different class name this time (`AdaptiveSpinBox`).

You can add an enumeration to the control to define the "modes" it can run in. This allows a user to specify, for example, down-level behavior, even if their browser supports the up-level features:

```
' enumeration of target browser types
Public Enum ClientTargetType
  AutoDetect = 0
```

```
  UpLevel = 1
  DownLevel = 2
End Enum
```

You also need a few more internal variables and the property declarations for the two new properties `Caption` and `ClientTarget`. The first two internal variables, `_usetable` and `_usecss2`, default to `False` and are used in other routines within the control to manage the type of output you send to the client. Notice that the `ClientTarget` property is read-only and is defined as a value from the `ClientTargetType` enumeration. The internal `_client` variable that shadows the value of the `ClientTarget` property sets the default to `AutoDetect` (see Listing 8.20).

LISTING 8.20 Registering for Postbacks in the `Init` Event

```
Private _usetable As Boolean = True
Private _usecss2 As Boolean = False
Private _caption As String = ""
Private _client As ClientTargetType = ClientTargetType.AutoDetect
Public Property Caption As String
  Get
    Return _caption
  End Get
  Set
    _caption = value
  End Set
End Property

Public WriteOnly Property ClientTarget As ClientTargetType
  Set
    _client = value
  End Set
End Property
```

Changes to the `CreateChildControls` Method

The largest number of changes occur in the `CreateChildControls` method, where you generate the control tree for the `SpinBox` control. In it, you add code that uses the ASP.NET `BrowserCapabilities` object (which you met in Chapter 7) to detect the current browser type and decide what features it supports.

Listing 8.21 assumes that the client is a down-level device and then checks whether it supports JavaScript. If it does not, there's no point in generating the interactive version of the control that uses CSS2 scripting. If JavaScript is supported, you can use the browser name and major version number to decide what to do next. Notice that for Internet Explorer 5 and higher, and for Opera 6 and higher, you specify that it's an up-level device and that you'll use CSS2 scripting, but you will not generate an HTML table.

LISTING 8.21 Detecting the Browser Type and Capabilities

```
...
' check if the current browser supports features
' required for "smart" operation and if user specified
' the mode they want (Version6 or Downlevel)
If _client <> ClientTargetType.DownLevel Then

  ' start by assuming DownLevel
  _client = ClientTargetType.DownLevel

  ' get reference to BrowserCapabilities object
  Dim oBrowser As HttpBrowserCapabilities = Context.Request.Browser

  ' must support client-side JavaScript
  If oBrowser("JavaScript") = True Then

    ' get browser type and version
    Dim sUAType As String = oBrowser("Browser")
    Dim sUAVer As String = oBrowser("MajorVersion")

    ' see if the current client is IE5 or above
    If (sUAType = "IE") And (sUAVer >= 5) Then
      _client = ClientTargetType.UpLevel
      _usetable = False
      _usecss2 = True
    End If

    ' see if the current client is Netscape 6.0/Mozilla 1.0
    If (sUAType = "Netscape") And (sUAVer >= 5)  Then
      _client = ClientTargetType.UpLevel
      _usetable = True
      _usecss2 = True
    End If

    ' see if the current client is Opera 6.0
    If (sUAType = "Opera" And sUAVer >= 6) Then
      _client = ClientTargetType.UpLevel
      _usetable = False
      _usecss2 = True
    End If

  End If

End If
```

LISTING 8.21 Continued

```
' save current value of _client in viewstate
ViewState(sCID & "target") = _client.ToString()

' display detected client type value in Trace
Context.Trace.Write("CreateChildControls:" & Me.UniqueID, _
        "Saved target '" & _client.ToString() & "' in viewstate")
...
```

The odd ones out as far as browsers go are Netscape and Mozilla. If the current browser is Netscape or Mozilla, with a version number of 5 or higher (which actually equates to Netscape 6.0 and Mozilla 1.0), it is up-level, and you can use CSS2 scripting. However, due to the problem with the `<span>` element and the absolute-positioned `ImageButton` controls shown earlier, you have to generate the structure of the control as an HTML table. It will still be interactive because you'll inject the client-side script and add the client-side event handlers.

You also need to save the client target value (the value of the `_client` variable) in the viewstate of the page so that you can extract it next time. This is a property of the control that users will expect to be maintained across postbacks. If they have set it to `DownLevel`, they won't expect the code to perform the detection again after each postback and reset the value.

Creating Browser-Specific Output

Now you can build the control tree needed. To make it easier to manage, the tasks required to create the control output have been separated into three routines:

- **`CreateCSS2Controls`**—This routine creates basically the same control tree as the standard version of the `SpinBox` control you saw earlier in this chapter. The only differences are that the root `<span>` control is no longer relative positioned, and it contains the caption text and the nested `<span>` control that is relative positioned (refer to Figure 8.14 for more details).
- **`CreateHTMLTable`**—This routine creates the control structure shown in Figure 8.13. This is the HTML table version, consisting of three cells that contain the caption, the text box, and the two image buttons. One interesting point here is that you have to use a `LiteralControl` instance to create the `<br />` element that is required to wrap the second `ImageButton` under the first one in the right-hand cell. If you use an `HtmlGenericControl` instance, you actually get the string `"<br></br>"`, which causes most browsers to insert *two* line breaks.
- **`InjectClientScript`**—This routine uses exactly the same code that is used in the standard version of the `SpinBox` control to generate the `<script>` element that references the client-side script file for the control (which must be located in the `/aspnet_client/custom/` folder of the Web site). It also adds the client-side event handler attributes to the `TextBox` control and the two `ImageButton` controls.

We don't describe the three routines in detail here because they are generally repetitive and do not introduce anything new to the discussion. You can view the source code to see these routines. (Remember that each sample contains a [view source] link at the foot of the page. See www.daveandal.net/books/6744/.)

Listing 8.22 shows the next section of the CreateChildControls method, where the _usetable and _usecss2 variables are used to decide which of the three routines just described are executed. The result is that the control generates output that is suitable for the current browser and provides the best possible support it can, depending on the features of that browser. Next, although not shown in Listing 8.22, the values of the properties are displayed in the Trace object in exactly the same way as in the standard SpinBox control example.

LISTING 8.22 Creating the Appropriate Control Tree

```
...
' now ready to create the appropriate set of controls
If _usetable = False Then

  ' serving to version-6 client, use absolute positioning
  ' (but not for Netscape 6.x or Mozilla 1.x)
  CreateCSS2Controls()

Else

  ' serving to down-level client, create HTML table
  ' (including Netscape 6.x or Mozilla 1.x)
  CreateHTMLTable()

End If

If _usecss2 = True Then

  ' serving to client that supports CSS2 so inject script
  InjectClientScript()

End If
...
```

Changes to the LoadPostData Method

For the SpinBox control example, the only other changes required to provide behavior that adapts to different clients are to the code in the LoadPostData routine. You have to extract the value from the postback and compare it to the existing value of the control, as stored in the viewstate of the page. If these two values differ from one another, you raise the ValueChanged event. If they are the same, you use the existing value from the viewstate to populate the control.

The issue with the adaptive control is that, in down-level clients, clicking the up and down buttons does not automatically change the value in the text box—because there is no client-side script to do that. Such clicks will always cause postbacks to the server. So you have to check for a click on either of the two `ImageButton` controls, and you have to see if the value in the text box has been changed.

Listing 8.23 shows the `LoadPostData` method. After it extracts the value for the text box from the postback collection, it gets the value when the page was originally created from the viewstate and the value of the client target type. (Both of these values are saved in the viewstate in the `CreateChildControls` method.)

LISTING 8.23 The `LoadPostData` Method in the Adaptive `SpinBox` Control

```
Overridable Function LoadPostData(key As String, _
                     vals As NameValueCollection) _
                     As Boolean _
  Implements IPostBackDataHandler.LoadPostData

  ' occurs when data in postback is available to control

  ' get value of control from postback collection
  Dim sNewValue As String = vals(key & "_textbox")
  Context.Trace.Write("LoadPostData:" & key, _
    "Loaded postback value '" & sNewValue & "' from Request")

  ' get value from viewstate - i.e. when page was last created
  Dim sExistingValue As String = ViewState(key & "_textbox")
  Context.Trace.Write("LoadPostData:" & key, _
    "Loaded existing value '" & sExistingValue & "' from viewstate")

  ' get client target type from viewstate
  Dim sClientType As String = ViewState(key & "_target")
  Context.Trace.Write("LoadPostData:" & key, _
    "Loaded target '" & sClientType & "' from viewstate")

  If (sClientType = ClientTargetType.UpLevel.ToString()) _
  Or (sNewValue <> sExistingValue) Then

    ' either client type is "UpLevel" and value was
    ' incremented by client-side script, or user typed
    ' new value in Textbox in "DownLevel" client

    If sNewValue <> sExistingValue Then

      ' value in control has been changed by user
      ' set internal member to posted value and return True
      ' so that PostDataChangedEvent will be raised
      _text = sNewValue
```

LISTING 8.23 Continued

```
      Return True

    Else

      ' value in control has not changed
      ' set internal member to viewstate value and write message
      ' return False because no need to raise ValueChanged event
      _text = sExistingValue
      Return False

    End If

  Else

    ' client type may be "DownLevel" and value was not incremented
    ' so check if "up" or "down" button caused the postback
    If vals(key & "_imageup.x") <> "" Then

      ' "up" image button was clicked so increment value
      ' new value will be checked in CreateChildControls event
      ' to ensure its within maximum and minimum value limits
      ' use Try...Catch in case viewstate empty or text not a number
      Try
        _text = CType(Int32.Parse(sExistingValue) + _increment, _
                      String)
        Context.Trace.Write("LoadPostData:" & key, _
                            "Incremented value to '" & _text)
      Catch
        Context.Trace.Write("LoadPostData:" & key, _
          "Error reading viewstate: " & sExistingValue)
      End Try
      ' return True so that PostDataChangedEvent will be raised
      Return True

    End If

    If vals(key & "_imagedown.x") <> "" Then

      ' "down" image button was clicked so decrement value
      Try
        _text = CType(Int32.Parse(sExistingValue) - _increment, _
                      String)
        Context.Trace.Write("LoadPostData:" & key, _
          "Decremented value to '" & _text)
```

LISTING 8.23 Continued

```
      Catch
        Context.Trace.Write("LoadPostData:" & key, _
          "Error reading viewstate: " & sExistingValue)
      End Try
      ' return True so that PostDataChangedEvent will be raised
      Return True

    End If

  End If

End Function
```

Then you can see if this is an up-level client or if the value of the text box has been changed. Remember that for down-level clients, the user could have typed a new value into the text box and then submitted the page. If the value has changed, you save it in the internal `_text` variable and return `True` to indicate that you want the page framework to call the `RaisePostBackDataChangedEvent` method, where you'll raise the `ValueChanged` event.

If the text box value has not changed, you must check whether the user submitted the page from a down-level client by clicking the up or down button. You can detect whether one of these buttons was clicked by looking for its value in the postback collection. `ImageButton` controls send the x and y coordinates of the mouse pointer within the image when they are clicked, or they send zero for both coordinates when the spacebar is used to click the image. All you have to do is try to increment or decrement the current value (stored in the `_text` variable) by the current value of the `Increment` property (stored in the `_increment` variable) and return `True` to cause the `ValueChanged` event to be raised.

If you turn on tracing for the page and initiate a postback by clicking the up or down button, you'll see the messages that the code writes to the `Trace` object. In Figure 8.16, you can see the values in the postback collection and the viewstate being loaded, and you can see the `ValueChanged` event being raised. You can also see the points at which the value and the client target type are saved back into the viewstate and the values of the other properties of the control.

Testing and Using an Adaptive `SpinBox` Control

The demonstration page for the adaptive `SpinBox` control that is provided with the samples for this book is just about identical to the one shown for the standard `SpinBox` control earlier in this chapter. The page allows the new `Caption` property to be set and shows that caption next to the control. Of course, the classname is different this time, so the `Register` directive looks like this:

```
<%@ Register TagPrefix="ahh" Namespace="Stonebroom"
             Assembly="adaptive-spinbox" %>
```

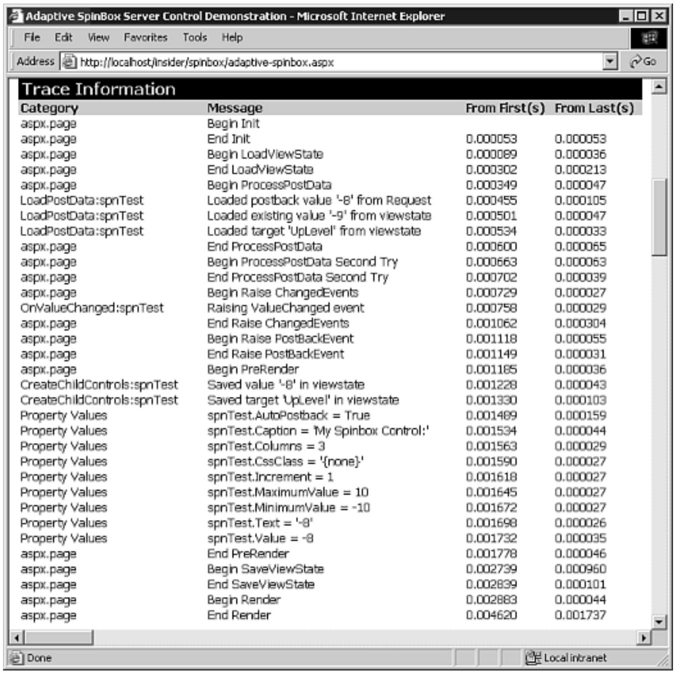


FIGURE 8.16 The trace output from the adaptive SpinBox control following a postback.

The adaptive version of the SpinBox control looks and behaves the same in Internet Explorer and Opera as the standard version does. However, it now works in other browsers as well. For example, Figure 8.17 shows it in Mozilla, where the up and down buttons now work as expected.

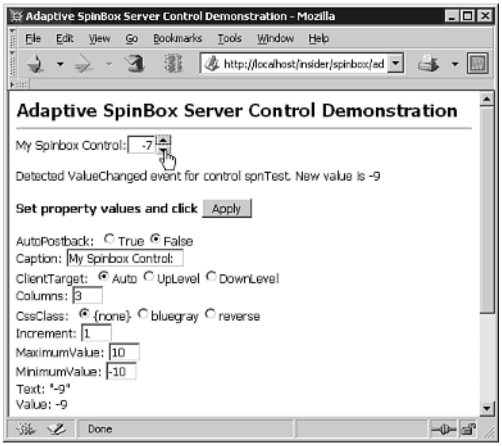


FIGURE 8.17 The adaptive SpinBox control in Mozilla 1.5.

Figure 8.18 shows the adaptive SpinBox control demonstration page in Netscape Navigator 4.5. The original version of the control fails to show the text box or position the up and down buttons correctly in this browser—but the adaptive version works as it should.

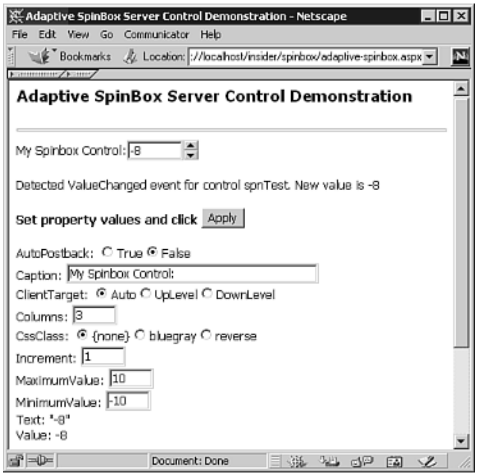


FIGURE 8.18 The adaptive `SpinBox` control in Netscape Navigator 4.5.

Finally, in Amaya, the standard version of the `SpinBox` control fails to work at all, even though it displays okay. The modifications in the adaptive version allow it to operate without requiring client-side script, and the result (shown in Figure 8.19) is that it is completely usable in Amaya.

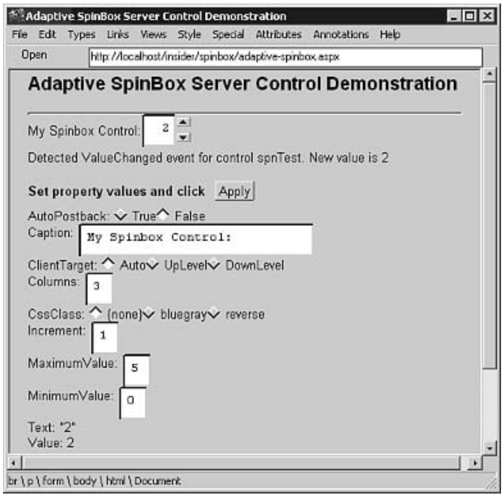


FIGURE 8.19 The adaptive `SpinBox` control in Amaya.

Installing a `SpinBox` Control in the GAC

To end this chapter, you'll adapt the `SpinBox` control so that it can be placed in the GAC, and you'll follow the steps required to achieve this. You need to make some minor changes to the class file to allow it to be registered in the GAC. Then you just have to create a key pair for the class file, compile it, and install it in the GAC.

Changes to the SpinBox Control Class File for GAC Installation

In order for the assembly that is generated when you compile the `SpinBox` control class to be registered in the GAC, it has to contain version information. You achieve this by adding attributes that specify (at a minimum) the location of the key pair file that will be used to digitally sign the assembly and the assembly version to the class. These attributes are defined in the `System.Reflection` namespace, so you must import that namespace into the class first:

```
Imports System.Reflection
```

The following are the two required attributes, which are added before the `Namespace` declaration:

```
<assembly:AssemblyKeyFileAttribute("GACSpinBox.snk")>
<assembly:AssemblyVersionAttribute("1.0.0.0")>
Namespace Stonebroom
  Public Class GACSpinBox
  ...
```

In this example, the key pair file is named `GACSpinBox.snk`, and it is located in the same folder as the class file. This class is also declared as being version 1.0.0.0.

> **Adding Other Attributes to a Class**
>
> **You can add plenty of other attributes to an assembly. You can specify your company name, copyright statement, product name, description, and culture information. Look at the topic "System.Reflection Namespace" in the Reference, Class Library section of the SDK for a full list of attributes and a description of each one.**

Compiling the SpinBox Control Class File

The remainder of the `SpinBox` control class file is identical to the adaptive `SpinBox` control you just built. The only changes you have to make are those shown in the preceding section. The next step is to create the key pair file referenced in `AssemblyKeyFileAttribute`. The `sn.exe` utility provided with the .NET Framework does this for you. You can run a batch file named `createkey.bat` (included in the samples you can download from `www.daveandal.net/books/6744/`) in a command window when the current folder contains the source class file. The following command is required:

```
"C:\Program Files\Microsoft.NET\SDK\v1.1\Bin\sn" -k GACSpinBox.snk
```

Notice that you provide the full path to the `sn.exe` utility to make sure that you use the correct version if you have more than one version of the .NET Framework installed. If all is well, you'll see the response "Key pair written to GACSpinBox.snk."

Now you can compile the class file in the usual way. The batch file `make.bat` (also in the samples you can download from `www.daveandal.net/books/6744/`) does this for you, by executing the following command:

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322\vbc /t:library
➥/out:GACSpinBox.dll /r:System.dll,System.Web.dll gac-spinbox.vb
```

Installing the `SpinBox` Assembly into the GAC

After you compile the class file, you install the assembly into the GAC. The batch file named `addtogac.bat` (in the samples you can download from `www.daveandal.net/books/6744/`) contains the command required:

```
"C:\Program Files\Microsoft.NET\SDK\v1.1\Bin\gacutil" /i GACSpinBox.dll
```

If all goes well, you'll see the message "Assembly successfully added to the cache."

> **Listing and Removing the Assembly from the GAC**
>
> **The samples for this book, which you can download from `www.daveandal.net/books/6744/`, also contain batch files that remove the assembly from the GAC (`removefromgac.bat`) and list the contents of the GAC (`viewgac.bat`).**

The alternative to using the command-line `gacutil.exe` utility is to run the .NET Configuration program provided with the .NET Framework. To do this, you select Start, Programs, Administrative Tools and then select Microsoft .NET Framework 1.1 Configuration. This useful program provides access to many features of the .NET Framework, including the GAC (shown as Assembly Cache in the left tree-view window).

To add an assembly, you simply right-click the Assembly Cache entry in the left window of the .NET Configuration tool and select Add; then you locate the assembly. In Figure 8.20, the assembly DLL has been copied into the `Framework\v1.1.4322` (version 1.1) folder of the `Winnt\Microsoft.NET\` folder tree.

FIGURE 8.20
Adding an assembly to the GAC with the .NET Configuration tool.

After the assembly is installed, either through the command-line utility or with the .NET Configuration tool, you'll see the assembly in the list of installed assemblies on the right. If you right-click it and select Properties, as shown in Figure 8.21, you can see the assembly name and version, the location, the public key token, any culture details, and other information. You can also use the context menu to remove the assembly from the GAC.

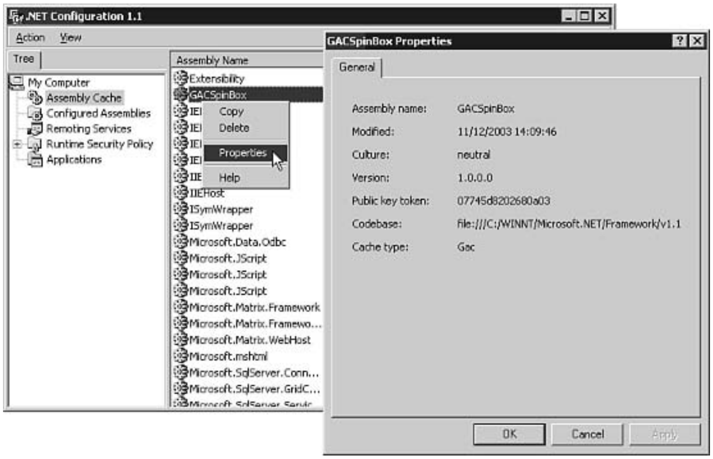


FIGURE 8.21
Viewing details of an assembly in the GAC with the .NET Configuration tool.

Testing the GAC-Installed Control

After you have installed the assembly for the SpinBox control in the GAC, you can use it in an ASP.NET page. The demonstration page provided for this is identical to the one for the adaptive version of the control, with the exception of the Register directive. To register an assembly that is in the GAC, you have to provide the fully qualified name rather than just the assembly name. In other words, you have to specify the version, the culture details, and the public key token of the assembly you want to use, as in the following example:

```
<%@Register TagPrefix="ahh" Namespace="Stonebroom"
            Assembly="GACSpinBox,Version=1.0.0.0,Culture=neutral,
            ➥PublicKeyToken=07745d8202680a03" %>
```

This is how the .NET Framework supports multiple versions and allows each application to specify the version of the control or assembly it wants to use. And if the assembly has been changed (perhaps a malicious or tampered version is installed), the public key token will not match the hash value calculated for the assembly at runtime, and that will prevent the code from running and protecting the application.

Instead of declaring the fully qualified name in every page, you can add the assembly to the default set for ASP.NET by declaring it in the <assemblies> section of machine.config or web.config:

```
<system.web>
  <compilation>
    <assemblies>
      <add assembly="GACSpinBox,Version=1.0.0.0,Culture=neutral,
                     ➥PublicKeyToken=744000b7e77ec1a6"/>
    </assemblies>
  <compilation debug="false" explicit="true" defaultLanguage="vb">
</system.web>
```

Then your ASP.NET pages can use the simple `Register` directive:

```
<%@Register TagPrefix="ahh" Namespace="Stonebroom"
          Assembly="GACSpinBox" %>
```

Now, if the version or public key token of the assembly is changed, you don't have to update every page. You only have to change the entry in the corresponding `machine.config` or `web.config` file.

Summary

This chapter focuses on what is generally considered the best way to create reusable content, in the form of controls that provide a user interface or methods you can use in multiple pages, applications, and Web sites. Building server controls and compiling them into an assembly is not nearly as simple as building user controls, but it does open up opportunities that aren't available with user controls. For example, with a server control you can do the following:

- Hide the implementation from the user in a far more comprehensive manner than with user controls.
- Easily raise events that can be handled in the hosting page just like the events of the standard ASP.NET controls.
- Install the controls in the GAC so that they are available to any application running on the machine.

This chapter looks at the basic issues involved in building server controls, including the choice of base classes to inherit from and the different approaches to the design of simple and composite controls. Also covered are how you can generate output directly during the rendering phase of a control's life cycle and how you can build a control tree and allow the .NET Framework to look after rendering it instead.

This chapter also demonstrates the building of two different server controls—the simple `MaskedEdit` control and the composite `SpinBox` control. These two controls demonstrate the techniques that are involved, the methods you can override to generate your own output, and the way that events can be raised from a control.

In this chapter you have learned how custom controls might behave in a range of browsers, and you discovered that in most cases it's necessary to build in some kind of adaptive behavior so that a control generates different output, depending on the current browser. You did this with the `SpinBox` control and demonstrated it working in several quite different types of browsers.

To finish off, you looked at how you can adapt controls so that you can install them into the GAC and use them in any application on the machine. As you have seen, this isn't difficult to do, and it does make it easier to maintain and update a control when (unlike with user controls) you have only one copy installed.

9

Page Templates

This chapter is all about site design—not in the “how to make it look good” way but in the “how to make it consistent” way. One of the problems you face when building a site is ensuring that all pages of the site look and perform in a similar manner. Consistency is a key goal in building any application, and given that Web sites are far reaching and liable to be used by people of all abilities, consistency is especially important here.

This chapter shows several solutions for building consistency into a site. It focuses on the solutions you can use to allow all pages (if that’s what you require) to look the same. The aim is to make Web site development easier and more maintainable—not only for adding features or fixing bugs but also for site redesigns.

IN THIS CHAPTER

Designing for Consistency

When you create a Web site, there are areas that often need to look the same across the whole site: corporate logo, menus, areas for user login, and so on. The problem you face is how to create this structure so that you gain consistency across pages without losing the ease of development that ASP.NET brings. What you want is the master pages scenario that ASP.NET 2.0 provides, but for ASP.NET 1.1. Master pages give you the ability to use some sort of template to define the content that should appear on all pages, and at runtime this content is combined with the content on individual pages, as shown in Figure 9.1.

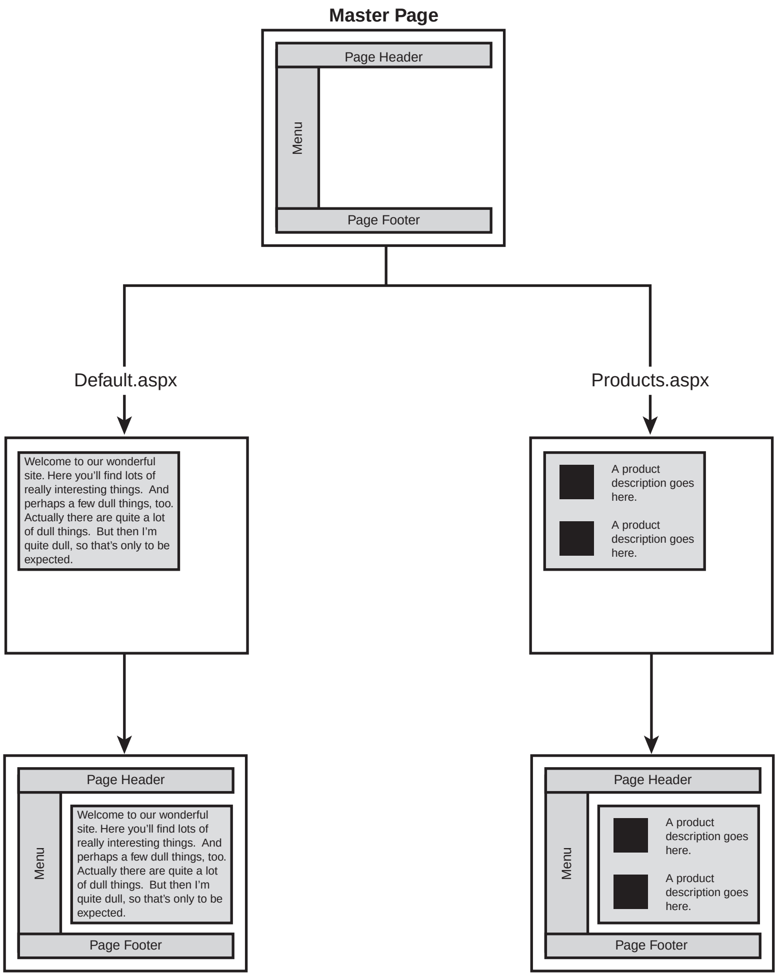


FIGURE 9.1
Combining a master template page with content pages.

Unfortunately, ASP.NET 1.1 has no built-in support for master pages, so you have to build a solution yourself. The simplest way is to define a site layout and simply enforce it—tell your developers "this is what it must look like" and then check it when it's done. It's not a very high-tech solution, but it works. However, this method is rather labor intensive as well as error prone—it's easy to leave something out or make a simple mistake in the layout. It also means a lot of work because the parts of the site that are the same have to be coded onto each page.

To get around this repetitive use of code and content, some form of template is needed. It's easy enough to create a template ASP.NET page that becomes the starting point for all other pages: You just copy it and rename it for the new page and implement the new content. However, this method still leaves lots of repeated content, which is particularly bad if you need to redesign the site. A common way around this is to use include files or user controls to define the regions of a page that should be the same. This way, you have reusable content that can be simply included on every page. You still need to ensure that the include files or user controls are actually included on the page, and it's possible for different controls to be placed on the page or in different areas of the page.

Templating Solutions

The two best ways to provide reusable content and consistency are to use a custom server control or a custom page class. With a custom server control, you still face the drawback of a control being required on each page, but you can use that control to provide all the mandatory content. A custom server control is easy for developers to use because all they need to do is drop it onto a page. However, it lacks really good designer support—you can create a custom designer, but there are issues, which we'll look at later in this chapter in the section "Using Custom Controls in Visual Studio .NET."

Using a custom page class is similar to using a custom control, but it doesn't require the addition of a custom control to the page; the underlying class provides the mandatory content. This isn't a perfect solution—again, it lacks designer support, and it requires a few changes to page classes created by Visual Studio .NET.

The following sections look at how you can implement custom user controls and custom page classes. In the process, you'll see how to add support in Visual Studio .NET.

A Simple Layout Server Control

Using a server control as a template is fairly easy. The process of creating custom server controls seems very scary to many developers, but it's actually a fairly simple process. The aim is to have a control that outputs all the mandatory content but that has an area where customized content can be added. Such a control might look something like this:

```
<sams:MasterPageControl runat="server" id="tctl1">
    Server controls and page content can go here
</sams:MasterPageControl>
```

You can simply drop this control onto every page and add the controls for the page within the `MasterPageControl` tags. `MasterPageControl` will output all the default content for the page.

Before you create a control like this, you first have to understand a bit about the control life cycle, and Figure 9.2 shows the methods called during the various phases of the life of a control.

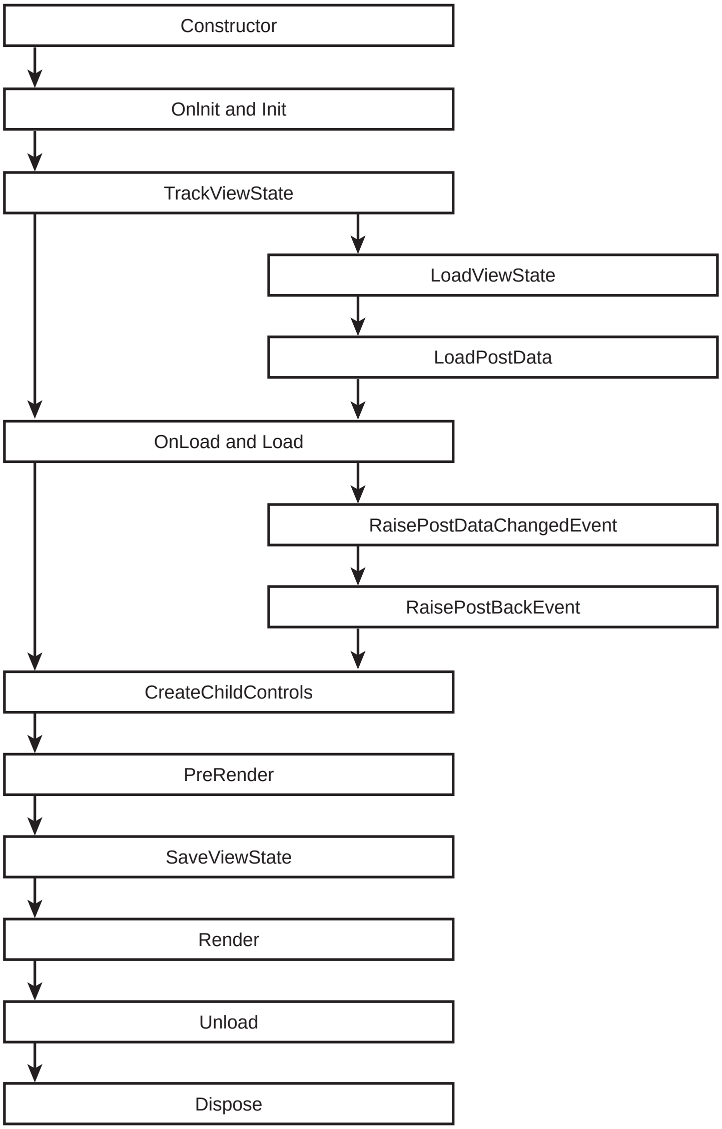


FIGURE 9.2
A control's life cycle.

Because this chapter isn't explicitly about control creation, it doesn't go into detail about all the methods shown in Figure 9.2, but it's worth seeing a quick description of them all before you begin coding:

> **For More Information**
> For more information on creating custom controls, see *Developing Microsoft ASP.NET Server Controls and Components*, from Microsoft Press.

- `Constructor`—This method is called when the control is added to the control tree.
- `OnInit` **and** `Init`—At the stage at which these methods are called, all properties of the control have been set, and all child controls have been instantiated.
- `TrackViewState`—This method is automatically invoked by the page to ensure that property changes are saved with the viewstate of the control.
- `LoadViewState`—This method is called only during postback, allowing you to restore the control to its state at the end of processing the previous request.
- `LoadPostData`—This method is called only during postback and only if the control participates in postback, and it allows you to update its state from the posted data.
- `OnLoad` **and** `Load`—At the stage at which these methods are called, all controls have been initialized.
- `RaisePostDataChangedEvent`—This method is called only on postback and only if the control participates in postback. It allows you to indicate that the control has changed its state because of postback.
- `RaisePostBackEvent`—This method is called only on postback and only if the control participates in postback. It allows you to map client events to the server.
- `CreateChildControls`—This method allows you to create child controls and add them to the control tree.
- `PreRender`—This method allows you to perform any processing before the control is rendered to the page, such as registering for postback.
- `SaveViewState`—This method allows you to perform any custom viewstate management.
- `Render`—This method writes the markup to the client and by default calls child controls to allow them to render their contents.
- `Unload`—This method is called when the control is to be unloaded.
- `Dispose`—This method is raised to enable you to clean up and dispose of any resources used by the control, such as database connections.

In creating a custom template control in this chapter, you aren't going to use all these methods because the base implementation is more than adequate, but knowing the order in which the events are called is useful.

Custom Layout Control Output

The layout of the sample pages in this chapter is tabular in format, as shown in Figure 9.3, so the layout control must output HTML that generates this structure.

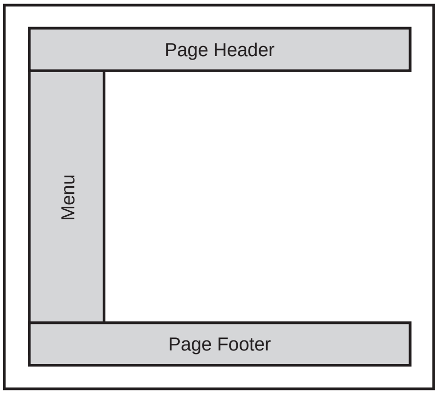


FIGURE 9.3
The layout created by the Custom Layout Control.

To get this structure, you can use an HTML table, with four cells; the header and footer cells each span two columns and the menu cell is narrow, leaving plenty of room for the content cell. The controls need to create the following HTML:

```
<table>
  <tr>
    <td colspan="2"> header </td>
  </tr>
  <tr>
    <td> menu </td>
    <td> content </td>
  </tr>
  <tr>
    <td colspan="2"> footer </td>
  </tr>
</table>
```

With the header, footer, and menu cells, you can also output any required HTML.

CSS Versus Table Layout

There are plenty of opponents to the use of tables for layout; these folks say that CSS should be used instead. While this is a valid point and CSS is just as easy to implement as HTML tables, for the purposes of this example, the table approach is best. It's simple to understand, and there's one fewer file (the CSS file) to worry about. Also, CSS support is not full across all browsers, so the table approach is guaranteed to work for all viewers of your page.

Creating Content from a Custom Control

There are two ways a custom control can create content: It can override the `CreateChildControls` method and add controls to the control tree, or it can override the `Render` method to directly render output (HTML, in this case). Both techniques are easy to implement, but which you use depends on what the control is going to do. For example, `CreateChildControls` would be used like this:

```
Protected Overrides Sub CreateChildControls()
  Dim tch As New TableCell()
  tch.Attributes.Add("colspan", "2")
  tch.Controls.Add(New LiteralControl("heading"))

  ' create more controls
  ' ...

  Dim tbl As New Table()
  Me.Controls.Add(tbl)
End Sub
```

In this case, a table cell is created and then is added to the `Controls` collection. (Other controls would also be created, but they are not shown to reduce the amount of code shown here.) When the control is rendered, the child controls are also rendered because they are part of the control tree.

The other way to output content is to override the `Render` method and render the actual contents yourself (as opposed to the preceding example, where the child controls render themselves). For example, the `Render` method might look like this:

```
Protected Overrides Sub Render(ByVal writer As System.Web.UI.HtmlTextWriter)

  writer.RenderBeginTag(HtmlTextWriterTag.Table)

  writer.RenderBeginTag(HtmlTextWriterTag.Tr)
  writer.AddAttribute(HtmlTextWriterAttribute.Colspan, "2")
  writer.RenderBeginTag(HtmlTextWriterTag.Td)
  writer.WriteLine("header content")
  writer.RenderEndTag()
  writer.RenderEndTag()
```

The ASP.NET page framework passes an `HtmlTextWriter` instance to the `Render` method. This provides a way to write HTML content directly to the output stream. Instead of creating controls (for example, `Table`, `TableCell`), you actually write out the actual HTML elements.

Using Controls Versus Rendering

The method you use to create the content for a control depends on a few factors:

- Using controls allows you to reuse the functionality of existing controls. Consider trying to render a `DataGrid` control, for instance.
- Controls are instantiated and added to the control tree, which incurs a performance penalty.
- Controls are easier to use than rendering if you need child controls to take part in postback.
- Rendering is quicker than using controls, but it's harder to implement things such as post-back handling and validation with rendering.

Ultimately, the choice between using controls and rendering is a trade-off—speed and size against ease of programming. The example in this chapter uses rendering because the rendered contents of the control itself (not the children) require no postback or validation; the contents simply provide a structure for other controls. Using server controls would bring the overhead of additional controls to be rendered, and those controls also increase the page size due to their requirements (such as setting attributes that might not be required).

Creating a Custom Layout Control

Creating the custom control is simply a matter of creating a new class—a Web control library, if you're using Visual Studio .NET. At the very minimum, the control must output the table structure, along with any controls that are contained within the control. For example, consider the control being used like so:

```
<sams:MasterPageControl runat="server" id="tctl1">
  <h1>Welcome to our site</h1>
  Please enter your email to subscribe:
  <asp:TextBox id="email" runat="server" />
  <asp:Button id="btnSubscribe" Text="Subscribe" runat="server"
    onClick="btnSubscribe_Click" />
</sams:MasterPageControl>
```

The contained content must be displayed within the content region of the table. Therefore, when the custom control is instantiated, it must grab the child controls as they are added to its `Controls` collection. You make this happen by overriding another method (`AddParsedSubObject`)—so instead of the ASP.NET page framework adding these controls to the collection, you can intercept them and keep your own store. Then, when rendering the page, you can output the controls in the desired location. The code for the `MasterPageControl` custom control is shown in Listing 9.1.

LISTING 9.1 The `MasterPageControl` Custom Control

```
Imports System.ComponentModel
Imports System.Web.UI
Imports System.Web.UI.WebControls

Public Class MasterPageControl
    Inherits System.Web.UI.Control

    ' add client page objects to our own collection
    Protected _controlBin As ControlCollection
    Protected Overrides Sub AddParsedSubObject(ByVal obj As Object)

        If IsNothing(_controlBin) Then
            _controlBin = New ControlCollection(Me)
        End If
```

LISTING 9.1 Continued

```
        Me._controlBin.Add(CType(obj, System.Web.UI.Control))

    End Sub

    Protected Overrides Sub Render(ByVal writer As System.Web.UI.HtmlTextWriter)

        writer.AddAttribute(HtmlTextWriterAttribute.Width, "100%")
        writer.AddAttribute(HtmlTextWriterAttribute.Border, "1")
        writer.RenderBeginTag(HtmlTextWriterTag.Table)

        ' header
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.AddAttribute(HtmlTextWriterAttribute.Colspan, "2")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        writer.WriteLine("header content")
        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr

        ' menu
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
        writer.AddAttribute(HtmlTextWriterAttribute.Width, "15%")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        writer.WriteLine("menu")
        writer.WriteLine("menu")
        writer.RenderEndTag() ' td

        ' content
        writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)

        ' render the client controls
        Dim i As Integer
        For i = 0 To _controlBin.Count - 1
            _controlBin(i).RenderControl(writer)
        Next

        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr

        ' footer
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.AddAttribute(HtmlTextWriterAttribute.Colspan, "2")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
```

LISTING 9.1 Continued

```
        writer.WriteLine("footer content")
        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr

        writer.RenderEndTag() ' table

    End Sub
End Class
```

The output from Listing 9.1 is shown in Figure 9.4.

FIGURE 9.4 A custom layout control in use.

Capturing contained controls as they are added to the page involves only two overridden methods, the first of which is `AddParsedSubObject`:

```
Protected _controlBin As ControlCollection

Protected Overrides Sub AddParsedSubObject(ByVal obj As Object)

    If IsNothing(_controlBin) Then
        _controlBin = New ControlCollection(Me)
    End If
    Me._controlBin.Add(CType(obj, System.Web.UI.Control))

End Sub
```

The ASP.NET page framework calls `AddParsedSubObject` for each control to be added to the page, and the control to be added is passed in as a parameter. The `MasterPageControl` control simply has a `ControlCollection` object (`_controlBin`) into which the control passed in as a parameter is stored for later use.

To render the contents of the structural table and the child controls, the `Render` method is overridden. Here the `HtmlTextWriter` instance (`writer`) is used to write HTML tags and attributes (`AddAttribute` is called before the tag to which the attributes apply). Here's an example:

```
Protected Overrides Sub Render(ByVal writer As System.Web.UI.HtmlTextWriter)

    writer.AddAttribute(HtmlTextWriterAttribute.Width, "100%")
    writer.AddAttribute(HtmlTextWriterAttribute.Border, "1")
    writer.RenderBeginTag(HtmlTextWriterTag.Table)
```

The child controls (that is, those that are the content of the page) are added by looping through controls in `_controlBin` and calling the `RenderControl` method on each control. This tells the control to render itself. The child controls are rendered within a table cell, as shown here:

```
writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
writer.RenderBeginTag(HtmlTextWriterTag.Td)

' render the client controls
Dim i As Integer
For i = 0 To _controlBin.Count - 1
    _controlBin(i).RenderControl(writer)
Next

writer.RenderEndTag() ' td
```

That's all there is to the control. You simply need to compile it and place it into the application's `bin` directory.

This template control does several things that wouldn't be useful in reality. First, it uses a border for the table, which would look pretty dreadful for a site design. It is included here to make it easy to see what is being output. Second, the actual mandatory content (header, menu, and footer) is simply text, just to illustrate how simple the control can be. For a real Web site, the content would include a logo, a menu control, and so on. Rather than render these yourself, you would probably want to use the same technique as for contained controls—call the `RenderControl` method on a control instance. The control instance can be created in the `CreateChildControls` method, using a global variable. For example, Listing 9.2 shows how the `MasterPageControl` custom control could be implemented to accommodate this.

LISTING 9.2 Creating and Rendering Child Controls

```
Imports System.ComponentModel
Imports System.Web.UI
Imports System.Web.UI.WebControls

Public Class MasterPageControl
    Inherits System.Web.UI.Control
```

LISTING 9.2 Continued

```
    Protected _logo As Image
    Protected _ads As AdRotator

    ' add client page objects to our own collection
    Protected _controlBin As ControlCollection
    Protected Overrides Sub AddParsedSubObject(ByVal obj As Object)

        If IsNothing(_controlBin) Then
            _controlBin = New ControlCollection(Me)
        End If
        Me._controlBin.Add(CType(obj, System.Web.UI.Control))

    End Sub

    Protected Overrides Sub CreateChildControls()

        Controls.Clear()

        _logo = New Image
        _logo.ImageUrl = "images/logo.gif"
        _logo.BorderStyle = BorderStyle.None
        Me.Controls.Add(_logo)

        _ads = New AdRotator
        _ads.AdvertisementFile = "adverts.xml"
        _ads.BorderStyle = BorderStyle.None
        Me.Controls.Add(_ads)

    End Sub

    Protected Overrides Sub Render(ByVal writer As System.Web.UI.HtmlTextWriter)

        writer.AddAttribute(HtmlTextWriterAttribute.Width, "100%")
        writer.AddAttribute(HtmlTextWriterAttribute.Border, "1")
        writer.RenderBeginTag(HtmlTextWriterTag.Table)

        ' header
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        _logo.RenderControl(writer)
        writer.RenderEndTag() ' td
        writer.AddAttribute(HtmlTextWriterAttribute.Align, "right")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        _ads.RenderControl(writer)
```

LISTING 9.2 Continued

```
        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr
        . . . rest of rendering

    End Sub
```

The preceding version of the control hasn't changed much. There are two global variables, for `Image` and `AdRotator` controls. In the `CreateChildControls` method, the current `Controls` collection is first cleared, to ensure that duplicate controls aren't added to the control tree (this can happen if your control, or one that derives from it, calls `CreateChildControls` multiple times). The properties for these controls are set, and they are added to the control tree.

The `Render` method also changes, converting the page header to two table cells: one for the logo and one for the advertisements. Within each of these cells, the `RenderControl` method is called on the appropriate control, telling the control to render itself. The same technique could be used for other mandatory content, such as the menu.

BEST PRACTICE

Creating Controls Versus Rendering

There is a natural overhead involved in using the `CreateChildControls` method, but that doesn't mean you should never use it. The example in this chapter is an excellent case for its use, where using a nontrivial control such as `AdRotator` is easier done by simply creating the control and adding it to the control tree than by re-creating the rotator logic and rendering the HTML. Generally, it's best to use `CreateChildControls` if you need postback handling or if the content you are creating is complex and already encapsulated by a server control.

A Server Control That Uses Templates

The `MasterPageControl` custom control example shows how simple a control can be, but it suffers from allowing page content to be placed in only a single area. What if, for example, you wanted to allow a content region but also allow the menu region to be replaced—or even added to? That isn't possible with the current `MasterPageControl` control because there is no way to determine which of the contained controls is intended for the content region and which for the menu region.

To solve this problem, you can build a templated control that is similar to the data bound controls (`DataList`, `DataGrid`, and `Repeater` controls) that have templates. The control could implement two templates, called `MenuTemplate` and `ContentTemplate`, providing a way to clearly differentiate the areas for contained controls. For example, a `MasterPageControlTemplated` control could be used like this:

```
<sams:MasterPageControlTemplated runat="server" id="tctl2">
  <MenuTemplate>
```

```
    menu 1<br />
    menu 2<br />
  </MenuTemplate>
  <ContentTemplate>
    <h1>Welcome to our site</h1>
    Please enter your email to subscribe:
    <asp:TextBox id="email" runat="server" />
    <asp:Button id="btnSubscribe" Text="Subscribe" runat="server"
      onClick="btnSubscribe_Click" />
  </ContentTemplate>
</sams:MasterPageControlTemplated>
```

Because the control will prove its own menu, the `MenuTemplate` element can be omitted. This provides the flexibility of templates but still allows for standardized content.

Creating a Templated Server Control

Many of the techniques used to create the `MasterPageControl` custom control also come into play when you create the templated server control in this example. The layout is the same, using a table, and both the `CreateChildControls` and `Render` methods are used.

The templated server control should inherit from `WebControl` to allow the use of templates, and it should implement the `INamingContainer` interface to ensure that any added child controls have unique names:

```
Public Class MasterPageControlTemplated
    Inherits System.Web.UI.WebControls.WebControl
    Implements INamingContainer
```

Because the control has two regions into which content can be put, there need to be two places to store those controls. In the previous example, the `AddParsedSubObject` method placed controls into a `ControlCollection` instance, from which they were later rendered. Because the templated server control uses templates, `AddParsedSubObject` isn't required; instead, you use a `TableCell` object for each of the storage areas (or placeholders, as they actually are in this case):

```
Protected _menuCell As TableCell
Protected _contentCell As TableCell
```

Then the templates need to be defined, and this is done as properties:

```
Dim _menuTemplate As ITemplate
Public Property MenuTemplate() As ITemplate
    Get
        Return _menuTemplate
    End Get
    Set(ByVal Value As ITemplate)
        _menuTemplate = Value
```

```
    End Set
End Property
```

There is one property for `MenuTemplate` and one for `ContentTemplate`—they are simple read/write properties of type `ITemplate`.

To enable controls within the templates to be rendered, the template needs to be created. You create the template by calling the `InstatiateIn` method of the template and passing in the container into which the content of the template is to be placed. This causes the ASP.NET page framework to read the controls from within the template and add them to the `Controls` collection of the container control. In this case, the container controls are the `TableCell` objects defined as global variables:

```
_menuCell = New TableCell
If Not (_menuTemplate Is Nothing) Then
    _menuTemplate.InstantiateIn(_menuCell)
End If
Me.Controls.Add(_menuCell)

_contentCell = New TableCell
If Not (_contentTemplate Is Nothing) Then
    _contentTemplate.InstantiateIn(_contentCell)
End If
Me.Controls.Add(_contentCell)
```

At this stage, the content from the templates has been parsed and added to the control tree. All that needs to happen is for the content to be rendered. Most of the rendering is the same as for the `MasterPageControl` server control, but there are changes for the menu and content regions. For both of these regions, there is no longer the need to manually render the table cells because the container control for the templates is a `TableCell` object. Therefore, they will automatically render the correct content.

For the menu, you need to supply default content if content is not supplied on the page. To do that, you first check the `_menuTemplate` object to see if it is `Nothing`; if it is, that means there is not a `MenuTemplate` element on the page. If it isn't `Nothing` (that is, there is a `MenuTemplate` element on the page), you check the `_menuCell` control—first to see if it has child controls and then to see if it has any text. Literal content will show up as text, whereas ASP.NET controls will show up as child controls. If no content has been supplied, the default content can be added to the content table cell before being rendered. The following code shows this in practice, checking that the contents of the menu template are empty before rendering default content:

```
writer.RenderBeginTag(HtmlTextWriterTag.Tr)
writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
writer.AddAttribute(HtmlTextWriterAttribute.Width, "15%")
If _menuTemplate Is Nothing _
    OrElse _menuCell.HasControls = False _
    And _menuCell.Text.Trim = "" Then
```

```
    ' either the template hasn't been supplied, or it has been
    ' supplied but with no contained controls. So add default content.
    _menuCell.Controls.Add(New LiteralControl("menu<br />"))
    _menuCell.Controls.Add(New LiteralControl("menu<br />"))
End If
_menuCell.RenderControl(writer)
```

For the content template, there is no default content, so the control can be rendered directly, regardless of whether the template has been defined or has content:

```
writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
_contentCell.RenderControl(writer)
writer.RenderEndTag() ' tr
```

The full code for the new control is shown in Listing 9.3.

LISTING 9.3 A Templated Master Page Control

```
Public Class MasterPageControlTemplated
    Inherits System.Web.UI.WebControls.WebControl
    Implements INamingContainer

    Protected _logo As Image
    Protected _ads As AdRotator
    Protected _menuCell As TableCell
    Protected _contentCell As TableCell

    ' The template used for the menu
    Dim _menuTemplate As ITemplate
    Public Property MenuTemplate() As ITemplate
        Get
            Return _menuTemplate
        End Get
        Set(ByVal Value As ITemplate)
            _menuTemplate = Value
        End Set
    End Property

    ' The template used for the content
    Dim _contentTemplate As ITemplate
    Public Property ContentTemplate() As ITemplate
        Get
            Return _contentTemplate
        End Get
        Set(ByVal Value As ITemplate)
            _contentTemplate = Value
```

LISTING 9.3 Continued

```
        End Set
    End Property

    Protected Overrides Sub CreateChildControls()

        Controls.Clear()

        _logo = New Image
        _logo.ImageUrl = "images/logo.gif"
        _logo.BorderStyle = BorderStyle.None
        Me.Controls.Add(_logo)

        _ads = New AdRotator
        _ads.AdvertisementFile = "adverts.xml"
        _ads.BorderStyle = BorderStyle.None
        Me.Controls.Add(_ads)

        ' create the table cell for the menu and
        ' instantiate the controls within the template
        _menuCell = New TableCell
        If Not (_menuTemplate Is Nothing) Then
            _menuTemplate.InstantiateIn(_menuCell)
        End If
        Me.Controls.Add(_menuCell)

        ' create the table cell for the content and
        ' instantiate the controls within the template
        _contentCell = New TableCell
        If Not (_contentTemplate Is Nothing) Then
            _contentTemplate.InstantiateIn(_contentCell)
        End If
        Me.Controls.Add(_contentCell)

    End Sub

    Protected Overrides Sub Render(ByVal writer As System.Web.UI.HtmlTextWriter)

        writer.AddAttribute(HtmlTextWriterAttribute.Width, "100%")
        writer.AddAttribute(HtmlTextWriterAttribute.Border, "1")
        writer.RenderBeginTag(HtmlTextWriterTag.Table)

        ' header
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
```

LISTING 9.3 Continued

```
        _logo.RenderControl(writer)
        writer.RenderEndTag() ' td
        writer.AddAttribute(HtmlTextWriterAttribute.Align, "right")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        _ads.RenderControl(writer)
        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr

        ' menu
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
        writer.AddAttribute(HtmlTextWriterAttribute.Width, "15%")
        If _menuTemplate Is Nothing _
            OrElse _menuCell.HasControls = False _
            And _menuCell.Text.Trim = "" Then
            ' either the template hasn't been supplied, or it has been
            ' supplied but with no contained controls. So add default content.
            _menuCell.Controls.Add(New LiteralControl("menu<br />"))
            _menuCell.Controls.Add(New LiteralControl("menu<br />"))
        End If
        _menuCell.RenderControl(writer)

        ' content
        writer.AddAttribute(HtmlTextWriterAttribute.Valign, "top")
        _contentCell.RenderControl(writer)
        writer.RenderEndTag() ' tr

        ' footer
        writer.RenderBeginTag(HtmlTextWriterTag.Tr)
        writer.AddAttribute(HtmlTextWriterAttribute.Colspan, "2")
        writer.RenderBeginTag(HtmlTextWriterTag.Td)
        writer.WriteLine("footer content")
        writer.RenderEndTag() ' td
        writer.RenderEndTag() ' tr

        writer.RenderEndTag() ' table

    End Sub

End Class
```

Creating Default Content for Templates

Both of the examples described in the preceding sections have one major drawback: The default content is hard-coded within the control. A good way to avoid this is to store the default content outside the control, such as in a user control. When you do this, not only can you easily change the content, but you can also design the content within a design tool. Reading the content from the user control is simple because the page has a `LoadControl` method:

```
Dim uc As UserControl
uc = LoadControl("HeaderTemplate.ascx")
```

The loaded user control can then be added to the `Controls` collection. The custom server control has to change to load the content dynamically. For a start, there are three global variables to hold the user controls:

```
Protected _headerContent As UserControl
Protected _menuContent As UserControl
Protected _footerContent As UserControl
```

The `Image` and `AdRotator` controls are no longer required because they will be stored in the user control.

The `CreateChildControls` method is modified to load the user controls and add them to the control tree. They need to be added to the control tree to ensure that they take part in the event life cycle. You can see this in the following code example, where the templates are stored as user controls and loaded dynamically as the custom control is created:

```
_headerContent = Page.LoadControl("Templates\HeaderTemplate.ascx")
_footerContent = Page.LoadControl("Templates\FooterTemplate.ascx")
Me.Controls.Add(_headerContent)
Me.Controls.Add(_footerContent)
```

The `Render` method also changes, simply to render the loaded user control:

```
writer.RenderBeginTag(HtmlTextWriterTag.Td)
_headerContent.RenderControl(writer)
writer.RenderEndTag() ' td
```

This solution provides an enforceable structure with the added benefit of easily maintainable content for the mandatory areas of the page. Of course the disadvantage is that the layout is almost too rigid—it's baked into the custom server control, so if it ever needs to be changed, the code has to be edited and compiled, and the assembly has to be redistributed. Also, the design of the layout is based on creating controls and rendering—which is not as simple as just designing pages. You can further extend this architecture by having properties of the custom server control that allow the definition of where the default content comes from. If these properties aren't set, the content can be fetched from a default location.

Creating Dynamic Regions for Page Content

If the design of the MasterPageControlTemplated example is too rigid, what's the best way of adding flexibility? One way is to take some ideas from the ASP.NET 2.0 implementation, which uses a master page and a content page. Each contains special server controls with matching ID attributes, and when the page is generated, the ID attributes are used to match content to the area where the content should go. For example, in the master page there is a ContentPlaceHolder control that indicates the area where content can be placed (a bit like a template). The master page (site.master) might look like this:

```
<table width="100%" border="1">
  <tr>
    <td><img src="images/logo.gif" border="0/></td>
    <td align="right"><asp: AdRotator id="AdRotator1"
        runat="server" AdvertisementFile="../adverts.xml"></td>
  </tr>
  <tr>
    <td valign="top" width="15%">
      <asp:ContentPlaceHolder id="MenuContent" runat="server" />
    </td>
    <td valign="top">
      <asp:ContentPlaceHolder id="PageContent" runat="server" />
    </td>
  </tr>
  <tr>
    <td colspan="2">footer content</td>
  </tr>
</table>
```

In this example, the two ContentPlaceHolder controls identify the regions where page content can go. The actual content page might then look like this:

```
<%@ Page MasterPageFile="site.master" %>
<asp:Content id="MenuContent" runat="server">
  The menu content goes here
</asp:Content>
<asp:Content id="PageContent" runat="server">
  <h1>Welcome to our site</h1>
  Please enter your email to subscribe:
  <asp:TextBox id="email" runat="server" />
  <asp:Button id="btnSubscribe" Text="Subscribe" runat="server"
      onClick="btnSubscribe_Click" />
</asp:Content>
```

All that the content page needs to include is the Content controls, with the same ID values as the ContentPlaceHolder controls in the master page. The master page is identified by the

MasterPageFile attribute on the Page directive. When the page is compiled, the following steps take place:

1. The page content is initially made up from the content of the master page.
2. The content page is checked for Content controls, and for each one, a ContentPlaceHolder control with a matching ID value in the master page is located.
3. The content from within the Content control is placed into the page, replacing any content that the master page defined.
4. If no Content control is found in the content page, the default content from the ContentPlaceHolder control is used.

This process is easy to re-create in ASP.NET 1.1, with one major exception: It's not part of the ASP.NET Page Framework. Therefore, you can't use a MasterPageFile attribute on the page directive, and you don't have automatic support for master pages. However, you can write your own page class that provides much of the same functionality.

Using a Custom Page Class for a Page Template

Creating a custom page class is in many ways the same as creating custom server controls: You need to parse the controls and add them to a control tree. For the master page implementation, though, you need to take this a bit further, by loading a master page (a user control), finding the ContentPlaceHolder controls, and copying content from the appropriate Content control.

Creating the Content and ContentPlaceHolder Controls

Both the Content and ContentPlaceHolder controls are simply placeholders—ways of defining a region into which controls are placed. Therefore, they can simply inherit from the Panel class, but no output needs to be rendered, so the RenderBeginTag and RenderEndTag methods are overridden. This means that when the page is rendered, it doesn't matter if these controls remain on the page because they don't actually render anything. The ContentPlaceHolder control looks as follows (and the Content control implementation is the same):

```
Public Class ContentPlaceHolder
  Inherits System.Web.UI.WebControls.Panel

  Public Sub New()
  End Sub

  Public Overrides Sub RenderBeginTag(ByVal writer As HtmlTextWriter)
  End Sub

  Public Overrides Sub RenderEndTag(ByVal writer As HtmlTextWriter)
  End Sub
End Class
```

Creating a Custom Page Class

The custom page class needs to take several steps:

1. The class inherits from `System.Web.UI.Page`, which provides all the default page processing.
2. Next, the class overrides `AddParsedSubObjects`, as demonstrated in previous examples in this chapter, to add the page controls to the private control collection. Figure 9.5 shows how each `Content` control is added to the internal controls collection.

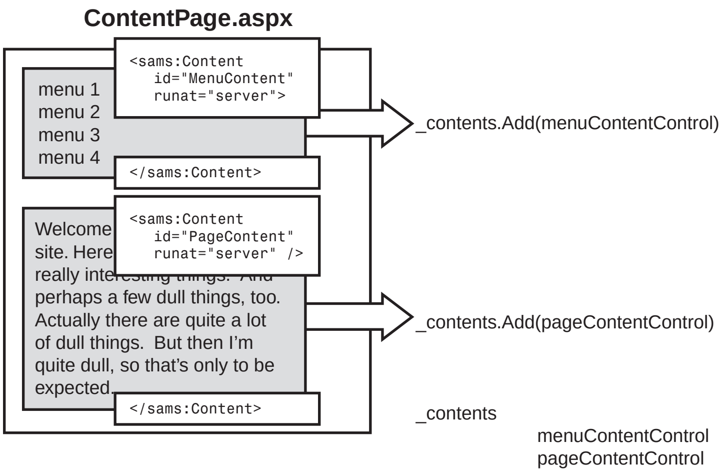


FIGURE 9.5
Adding controls to the internal controls collection from the content page.

At this stage, the private control collection (`_contents`) contains the contents of the page. This is only the two `Content` controls; any controls defined within the `Content` control are children and therefore don't show up as top-level controls.

3. The class parses the master page, adding its controls to the control tree (as shown in Figure 9.6), thus making the page look like the master.

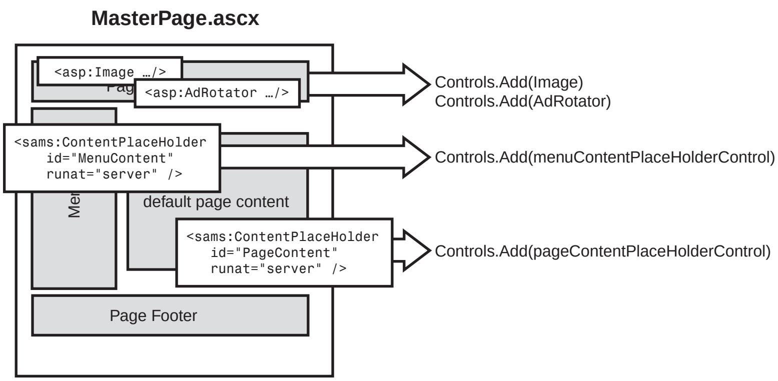


FIGURE 9.6
Adding controls to the controls collection from the master page.

At this stage, the `Controls` collection for the page contains all the controls from the master page. This includes all content, such as the table for layout, the `Image` and `AdRotator` controls in the header, and the `ContentPlaceHolder` controls (and any children).

4. The class loops through the private control collection and matches the `ID` values to those on the page (which were copied from the master page). For each control that is found, the class adds the contents, thus replacing any default content from the master page with the actual required content, as shown in Figure 9.7.

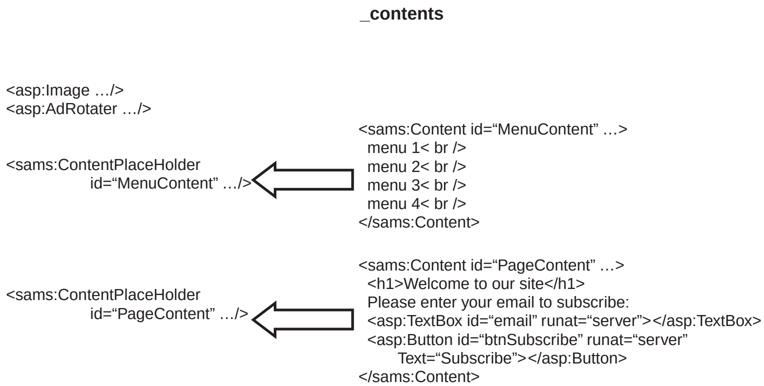


FIGURE 9.7 Replacing the `ContentPlaceHolder` controls with actual content.

After step 4, all content from the actual page is placed into the placeholders rather than into its default location (that is, at the end of the page—controls are added in the order in which they are defined).

To implement the process described in the preceding steps, one additional piece of information needs to be known—the name of the master page file. This is defined as a public property of the custom page class. Listing 9.4 shows the full code for the `MasterPage` custom page class.

LISTING 9.4 The `MasterPage` Custom Page Class

```
Public Class MasterPage
  Inherits System.Web.UI.Page

  Private Const ErrOnlyContent As String = _
              "Content page can only contain <sams:Content /> controls: {0}"
  Private Const ErrNoMaster As String = _
              "MasterPageFile property not set"
  Private Const ErrNoHolder As String = _
              "<sams:ContentPlaceHolder id='{0}'/> must be defined in {1}"

  Private _contents As New ArrayList
  Private _template As Control
  Private _masterPageFile As String = String.Empty
```

LISTING 9.4 Continued

```
Public Property MasterPageFile() As String
  Get
    Return _masterPageFile
  End Get
  Set(ByVal Value As String)
    _masterPageFile = Value
  End Set
End Property

Protected Overrides Sub OnInit(ByVal e As EventArgs)

  MyBase.OnInit(e)

  Me.BuildMasterPage()
  Me.BuildContents()

End Sub

Protected Overrides Sub AddParsedSubObject(ByVal obj As Object)

  If TypeOf (obj) Is Content Then
    ' add it to the internal controls collection
    _contents.Add(obj)
  Else
    '  Should only allow controls of type Content
    If TypeOf (obj) Is LiteralControl Then
      Dim ctl As LiteralControl = CType(obj, LiteralControl)
      If ctl.Text.Trim <> "" Then
        Throw New Exception(String.Format(ErrOnlyContent, ctl.Text))
      End If
    ElseIf Not (TypeOf (obj) Is LiteralControl) Then
      Throw New Exception(String.Format(ErrOnlyContent, obj.ToString))
    End If
  End If

End Sub

' add the controls from the master file
Private Sub BuildMasterPage()

  If _masterPageFile = String.Empty Then
    ' if not set at the page level check for being set at the config level
    _masterPageFile = ConfigurationSettings.AppSettings("MasterPageFile")
    If _masterPageFile = String.Empty Then
      Throw New Exception(ErrNoMaster)
```

LISTING 9.4 Continued

```
      End If
    End If

    ' load the master file
    Me._template = Me.Page.LoadControl(Me._masterPageFile)

    ' iterate through the controls of the master file, adding
    ' them to the internal controls collection
    Dim index As Integer
    For index = 0 To Me._template.Controls.Count - 1
      Dim ctl As Control = Me._template.Controls(0)
      Me._template.Controls.Remove(ctl)
      If (ctl.Visible) Then
        Me.Controls.Add(ctl)
      End If
    Next
    Me.Controls.AddAt(0, Me._template)

  End Sub

  ' add the controls from the content page
  Private Sub BuildContents()
    Dim ct As Content
    For Each ct In Me._contents
      Dim holder As Control = Me.FindControl(ct.ID)

      ' control with same name must be of type ContentPlaceHolder
      If holder Is Nothing Or Not (TypeOf (holder) Is ContentPlaceHolder) Then
        Throw New Exception(String.Format(ErrNoHolder, ct.ID, _masterPageFile))
      End If

      ' only clear default content if the Content control actually has content
      If ct.HasControls Then
        holder.Controls.Clear()
      End If

      ' add the individual controls from the current page
      Dim index As Integer
      For index = 0 To ct.Controls.Count - 1
        holder.Controls.Add(ct.Controls(0))
      Next
    Next

  End Sub
End Class
```

Creating a Master Page

A master page itself is simply a user control for which all replaceable content must be placed within `ContentPlaceHolder` controls. For example, the master page used in the `MasterPage` custom page class example would look like this:

```
<%@ Control Language="vb" AutoEventWireup="false"
    Codebehind="siteMaster.ascx.vb" Inherits=".siteMaster"
    TargetSchema="http://schemas.microsoft.com/intellisense/ie5" %>
<%@ Register TagPrefix="sams" Namespace="SAMS.PageTemplates"
    Assembly="MasterPageControls" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
      <table border="1" width="100%">
        <tr>
          <td><asp:Image ImageUrl="~/images/logo.gif"
                   Runat="server" id="Image1" /></td>
          <td align="right">
            <asp:AdRotator id="AdRotator1" runat="server"
                 AdvertisementFile="~/adverts.xml"></asp:AdRotator>
          </td>
        </tr>
        <tr>
          <td width="15%" valign="top">
            <sams:ContentPlaceHolder id="MenuContent" runat="server">
              menu<br />
              menu<br />
            </sams:ContentPlaceHolder>
          </td>
          <td valign="top">
            <sams:ContentPlaceHolder id="PageContent" runat="server">
              Default Content
           </sams:ContentPlaceHolder>
          </td>
        </tr>
        <tr>
          <td colspan="2">
            <p><font size="2">Copyright (c) SAMS</font></p>
          </td>
        </tr>
      </table>
    </form>
  </body>
</HTML>
```

You can see that this is a standard page, using the same table layout you've seen before in this chapter. The content that can be replaced is defined by `ContentPlaceHolder` controls.

Using a Custom Page Class

To use the custom page class created in the preceding section, you have to do a little work, but it's not much. First, you have to make sure your page inherits from the custom class rather than from `System.Web.UI.Page`. You then have to set the `MasterPageFile` property so that the page knows which master page to use. Then you need to put the content into `Content` controls.

For the first of the `Content` controls, you need to use code-behind files, where you'd normally see something like this:

```
Public Class default
    Inherits System.Web.UI.Page
```

You must change this code to the following:

```
Public Class UsingMasterPage
    Inherits SAMS.PageTemplates.MasterPage
```

Now when the page loads, the custom class will be run. To enable the custom class to work, though, you need to set the `MasterPageFile` attribute, which you can do in a number of ways. The first is to set it in the `Page_Init` event procedure—usually in a region full of scary warnings:

```
Private Sub Page_Init(ByVal sender As System.Object, _
                      ByVal e As System.EventArgs) Handles MyBase.Init
    'CODEGEN: This method call is required by the Web Form Designer
    'Do not modify it using the code editor.
    InitializeComponent()
    MasterPageFile = "siteMaster.ascx"
End Sub
```

Don't worry, though; it's perfectly safe to add code to this procedure, as long as you don't delete anything that's already there. You can't use the `Page_Load` event because it occurs too late in the event life cycle—after the controls have already been added to the page.

Another way to set the master page file is to use the Web configuration file, `web.config`, where you can have a custom `appSetting` element to define the master file:

```
<configuration>
  <appSettings>
    <add key="MasterPageFile" value="siteMaster.ascx" />
  </appSettings>
```

Using a Custom Event for Setting the Master Page

Another way of setting the master page file, besides using the `Page_Init` event, could be to have a custom event that is raised by the custom class. This event could be handled in the page where you set the master page. The advantage of this method is that it makes it explicitly clear what the event is for. The downloadable sample code (see `www.daveandal.net/books/6744/`) has this method implemented so you can see how it could be done.

The advantage of this method is that you don't have to modify each page; the master page file can be set automatically for each page that needs it. This also allows you to change the entire site design in one simple step.

After you have changed the page inheritance and set the master, you can design the page. This is simply a matter of adding `Content` controls with `ID` attributes that match those of the `ContentPlaceHolder` controls from the master. Within those `Content` controls you place the actual content. Here's an example:

```
<%@ Page trace="false" Language="vb" AutoEventWireup="false"
    Codebehind="UsingMasterPage.aspx.vb" Inherits=".UsingMasterPage"%>
<%@ Register TagPrefix="sams" Namespace="SAMS.PageTemplates"
    Assembly="MasterPageControls" %>

<sams:Content id="MenuContent" runat="server">
  menu 1<br />
  menu 2<br />
  menu 3<br />
  menu 4<br />
</sams:Content>

<sams:Content id="PageContent" runat="server">
  <h1>Welcome to our site</h1>
  Please enter your email to subscribe:
  <asp:TextBox id="email" runat="server"></asp:TextBox>
  <asp:Button id="btnSubscribe" runat="server" Text="Subscribe"></asp:Button>
</sams:Content>
```

The page cannot contain anything other than `Content` controls at the top level (blank space is allowed to aid readability, though). This is because all other content is supplied by the master page. If you don't want to override the default content from the master page, you can simply leave out the `Content` control.

This method is a little more involved than the `MasterPageControl` or `MasterPageControlTemplated` methods, especially for the design of content pages, but it offers the best flexibility. It is also very similar to the ASP.NET 2.0 approach, which means that migrating applications to ASP.NET 2.0 will be a simple matter; all you'll need to do is change the base class inheritance and modify the setting of the `MasterPageFile` attribute (either to set the attribute on the page directive or globally in `web.config`).

Using Custom Controls in Visual Studio .NET

The one real problem that all the examples described in this chapter have is lack of designer support. When you're using Visual Studio .NET, most controls look the same in the designer as they do on the page. This is because the designer renders HTML to the design surface. It's fairly easy to write a designer for custom controls. However, the designer architecture in ASP.NET 1.1

is limited and doesn't provide any easy support for what you are trying to do in this chapter: display some static content (the master content) while allowing editing of the page content.

Designers are limited to three modes of working:

- A read-only designer that displays the required HTML but that doesn't allow drag and drop.
- A read/writer designer that displays an editable region (that is, one that allows drag and drop).
- A designer for templated controls that can show static content but that allows editing of templates. However, the templates cannot be edited *in situ*, and a template editor replaces the control design to allow editing.

This means that for composite controls, which the template controls described in this chapter are, you cannot have some regions that are read-only and some that are read/write. You can have editable regions with a template, but that doesn't allow you to edit the controls within the look of the custom control. You can have the layout but without editing.

The choice you therefore have is whether to have no designer (and rely on using HTML view for editing your page) or whether to use a designer that doesn't really do one thing or the other. We think it's better to stick with no designer because at least that way it's clear that there is no design support, whereas with a designer, you're never quite sure what features it provides.

Summary

Although this chapter shows how to create custom controls, it is actually about how to provide support for master pages, which lead to more consistent and maintainable sites. In ASP.NET 1.1, using custom controls and using custom page classes are the only ways to achieve this functionality.

In this chapter you've seen that it's extremely easy to create custom controls in a variety of ways, from a simple control that renders a table and within the table the user content to a custom page class that allows specification of the rendering regions. The latter approach gives the best flexibility and is closest to the ASP.NET 2.0 implementation of master pages, thus providing an easy migration path.

PART III

Data Techniques

10

Relational Data-Handling Techniques

IN THIS CHAPTER

Chapter 4, "Working with Nested List Controls," is devoted to working with ASP.NET list controls, primarily nested `DataGrid` controls. You'll see the `DataGrid` control in use again elsewhere in this book; however, this chapter focuses on a range of issues that you might come up against when working with relational data.

This chapter starts with three topics related to SQL statements and stored procedures. The first of these is really a call to action, to make sure that you are not risking exposing your data or applications to damage or misuse by visitors with malicious intent. The other two topics are concerned with getting the most from stored procedures and understanding a particular issue with the SQL Server Tabular Data Service (TDS) provider (`SqlClient`) in the .NET Framework.

Next, this chapter covers getting the results you expect with filling a `DataSet` instance and writing code that can easily be converted to use any of the .NET Framework data providers. Finally, this chapter discusses an approach to editing data in a `DataGrid` control that allows multiple changes to be submitted to the server for updating in one go, rather than the individual postback approach that is used by default.

Using Parameters with SQL Statements and Stored Procedures

Despite the fact that it's usually best to use stored procedures when accessing a relational database such as SQL Server, many people find that they occasionally still need to use declarative SQL statements. Although stored procedures are usually more efficient than declarative SQL statements (stored procedures are compiled and can be reused) and offer better security and hide the database structure, there are cases in which creating a SQL statement dynamically at runtime is the obvious solution.

Moreover, when you're testing and developing Web pages that display data (as opposed to an application that has a separate data tier), SQL statements make it easy to get preview or test code working. Also, when you face situations in which the number of parameters or the number of columns you need to extract is not identifiable at design time, constructing a SQL statement dynamically may be the only solution.

Using Submitted Values in a SQL Statement

The issue that we're concerned with in this section comes about when you allow users to submit values that you subsequently use in a SQL statement. The sample page we provide, shown in Figure 10.1, demonstrates this quite neatly. The page provides two text boxes into which you can enter search strings for customer ID values in the Northwind sample database. The first text box uses the value you enter to build up a literal SQL statement that contains the string. When you click the first Go button, you see the result of the execution of this SQL statement, and the statement itself is displayed as well. This seems to work fine, and it returns the rows you would expect.

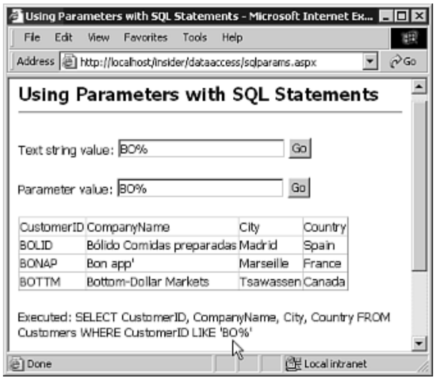


FIGURE 10.1 Entering a text string that is used to build a SQL statement.

If you enter the same search string into the second text box and click the second Go button, the set of rows that are returned is the same as the set returned when you enter that string in the first text box (see Figure 10.2). However, you can see that the SQL statement uses a parameter named `@CustomerID` this time. This parameter is populated by code in the page, before the SQL statement is executed (you'll see how later in this chapter).

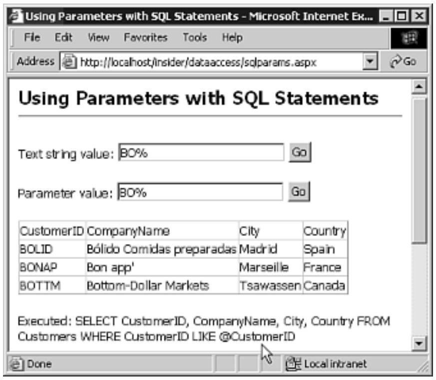


FIGURE 10.2 Entering a text string that is used to populate a parameter in the SQL statement.

The Effects of Malicious Input

There appears to be no difference in behavior when using stored procedures and when using declarative SQL statements when accessing a relational database. However, try the example from the preceding section again, this time using a different search string: `' or '1'='1`. Figure 10.3 shows quite clearly that there is a problem. You can see that now *all* the rows are returned. Normally you would expect to see all the rows only by entering `%` into the text box in this example, but entering `' or '1'='1` clearly demonstrates that you get something other than the expected result.

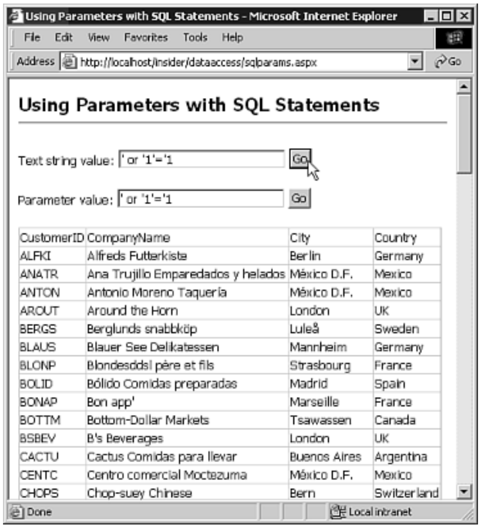


FIGURE 10.3 The result of entering a malicious string for the SQL statement.

The SQL statement is shown in the page, although it's not visible in Figure 10.3. If you scrolled down to the bottom of the page, you'd see the statement that was executed:

```
SELECT CustomerID, CompanyName, City, Country
FROM Customers WHERE CustomerID LIKE '' or '1' = '1'
```

The text entered into the text box has added an extra test to the `WHERE` clause, and this test will be `true` for every row in the table; therefore, all the rows are returned. If, for example, you were collecting a username and password from a visitor and creating a SQL statement this way, you could find that your system is open to attack from this type of value entered by a user. For example, you might construct the SQL statement for such a process by using code like this:

```
sSQL = "SELECT UserID FROM Users WHERE UserID = '" & txtUser.Text
& "' AND Password = '" & txtPassword.Text & "'"
```

In theory, this will return a row only when the user ID and password match the entries in the database. However, by using the technique just demonstrated, a visitor could contrive to have the following SQL statement executed:

```
SELECT UserID FROM Users WHERE UserID = 'johndoe'
AND Password = 'secret' or '1' = '1'
```

This would return a non-empty rowset, and if you only check whether there are any rows returned, you might find that your security has been breached.

However, if you enter the same text into the second text box in the sample page and click the second Go button to execute the SQL string with the value as a parameter, you'll see that no rows are returned (see Figure 10.4).

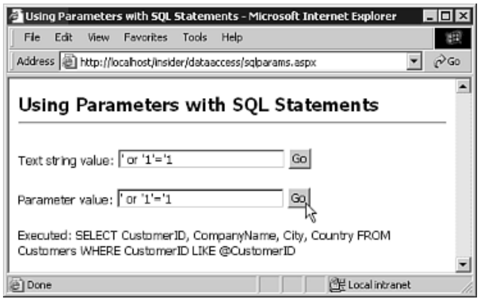


FIGURE 10.4 The result of entering a malicious string for the parameter to a SQL statement.

To understand why no rows are returned in this example, you can open the Profiler utility (by selecting Start, Microsoft SQL Server, Profiler) and trace the actions taken in the database. In this case, this is the instruction that SQL Server executes:

```
exec sp_executesql N'SELECT CustomerID, CompanyName, City, Country
FROM Customers WHERE CustomerID LIKE @CustomerID',
N'@CustomerID nvarchar(4000)', @CustomerID = N''' or ''1''=''1'
```

In other words, SQL Server is passing the SQL statement and the parameter separately to the system stored procedure named `sp_executesql`, and it is specifying that the parameter is a character string (nvarchar). This string does not match the value in the CustomerID column of any row, so no rows are returned.

How to Use SQL Profiler

To use SQL Profiler, open it from the Start menu or the Tools menu in Enterprise Manager and select File, New Trace to connect to your database. In the Trace Properties dialog that appears, select the Events tab and make sure that the complete set of actions for the Stored Procedure entry in the list of available TSQL event classes (displayed in the right-hand list) is selected. Then click Run, and you'll see the statements that are being executed appear in the main Profiler window.

It Gets Even Worse…

The problem with the literal construction of the SQL statement described in the preceding section actually leaves you open to risks that are even more serious than you might think. For example, if you enter the following text into the first text box and execute it, you'll see that the SQL statement shown in Figure 10.5 is used:

```
'; update customers set city='here!' where customerID like 'BOLID
```

In fact, this is a *batch statement* that contains two separate SQL statements. The first one fails to find any rows that match the empty string in the WHERE clause, but then the second one is executed, and it *updates* the table.

FIGURE 10.5 A malicious value that updates the source table in the database.

If you now change the value in the first text box and display the rows for customers whose IDs start with BO, you'll see that the first one has been updated (see Figure 10.6). However, if you try this with the second text box, you'll find that—as before—the process has no effect on the original data. The value that is entered and passed to SQL Server as a parameter simply fails to match any existing rows in the table, and nothing is changed or returned.

One consolation is that the attack could be worse. For example, your malicious visitor could have entered this instead:

```
'; drop database Northwind --
```

This deletes the database altogether. (The double hyphen at the end is a *rem* or *comment* marker that forces SQL Server to ignore the final apostrophe that gets added to the statement batch.)

So if you construct SQL statements dynamically in your code and there's any risk at all that the values you use might contain something other than you expect, you should always use parameters to build the SQL statement. In fact, it's not a bad idea to do it every time!

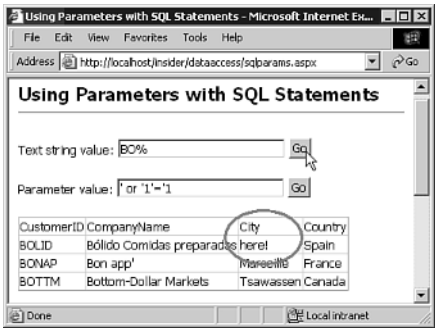


FIGURE 10.6 The result of executing the batch command shown in Figure 10.5.

The Code for Adding Parameters

The code used in the sample page shown in Figures 10.1 through 10.6 contains two routines—one for each of the two Go buttons in the page. The first one builds the SQL statement in literal fashion, using the following:

```
Dim sSQL As String = "SELECT CustomerID, CompanyName, City, " _
                   & "Country FROM Customers " _
                   & "WHERE CustomerID LIKE '" & sParam & "'"
```

In this case, `sParam` is the value extracted from the first text box on the page. This SQL statement is then executed and the result is assigned to the `DataSource` property of the `DataGrid` control on the page in the usual way.

The second routine, which runs when the second Go button is clicked, works a little differently from the first. Listing 10.1 shows the complete routine. After collecting the value from the second text box, the routine declares the SQL statement. However, this time, the `WHERE` clause contains a parameter named `@CustomerID`:

```
... "WHERE CustomerID LIKE @CustomerID"
```

LISTING 10.1 A Routine to Execute a SQL Statement with a Parameter

```
Sub UseParamValue(sender As Object, e As EventArgs)

  ' get input value from TextBox
  Dim sParam As String = txtParam.Text

  ' declare SQL statement containing parameter
  Dim sSQL As String = "SELECT CustomerID, CompanyName, City, " _
                     & "Country FROM Customers " _
                     & "WHERE CustomerID LIKE @CustomerID"

  ' get connection string, create connection and command
  Dim sConnect As String = _
    ConfigurationSettings.AppSettings("NorthwindSqlClientConnectString")
```

LISTING 10.1 Continued

```
  Dim oCon As New SqlConnection(sConnect)
  Dim oCmd As New SqlCommand(sSQL, oCon)

  Try

    ' specify query type, add parameter and open connection
    oCmd.Parameters.Add("@CustomerID", sParam)
    oCmd.CommandType = CommandType.Text
    oCon.Open()

    ' execute query and assign result to DataGrid
    dgr1.DataSource = oCmd.ExecuteReader()
    dgr1.DataBind()

    'close connection afterwards
    oCon.Close()

    ' display SQL statement in page and show hint
    lblResult.Text = "Executed: " & sSQL
    lblHint.Visible = True

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    ' can call Close more than once if required - no exception
    ' is generated if Connection is already closed
    oCon.Close()
    lblResult.Text = "<font color='red'><b>ERROR: </b>" _
                   & oErr.Message & "</font>"

  End Try

End Sub
```

Next, you create the `Connection` instance and `Command` instance as usual. Before executing the SQL statement, however, you have to add a parameter to the `Command` instance to match the parameter declared within the SQL statement. The sample code uses the simplest override of the `Add` method for the `Parameters` collection of the `Command` instance and specifies the name and value of the parameter. The data type of the variable is automatically used to set the data type of the parameter—in this case, a `String` data type, which means that the parameter will be treated as being of type `nvarchar` (`System.Data.SqlDbType.NVarChar`) when SQL Server processes it.

Parameter Name Prefixes in SQL Server

In databases *other than* SQL Server or Sybase databases, you use just a question mark (?) as the parameter placeholder. If there is more than one parameter, you use multiple question mark placeholders and you *must* add the parameters to the `Parameters` collection of the `Command` instance in the same order that the placeholders appear in the SQL statement. The names of the parameters are ignored in this case. You *can* use this same syntax with SQL Server and Sybase as well, although the named parameter technique is usually more readable and less error prone.

Notice also that you still have to use the value `Text` for the `CommandType` property of the `Command` instance because this is still a SQL statement and not a stored procedure. (`Text` is the default, so you could, in fact, omit it altogether.)

Ordering of Stored Procedures and Query Parameters

A parameter-related issue can cause problems if you are not aware of it. It concerns the way that the different .NET Framework data providers handle parameters when you specify them by name. The sample page shown in Figure 10.7 helps illustrate this issue.

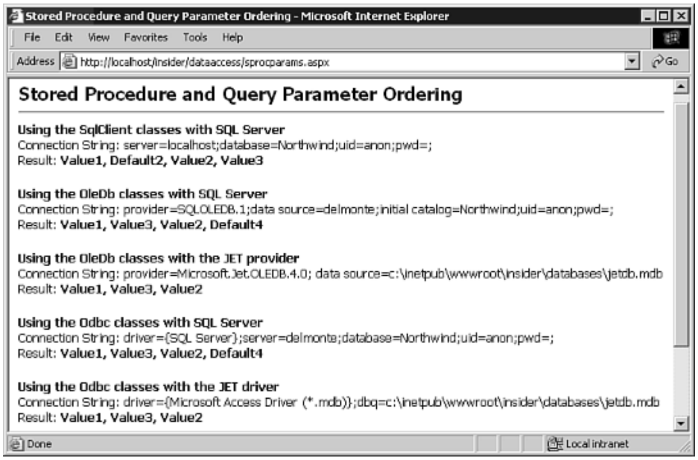


FIGURE 10.7
The ordering of stored procedure parameters that are specified by name.

The sample page uses two stored procedures—one in SQL Server and one in an Access database—and executes them by using the various data providers that are part of the .NET Framework. The SQL Server stored procedure is as follows:

```
CREATE PROCEDURE ParamOrderProc
@Param1 varchar (10) = 'Default1',
@Default varchar (10) = 'Default2',
@Param2 varchar (10) = 'Default3',
@Param3 varchar (10) = 'Default4'
AS
SELECT @Param1 + ', ' + @Default + ', ' + @Param2 + ', ' + @Param3
```

This SQL Server stored procedure simply collects the values of the four parameters you provide when the stored procedure is executed, and it returns a character string that concatenates their values together. However, the point here is that they are all optional parameters, which means that not all of them must be specified when you execute the stored procedure. If the code that executes the procedure does not provide a value for one of the parameters, the default value specified within the stored procedure is used instead.

Installing the Stored Procedure for This Example

A SQL script named `ParamOrderProc.sql` is provided in the `databases` subfolder of the samples you can download for this book (see `www.daveandal.net/books/6744/`). You can use this script to create the stored procedure for the example. For SQL Server, you open Query Analyzer from the SQL Server section of your Start menu, select the Northwind database, and then open the script file and execute it. You must have owner or administrator permission to create the stored procedure.

Unfortunately, however, Access doesn't support optional parameters, so the Access query used in the sample page has only three parameters and no default values:

```
PARAMETERS Param1 Text(10), Param2 Text(10), Param3 Text(10);
SELECT [Param1] + ', ' + [Param2] + ', ' + [Param3] AS Expr1;
```

The strange effects shown in Figure 10.7 come about because when you call the stored procedure, you add the parameters in a different order from which they are defined in the stored procedures:

```
oCmd.Parameters.Add("@Param1", "Value1")
oCmd.Parameters.Add("@Param3", "Value3")
oCmd.Parameters.Add("@Param2", "Value2")
```

The result shown in Figure 10.7 proves that with the exception of the `SqlClient` classes, the names you provide for parameters have no effect. They are ignored, and the parameters are passed to the stored procedure by position and *not* by name. You get back the three values in the same order as you specified them, even though the parameters' names don't match.

However, with the `SqlClient` classes, the result is different. With these classes, parameters are passed by name, so you get back the values in an order that matches the order within the `Parameters` collection. The order in which you add them to the `Parameters` collection doesn't matter; each one will match up with the corresponding named parameter in the stored procedure.

Using Default Values in a Stored Procedure

The previous example uses a stored procedure containing *optional parameters*. When you declare a parameter in a stored procedure in SQL Server and most other enterprise-level database systems, you can provide a default value for the parameter. In fact, it is required because this is how the database knows that it is an optional parameter. Without a default value, you'll get an error if you call the procedure without providing a value for that parameter.

BEST PRACTICE

Using Optional Parameters in a Stored Procedure

Optional parameters will only work really successfully when you use the `SqlClient` data provider because none of the other data providers (as discussed earlier in this chapter) pass parameters by name. To use other data providers, which pass parameters by position, you would have to make sure that the optional parameters are located at the end of the list and provide values for all the parameters up to the ones that you want to use the default values.

By taking advantage of sensible defaults for your parameters, you can simplify the data access code you have to write in your ASP.NET pages and data access components. Listing 10.2 shows the stored procedure used in the sample page for this section of the chapter. It is designed to update rows in the `Orders` table of the Northwind sample database, and you can see that it takes 12 parameters.

LISTING 10.2 A Stored Procedure That Provides Sensible Default Values

```
CREATE PROCEDURE ParamDefaultProc
  @OrderID int,                       @CustomerID nchar(5),
  @OrderDate datetime = NULL,         @RequiredDate datetime = NULL,
  @ShippedDate datetime = NULL,       @ShipVia int = 1,
  @Freight money = 25,                @ShipName nvarchar(40) = NULL,
  @ShipAddress nvarchar(60) = NULL,   @ShipCity nvarchar(15) = NULL,
  @ShipPostalCode nvarchar(10) = NULL, @ShipCountry nvarchar(15) = NULL
AS
IF @OrderDate IS NULL
  BEGIN
    SET @OrderDate = GETDATE()
  END
IF @RequiredDate IS NULL
  BEGIN
    RAISERROR('Procedure ParamDefaultProc: you must
              provide a value for the RequiredDate',
              1, 1) WITH LOG
    RETURN
  END
IF @ShipName IS NULL
  BEGIN
    SELECT @ShipName = CompanyName,  @ShipAddress = Address,
           @ShipCity = City,         @ShipPostalCode = PostalCode,
           @ShipCountry = Country
           FROM Customers
           WHERE CustomerID = @CustomerID
  END
```

LISTING 10.2 Continued

```
UPDATE Orders SET
  OrderDate = @OrderDate,           RequiredDate = @RequiredDate,
  ShippedDate = @ShippedDate,       ShipVia = @ShipVia,
  Freight = @Freight,               ShipName = @ShipName,
  ShipAddress = @ShipAddress,       ShipCity = @ShipCity,
  ShipPostalCode = @ShipPostalCode, ShipCountry = @ShipCountry
WHERE
  OrderID = @OrderID
```

The first 2 parameters, the order ID and the customer ID, are required. They are used to select the correct rows in the `Orders` and `Customers` tables within the stored procedure. However, the remaining 10 parameters are all optional. Notice that a couple of them are set to sensible default values (the freight cost and shipper ID), but the remainder are set to `NULL` by default.

Inside the stored procedure, the code can figure out what to do if the user doesn't provide values for some of the parameters. For example, if the order date is not specified, the obvious value to use is the current date, which is provided by the `GETDATE` function in SQL Server. All you have to do is test for the parameter being `NULL` (`IF @OrderDate IS NULL`).

Writing to the Event Log from SQL Server

If the user doesn't provide a value for the `RequiredDate` parameter when he or she executes the stored procedure, you want to prevent the update and flag this as an invalid operation. You can do this by calling the `RAISERROR` method in SQL Server and providing the error message that will be returned to the user. By adding the `WITH LOG` suffix, you force SQL Server to write a message to its own error log file and into the Application section of Windows Event Log as well.

The values used for the `RAISERROR` method are the message to write to the error and event logs, the severity level (which should be between 0 and 18 for non-system-critical messages), and an arbitrary state value that must be between 1 and 127. It's also possible to use the `RAISERROR` method to raise system-defined messages or custom messages stored in the SQL Server `sysmessages` table. SQL Server's Books Online contains more details.

After executing the `RAISERROR` method, the sample page's code simply returns from the stored procedure without updating the database row.

Providing a Default Shipping Address

The sample database contains details of the existing customers in the `Customers` table, so it would seem sensible that when a new order is added, the customer's address details are used by default. In this case, you're *updating* order rows rather than *adding* them, but the code still demonstrates a technique you could use when inserting rows.

If the user does not provide a value for the `@ShipName` parameter (the name of the order recipient), the stored procedure collects the values for all the address columns from the `Customer` table, using the `CustomerID` value provided in the mandatory second parameter to the stored procedure.

Then, finally, the stored procedure executes a SQL statement with a combination of the values that were specified for the parameters, specified as defaults, or calculated within the stored procedure code.

The sample page shown in Figure 10.8 uses this stored procedure. It contains a series of controls where you can enter the values for the parameters and specify whether they are to be set. If a check box is not set, that parameter will not be added to the `Parameters` collection of the `Command` instance, so the default parameter value will be used within the stored procedure.

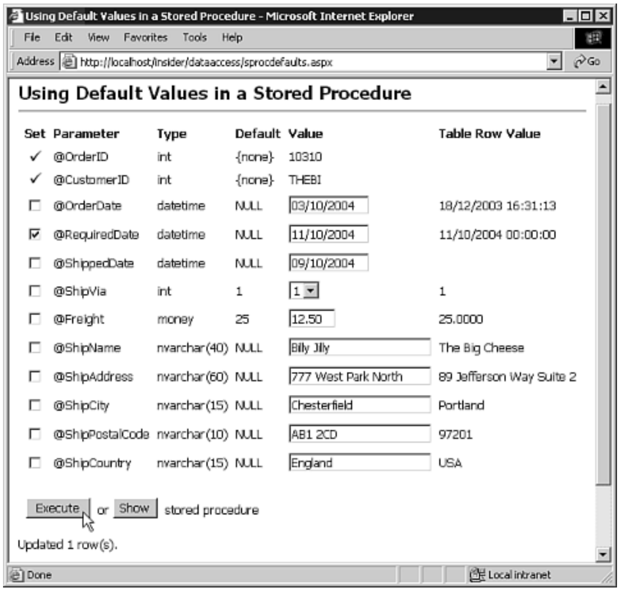


FIGURE 10.8
The sample page that uses the stored procedure with optional parameters.

> **The Sample Page Sets Some of the Values to Sensible Defaults**
>
> **By default, the sample page sets the check box for the `RequiredDate` parameter and fills in some suggested values for this and the other parameters. Even though `RequiredDate` is an optional parameter, a value must be provided to prevent an error from being reported within the procedure. You can click the Show button on the page to view the stored procedure code.**

The right-hand column of the page shows the values currently in the row in the database (for columns that can be edited). When you first load the page, this column is empty. You'll see that it is populated after you execute the stored procedure, so you can tell what effects your settings have had on the row.

The Code for the Stored Procedure Default Values Sample Page

The code used for the sample page contains an event handler routine named `ExecuteSproc` that runs when the Execute button is clicked. Listing 10.3 shows the relevant sections of this code. After you create the `Connection` and `Command` instances and specify that you're working with a stored procedure, you add the two mandatory parameters (the values for which are specified in page-level variables).

Then you test each check box to see if it's set. If it is set, you add a parameter to the `Command` instance with the value collected from the appropriate text box or drop-down list. After you've added all the parameters, you execute the stored procedure and then check whether any rows were updated. If no rows were updated, you display an error message in the page.

LISTING 10.3 The `ExecuteSproc` Routine That Executes the Stored Procedure

```
Sub ExecuteSproc(sender As Object, args As EventArgs)

  ' get connection string, create connection and command
  Dim sConnect As String = ConfigurationSettings.AppSettings( _
                           "NorthwindSqlClientConnectString")
  Dim oCon As New SqlConnection(sConnect)
  Dim oCmd As New SqlCommand("ParamDefaultProc", oCon)
  Dim iRows As Integer

  Try

    ' specify query type, add parameters and execute query
    oCmd.Parameters.Add("@OrderID", iOrderID)
    oCmd.CommandType = CommandType.StoredProcedure
    oCmd.Parameters.Add("@CustomerID", sCustomerID)
    If chkOrderDate.Checked Then
      oCmd.Parameters.Add("@OrderDate", _
                          DateTime.Parse(txtOrderDate.Text))
    End If
    If chkRequiredDate.Checked Then
      oCmd.Parameters.Add("@RequiredDate", _
                          DateTime.Parse(txtRequiredDate.Text))
    End If
    If chkShippedDate.Checked Then
      oCmd.Parameters.Add("@ShippedDate", _
                          DateTime.Parse(txtShippedDate.Text))
    End If
    If chkShipVia.Checked Then
      oCmd.Parameters.Add("@ShipVia", _
                          Integer.Parse(lstShipVia.SelectedValue))
    End If
    If chkFreight.Checked Then
      oCmd.Parameters.Add("@Freight", _
                          Decimal.Parse(txtFreight.Text))
    End If
    If chkShipName.Checked Then
      oCmd.Parameters.Add("@ShipName", txtShipName.Text)
    End If
    If chkShipAddress.Checked Then
```

LISTING 10.3 Continued

```
      oCmd.Parameters.Add("@ShipAddress", txtShipAddress.Text)
    End If
    If chkShipCity.Checked Then
      oCmd.Parameters.Add("@ShipCity", txtShipCity.Text)
    End If
    If chkShipPostalCode.Checked Then
      oCmd.Parameters.Add("@ShipPostalCode", txtShipPostalCode.Text)
    End If
    If chkShipCountry.Checked Then
      oCmd.Parameters.Add("@ShipCountry", txtShipCountry.Text)
    End If

    ' execute procedure and see how many rows were affected
    oCon.Open()
    iRows = oCmd.ExecuteNonQuery()

    'close connection afterwards
    oCon.Close()

    ' display confirmation or error message. If RequiredDate value
    ' not specified the error will be recorded in Windows Event Log
    If iRows > 0 Then
      lblResult.Text = "Updated " & iRows.ToString() & " row(s)."
    Else
      lblResult.Text = "<font color='red'><b>ERROR: </b> No " _
                     & "rows were updated - see the " _
                     & "Application Log in Event Viewer</font>"
    End If

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    ' can call Close more than once if required - no exception
    ' is generated if Connection is already closed
    oCon.Close()
    lblResult.Text = "<b>ERROR: </b>" & oErr.Message

  End Try

  ' now collect values from table and display them in the page
  ' ... code not shown here ...

End Sub
```

Experimenting with the Stored Procedure Default Values Sample Page

To check that the sample page's code works as expected, you can try entering values for the various columns in the row and setting the check boxes to force a parameter to be supplied for that column. For example, if you set the order date, shipper ID, and address details check boxes, you'll see that these columns are updated within the row (see Figure 10.9).

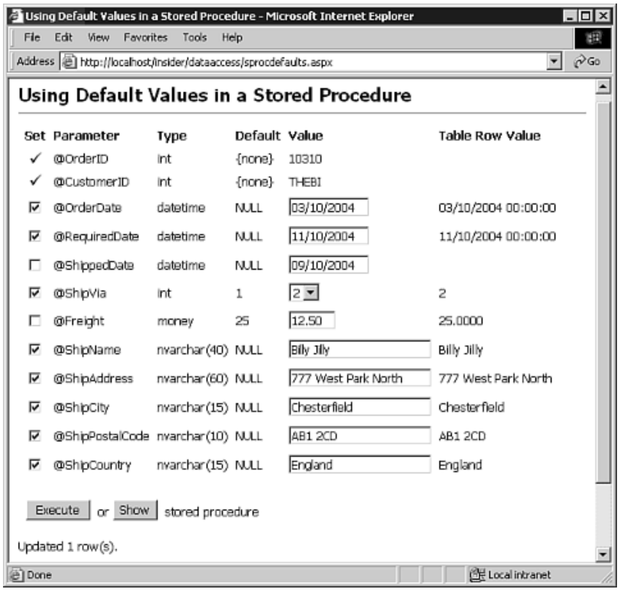


FIGURE 10.9
Updating the order date, shipper, and address columns.

However, if you then clear the check box for the customer name (`@ShipName`) and execute the stored procedure again, you'll see that the values in the `Customers` table for this customer are collected and used to update the row (see Figure 10.10).

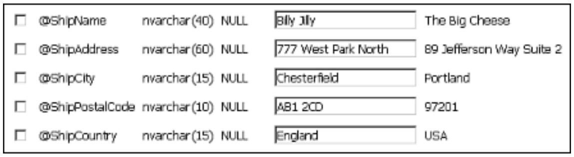


FIGURE 10.10
Using the default address details if they are not specified as parameters.

Finally, you can try clearing the check box for the `@RequiredDate` parameter and executing the stored procedure again. You'll see an error message displayed at the foot of the page. If you select Start, Programs, Administrative Tools; open Event Viewer; and look in the Application Log section, you'll see the entry that the stored procedure creates (see Figure 10.11).

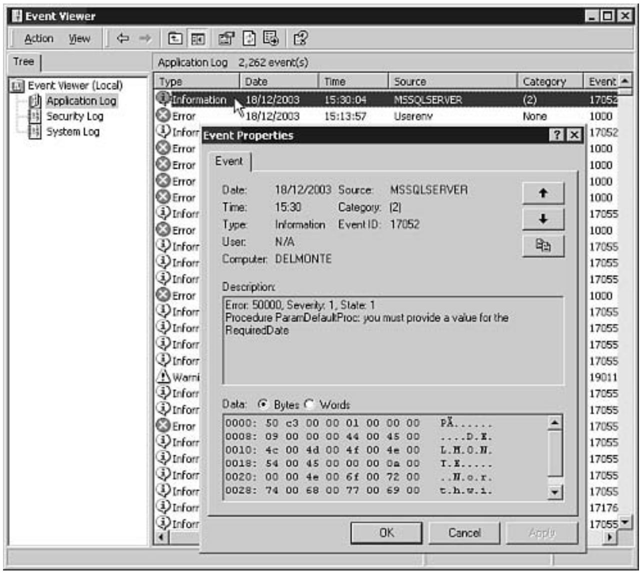


FIGURE 10.11
The message written to the event log when no `RequiredDate` value is provided.

Filling a `DataSet` Instance With and Without a Schema

ADO.NET developers take for granted the ease with which they can fill a `DataSet` instance from a database. To do this, you simply create an empty `DataSet` instance, create a `Connection` instance, and specify a SQL statement or stored procedure. Then you create a `DataAdapter` instance from these and call its `Fill` method to pull the data from the database and push it into the data set.

However, when you think about it, there's a lot going on here. The internal `DataSet` code has to figure out the schema of the database table(s) and build this structure. And what happens if the table has a primary key defined or if there are relationships between the tables in the database? What about if there are `NULL` values in some rows or orphan rows in a child table?

The same questions apply when you fill a `DataSet` instance from an XML document. Where does the primary key come from, if there is one? And because XML documents are often hierarchical in nature, how does the internal `DataSet` code know what tables and columns to create, and what does it do when values are missing for some of the columns?

Loading the Schema for a `DataSet` Instance

In response to most of the concerns described in the preceding section, many developers load a schema first, before they attempt to load either relational data (via a `DataAdapter` instance) or an XML document. The schema causes the `DataSet` instance to create the required tables(s), with columns that are of the required data type, size, and precision. The schema can also force the internal `DataSet` code to create the primary keys and foreign keys for the tables, establishing the `DataRelation` objects that reference the relationships between the tables.

What is the most efficient way to do this? The internal `DataSet` code seems to cope perfectly well without a schema in most cases; the only common exception is irregularly structured XML documents. The following sections look at an example that gives you a chance to compare the performance on your system.

The `DataAdapter.MissingSchemaAction` Property

Do you usually specify a value for the `MissingSchemaAction` property of the `DataAdapter` instance when you fill a data set? If you create the structure from a schema, what happens if the data you load subsequently doesn't match the schema? For example, there may be extra columns in the tables that are returned by the SQL statement or stored procedure, or there may be extra nested elements in an XML document that you use to load your `DataSet` instance.

By default, the internal `DataSet` code will automatically add to its tables any extra columns it requires, and it populates these from the data that is used to fill or load the `DataSet` instance (regardless of whether you use the `Fill` method for relational data or load an XML document). However, you can control this process yourself by setting the `MissingSchemaAction` property of the `DataAdapter` instance to one of the values shown in Table 10.1.

> **Filling a Data Set when the Data Contains Extra Column Elements**
>
> **It's possible for an irregularly structured XML document to have extra nested elements that do not match the schema you use. In this case, the default behavior of the `DataSet` instance is to add any columns (and tables) required to load all the data—just as if there were extra columns in relational data. However, it's also possible that an XML document has nested elements missing (that is, omitted) so that there is no data available to fill some of the columns in some of the rows in a table in the `DataSet` instance. In that case, the values in these columns are all set to `NULL`.**

TABLE 10.1

The Values from the `MissingSchemaAction` Enumeration

Value	Description
`Add`	This is the default. Tables and columns that occur in the source data are added to the `DataSet` instance. Only the data type of the column is set automatically. Other metadata, such as the primary key, column size, and precision, is not set.
`AddWithKey`	Tables and columns that occur in the source data are added to the `DataSet` instance. All metadata about the columns is loaded, including the primary key, column size, and precision.
`Ignore`	Any tables or columns not already in the `DataSet` instance are ignored and are not added. Using this value is a good way to prevent the contents of the `DataSet` instance from varying from a predefined structure.
`Error`	An exception is raised if a table or column is found in the source data that does not already exist in this `DataSet` instance. Using this value is a good way to detect when the source data varies from the predefined structure.

The Sample Page for Filling a `DataSet` Instance

You can use the sample page discussed in this section in several ways. It contains a function named `FillDataSet` that generates a `DataSet` instance containing three related tables. This is much the same code as is used several times in Chapter 4. The data is extracted from the `Customers`, `Orders`, and `Order Details` tables in the Northwind database, and the code adds two relationships, named `CustOrders` and `OrdersODetails`, to the `DataSet` instance (see Figure 10.12).

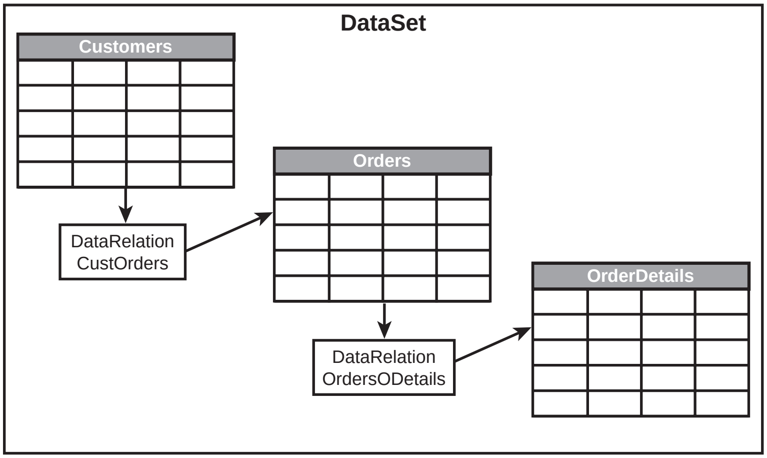


FIGURE 10.12
The structure of the `DataSet` instance for an example of filling a data set.

Your major aim for the routine that generates the `DataSet` instance in this example is that you want to be able to compare the performance and results when you load a schema first and when you do not. You also want to be able to compare the results when you use different values for the `MissingSchemaAction` property of the `DataAdapter` instance. After you create a new empty `DataSet` instance, you test the value of a parameter named `bLoadSchema`. If this value is `True`, you load a schema from disk into the `DataSet` instance:

```
If bLoadSchema = True Then
  Dim sSchemaFile As String _
      = Request.MapPath(Request.ApplicationPath) _
      & "\dataaccess\datasetschema.xsd"
  oDataSet.ReadXmlSchema(sSchemaFile)
End If
```

After you create the `Connection`, you can create the `DataAdapter` instance and set the `MissingSchemaAction` property. The value of this property is taken from a drop-down list control named `lstMissingSchema` in the page:

```
Dim oDA As New OleDbDataAdapter(sCustSQL, oConnect)
oDA.MissingSchemaAction = lstMissingSchema.SelectedValue
```

Then you can fill that `DataSet` instance with the three tables you want. Afterward, you add the two relationships named `CustOrders` and `OrdersODetails` to the `DataSet` instance:

```
If bLoadSchema = False Then
  ' create relations between the tables
  ' ... as in previous examples ...
End If
```

However, you do this only if you didn't load a schema first because the schema declares, and will have created, the relationships.

Viewing the Schema

The sample page provided with the samples for this book contains a routine named ShowSchema. This routine uses the FillDataSet function to create and populate a DataSet instance and then displays the schema in the page so that you can see the result. The FillDataSet function is called with the bLoadSchema parameter set to False so that the internal schema generated within the DataSet instance is based on the data it loads and the current setting of the MissingSchemaAction property.

Listing 10.4 shows the ShowSchema routine. You can see the routine displays the schema only if the MissingSchemaAction property (as specified in the drop-down list named lstMissingSchema) has the value 1 (Add) or 4 (AddWithKey). If you use any other value for the MissingSchemaAction property and don't load a schema first, you won't get any tables generated in the DataSet instance. (Look back at Table 10.1 if you're not sure why this should be the case.)

The routine named CreateSQLStatements that is called in Listing 10.4 simply creates the SQL statements that the FillDataSet function uses; the CreateSQLStatements routine isn't shown in Listing 10.4. After the routine fills the DataSet instance, the GetXmlSchema method is called to get the schema as a String value, and the code HTML encodes it and inserts it into a Label control on the page.

LISTING 10.4 The ShowSchema Routine That Displays a Schema

```
Sub ShowSchema(sender As Object, e As EventArgs)

  If lstMissingSchema.SelectedValue = 1 _
  Or lstMissingSchema.SelectedValue = 4 Then
    CreateSQLStatements()
    Dim oDS As DataSet = FillDataSet(False)
    lblSchema.Text = "<pre>" _
      & Server.HtmlEncode(oDS.GetXmlSchema()) & "</pre>"
  Else
    lblSchema.Text = "Cannot create schema dynamically " _
      & "for Ignore or Error values"
  End If

End Sub
```

Figure 10.13 shows the sample page in action. Clicking the View Schema button calls the ShowSchema routine and shows the result in the page.

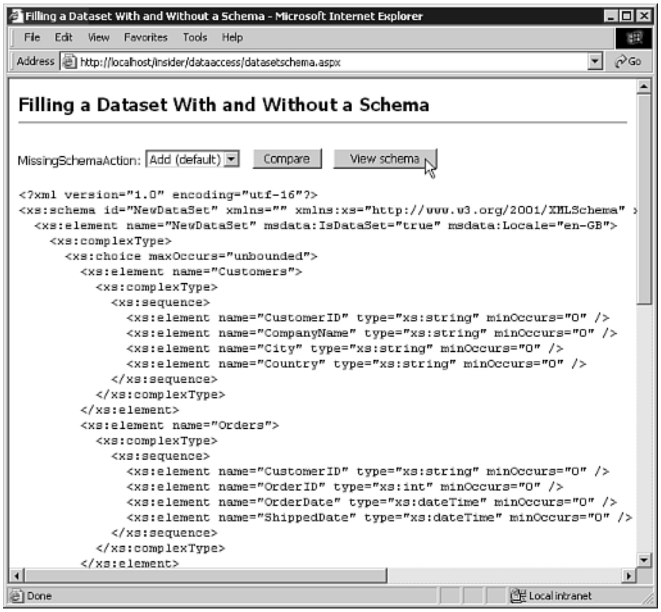


FIGURE 10.13
Viewing the schema for the `DataSet` instance when `MissingSchemaAction` is set to Add.

The Schema for `MissingSchemaAction.Add`

Listing 10.5 contains two extracts from the schema displayed in Figure 10.13, when `MissingSchemaAction` is set to `Add`. The first section shows the definition of the `Customers` table in the `DataSet` instance, and it's obvious that the only information it provides is the column name and the data type. The `minOccurs` attribute indicates that values for all the columns are optional. In other words, they could be `NULL` in the database table, and the equivalent elements could be omitted from an XML representation of the data.

LISTING 10.5 The Schema Generated when `MissingSchemaAction` Is Set to Add

```
<xs:element name="Customers">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
   <xs:element name="CompanyName" type="xs:string" minOccurs="0" />
   <xs:element name="City" type="xs:string" minOccurs="0" />
   <xs:element name="Country" type="xs:string" minOccurs="0" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
...
... Orders and Order Details tables here ...
...
<xs:unique name="Constraint1">
  <xs:selector xpath=".//Customers" />
```

LISTING 10.5 Continued

```
  <xs:field xpath="CustomerID" />
</xs:unique>
<xs:unique name="Orders_Constraint1"
           msdata:ConstraintName="Constraint1">
  <xs:selector xpath=".//Orders" />
  <xs:field xpath="OrderID" />
</xs:unique>
<xs:keyref name="OrdersODetails" refer="Orders_Constraint1">
  <xs:selector xpath=".//OrderDetails" />
  <xs:field xpath="OrderID" />
</xs:keyref>
<xs:keyref name="CustOrders" refer="Constraint1">
  <xs:selector xpath=".//Orders" />
  <xs:field xpath="CustomerID" />
</xs:keyref>
```

The `FillDataSet` function creates the two relationships within the `DataSet` instance that link the three tables. At the end of the schema are the `xs:unique` and `xs:keyref` elements, which represent these relationships. To allow the relationships to exist, there must be a unique constraint on the parent column, and such constraints are specified for the `Customers` and `Orders` tables by the `xs:unique` elements, which specify the path (the table name) and the name of the column for each constraint.

The `xs:keyref` elements can then specify the name of the relationship, a reference to the unique constraint that identifies the parent column, and the path and name of the child column. Bear in mind that these constraints are created by the relationships added to the `DataSet` instance and are not implemented by the `Fill` method. If you didn't create the relationships, there would be no `xs:unique` and `xs:keyref` elements. In other words, if you don't create the relationships, all the columns in the table will be optional and not forced to contain unique values.

The Schema for *MissingSchemaAction.AddWithKey*

Listing 10.6 shows the definition of the `Customers` table in the schema when `MissingSchemaAction` is set to `AddWithKey`. This time, the declaration of each column contains an `xs:restriction` element that defines the data type and the size of the column. (For the string values shown here, the size of the column is the number of characters.)

LISTING 10.6 The Customers Table Definition Generated when `MissingSchemaAction` Is Set to `AddWithKey`

```
<xs:element name="Customers">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CustomerID">
        <xs:simpleType>
          <xs:restriction base="xs:string">
```

LISTING 10.6 Continued

```
            <xs:maxLength value="5" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="CompanyName">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="40" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="City" minOccurs="0">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="15" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="Country" minOccurs="0">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="15" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

For the `Orders` and `OrderDetails` tables, the schema also contains information about the `IDENTITY` columns. For example, the definition of the `OrderID` column in the `Orders` table specifies it to be an *auto-increment* or `IDENTITY` column, of type `int` (`Integer`) and specifies it to be read-only:

```
<xs:element name="OrderID" msdata:ReadOnly="true"
    msdata:AutoIncrement="true" type="xs:int" />
```

When you use `MissingSchemaAction.Add`, there is no indication at all of the primary keys for the tables—just the specification of the unique column constraints generated by the relationships added to the `DataSet` instance. However, with `MissingSchemaAction.AddWithKey`, the final section of the schema specifies the primary keys of the `Customers` and `Orders` tables, using the `msdata:PrimaryKey` attribute. You can see these constraints in Listing 10.7.

LISTING 10.7 The DataSet Instance Constraints Generated when MissingSchemaAction Is Set to AddWithKey

```
<xs:unique name="Constraint1" msdata:PrimaryKey="true">
  <xs:selector xpath=".//Customers" />
  <xs:field xpath="CustomerID" />
</xs:unique>
<xs:unique name="Orders_Constraint1"
           msdata:ConstraintName="Constraint1"
           msdata:PrimaryKey="true">
  <xs:selector xpath=".//Orders" />
  <xs:field xpath="OrderID" />
</xs:unique>
<xs:keyref name="OrdersODetails" refer="Orders_Constraint1">
  <xs:selector xpath=".//OrderDetails" />
  <xs:field xpath="OrderID" />
</xs:keyref>
<xs:keyref name="CustOrders" refer="Constraint1">
  <xs:selector xpath=".//Orders" />
  <xs:field xpath="CustomerID" />
</xs:keyref>
```

Comparing Performance With and Without a Schema

The final section of code in the sample page is a routine named DoTest that runs when the Compare button is clicked. Listing 10.8 shows this routine, which declares some variables you'll need and calls the CreateSQLStatements routine used earlier in this chapter to create the SQL statements for the FillDataSet routine. Then the code calls the FillDataSet method a number of times, with and without a schema, and times each set of operations to see how they compare.

LISTING 10.8 The DoTest Routine to Compare Performance With and Without a Schema

```
Sub DoTest(sender As Object, e As EventArgs)

  ' declare local variables
  Dim iCount As Integer = 100
  Dim iLoop As Integer
  Dim oDS As DataSet
  Dim dStart As DateTime
  Dim dDiff1, dDiff2 As TimeSpan

  CreateSQLStatements()

  ' load DataSet with schema
  Trace.Write("With Schema", "Start")
  dStart = DateTime.Now
  For iLoop = 1 To iCount
```

LISTING 10.8 Continued

```
    oDS = FillDataSet(True)
  Next
  dDiff1 = DateTime.Now.Subtract(dStart)
  Trace.Write("With Schema", "End")
  lblResult.Text &= "Loaded DataSet with schema " _
    & iCount.ToString() & " times in " _
    & dDiff1.TotalMilliseconds.ToString() & " ms.<br />"

  ' load DataSet without schema - can't do it
  ' when MissingSchemaAction is Ignore or Error
  If lstMissingSchema.SelectedValue = 1 _
  Or lstMissingSchema.SelectedValue = 4 Then
    Trace.Write("Without Schema", "Start")
    dStart = DateTime.Now
    For iLoop = 1 To iCount
       oDS = FillDataSet(False)
    Next
    dDiff2 = DateTime.Now.Subtract(dStart)
    Trace.Write("Without Schema", "End")
    lblResult.Text &= "Loaded DataSet without schema " _
      & iCount.ToString() & " times in " _
      & dDiff2.TotalMilliseconds.ToString() & " ms.<br />"

    ' calculate difference
    Dim fRatio As Decimal = (dDiff1.TotalMilliseconds _
      - dDiff2.TotalMilliseconds) / dDiff2.TotalMilliseconds
    lblResult.Text &= "With schema is " & Math.Abs(fRatio).ToString("p")
    If dDiff1.TotalMilliseconds > dDiff2.TotalMilliseconds Then
      lblResult.Text &= " slower."
    Else
      lblResult.Text &= " faster."
    End If

  End If

End Sub
```

You save the current system time in a variable; execute the `FillDataSet` routine `iCount` number of times, specifying that it should load the schema first; and then calculate the number of milliseconds that have elapsed. As well as displaying this in a `Label` control on the page, you also write start and end messages to the `Trace` object. You'll see these in the trace output if you turn on tracing for the page (by adding `Trace="True"` to the `Page` directive).

You then repeat the process, but this time you instruct the FillDataSet routine to not load a schema first. And, of course, you can do this only when the MissingSchemaAction property is set to Add or AddWithKey. If it is set to Error or Ignore, there will be no data in the DataSet instance if no schema was loaded first. After all this, you calculate the percentage difference in the times taken and display this in the page.

The Results of Comparing Performance With and Without a Schema

Figure 10.14 shows the result of comparing performance with and without a schema on one of our (rather aging) test servers. You can see that when MissingSchemaAction is set to Add, loading a schema turns out to be on average around 25% slower than simply loading the data into an empty DataSet instance. When you set MissingSchemaAction to AddWithKey, loading a schema first is around 15% slower on average.

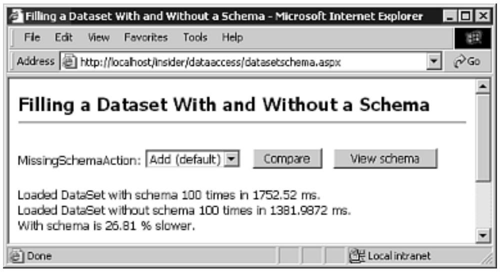


FIGURE 10.14 The results of comparing load times with and without a schema.

This isn't quite what you would expect, especially because the general opinion seems to be that loading a schema first gives better performance. Of course, performance will vary wildly, depending on a whole raft of factors such as disk access times for loading the schema, memory and resources availability on the server, and where the database is located and the network connection speed. However, we repeated the test on two other machines, including one with SCSI rather than IDE disks, and the results were broadly similar.

> **Let Us Know the Result on Your Servers**
>
> **You can use the code and techniques shown in this chapter to repeat the test on your own systems, and you might get very different results. We'd be pleased to hear what you discover. You can post your comments and results at our Web site:** www.daveandal.net.

Remember that loading the schema first always sets the primary keys and the size and precision of the columns, whereas that information is not added to the DataSet instance when MissingSchemaAction is set to Add and no schema is loaded. It is possible to add the primary key information, set the size and precision of the columns, and add extra columns, if required, after the data has been loaded into the DataSet instance.

BEST PRACTICE

Using a DataReader Object when You Don't Need a DataSet Object

It's easy to get into the habit of using DataSet objects for all your projects. However, remember that in many cases you don't actually need all the extra features that this object has compared to the DataReader object. The times when you absolutely require a DataSet instance include the following:

- **When you want to remote the data to another server or client**
- **When you need to store multiple tables and perhaps the relationships between them**
- **When you need to preserve the full metadata for each column, such as primary keys, default values, and constraints**
- **When you intend to perform a subsequent update to the source data**
- **When you are using sorting or paging in an ASP.NET DataGrid control**

For other tasks, especially simple server-side data binding, the subsequently lower processing and memory overhead of the DataReader class can substantially improve performance.

Writing Provider-Independent Data Access Code

A regular inquiry from ADO.NET developers is how you go about writing code that can easily be converted to use a different data provider. For example, you might have built your pages to access a range of database types, using the OLE DB provider and the classes from the `System.Data.OleDb` namespace. However, if you subsequently decide to use SQL Server as your data source, you can benefit from using the `System.Data.SqlClient` classes with the native TDS provider for SQL Server. But this means changing all the references and classnames in your code. To do that, you could do a search-and-replace operation. However, you could instead write code that is provider independent. The only downside of this is that it is marginally less efficient because you have to use dynamic linking to the classes at runtime, rather than the static linking approach that is used when you specify the classname directly.

The sample page shown in Figure 10.15 demonstrates the use of provider-independent data access code. The drop-down list allows you to select any of the three provider types (`SqlClient`, `OleDb`, or `Odbc`) and then execute a SQL statement that extracts some values from the database. As long as you are running the page on your local server (`http://localhost`), you'll see the connection string displayed as well.

Dynamically Instantiating a .NET Framework Class

To instantiate classes dynamically at runtime, as you need to do in this example, you can take advantage of the *remoting* technology that is built into the .NET Framework. Remoting allows you to call the `CreateInstance` method of the static `Activator` object that is exposed by the remoting system. You specify as parameters the fully qualified namespace name and the name of the class you want; the remoting system returns a handle to a wrapped instance of that class.

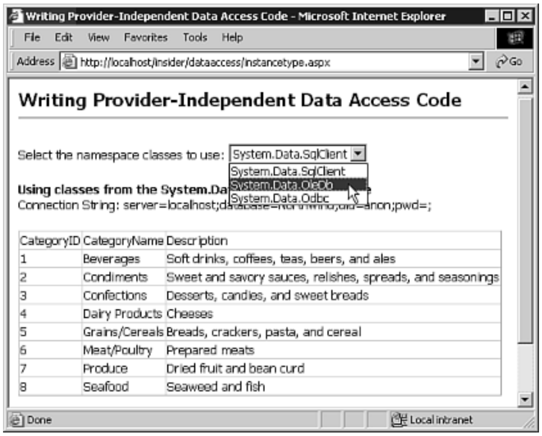


FIGURE 10.15 A demonstration page for provider-independent data access code.

When the object handle is returned, you then call its `Unwrap` method. This instantiates the class and returns a reference to the resulting object instance. To use these methods, you have to import the `System.Runtime.Remoting` namespace into your ASP.NET page, along with any namespaces for other classes that you use through static binding. However, you don't have to import the namespaces for the classes that you instantiate dynamically through the remoting system.

> **Why Remoted Instances Are Wrapped**
>
> The instance is wrapped (yes, this is the correct technical term) so that it is not instantiated automatically. Remember that the remoting system is designed to allow objects to be passed from one application domain to another (for example, from one application, across the network, to a remote client application). The reference to the class may have to pass through intermediate application domains on its way to the client, and this means it can avoid being instantiated within those domains.

This example uses a `DataReader` instance to extract the data rows and populate the `DataGrid` control on the page, so you need to create a `Connection` instance and a `Command` instance from the namespace selected in the drop-down list. However, exactly the same principle applies if you want to use a `DataSet` instance, in which case you'd create a `DataAdapter` instance from the appropriate namespace. Plus, of course, you might need to create `Parameter` objects or objects of other types as well.

The Code in the Provider-Independent Data Access Sample Page

The code in the sample page is broken into several routines and a page-level variable that holds the fully qualified name of the `System.Data` assembly. This assembly contains the `SqlClient`, `OleDb`, and `Odbc` namespaces, from where you create `Connection` and `Command` objects. Listing 10.9 shows the `Import` directives, the page-level variable, the `Page_Load` event handler, and the `ShowData` routine.

LISTING 10.9 The Code for the Provider-Independent Data Access Example

```
<%@Import Namespace="System.Data" %>
<%@Import Namespace="System.Runtime.Remoting" %>

<script runat="server">

' assembly details for System.Data in version 1.1
Dim sFQName As String = "System.Data, Version=1.0.5000.0, " _
    & "Culture=neutral, PublicKeyToken=b77a5c561934e089"


Sub Page_Load()

  ' display data using SqlClient classes first time
  If Not Page.IsPostback Then
    ShowData("Sql")
  End If

End Sub


Sub ShowData(sTypePrefix As String)

  ' set values of namespace and class prefix
  Dim sNameSpace As String = sTypePrefix
  If sNameSpace = "Sql" Then
    sNameSpace = "SqlClient"
  End If

  ' bind result to DataGrid to display data
  dgr1.DataSource = GetDataReader(sNameSpace, sTypePrefix)
  dgr1.DataBind()

End Sub
```

Notice that the fully qualified name of the assembly contains not only the assembly name (the DLL name without the file extension) but also the version, culture, and public key token values. You can obtain these values from the .NET Configuration Wizard that is installed with the .NET Framework. You simply select Start, Programs, Administrative Tools, Microsoft .NET Framework 1.1 Configuration and open the Assembly Cache section by clicking the link in the left pane of the window. All the installed assemblies are listed in the right pane (see Figure 10.16).

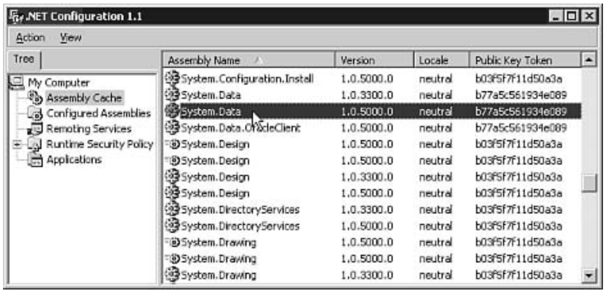


FIGURE 10.16
Using the .NET Framework 1.1 Configuration utility.

The `Page_Load` event handler simply calls the `ShowData` routine and passes the parameter value `"Sql"` to it the first time the page is loaded. The `ShowData` routine uses this value as the type prefix and the class prefix, with the exception that it has to change the value `"Sql"` for the namespace prefix to `"SqlClient"`. It uses these two values when it calls another routine named `GetDataReader`. This routine returns a `DataReader` instance open against the `Categories` table in the Northwind database, which is then used to populate a `DataGrid` control located elsewhere in the sample page.

Listing 10.10 shows the `GetDataReader` routine. This is a function, and it returns an `Object` type because you don't know at design time what type of `DataReader` instance you'll be creating. After declaring the SQL statement you'll be using, you collect the connection string from the `web.config` file. Of course, this string needs to be specific to the type of data access provider you're using. The `web.config` file for this book's examples contains all three connection strings, and you can create the key for the appropriate one by using the namespace prefix passed to the `GetDataReader` routine.

Next, you create the full class path and prefix for the classname as a `String` (for example `"System.Data.SqlClient.Sql"`). Then you can call the `CreateInstance` method of the `Activator` object, specifying the fully qualified namespace name (from the page-level variable `sFQName`) and the full classname. (In the first instance you want a `Connection` object.)

The `ObjectHandle` you get back references the wrapped `Connection` object from the relevant namespace, so you call the `Unwrap` method of the handle to get back the instantiated object you want. Note that, again, this has to be declared as an `Object` type. Then you repeat the process to get a `Command` object from the same namespace.

Error Messages when Creating Class Instances Dynamically

One side effect of creating class instances dynamically is that you often lose the precise error messages that the classes would return. Because the variables that reference the `Connection` and `Command` instances returned by the remoting system have to be declared as `Object` types, any error will most likely return the generic error message "ERROR: Exception has been thrown by the target of an invocation."

LISTING 10.10 The Provider-Independent `GetDataReader` Routine

```
Function GetDataReader(sNameSpace As String, _
                       sTypePrefix As String) As Object

  Dim sSQL As String = "SELECT * FROM Categories"
  Dim sCon As String = _
    ConfigurationSettings.AppSettings("Northwind" & sNameSpace _
                                      & "ConnectString")

  ' create class prefix, e.g. "System.Data.SqlClient.Sql"
  Dim sClassPrefix As String = "System.Data." & sNameSpace _
                            & "." & sTypePrefix

  ' create instance of provider-specific Connection class
  ' uses default constructor and returns a handle to
  ' the "wrapped" object instance via remoting
  ' requires import of System.Runtime.Remoting namespace
  Dim oHandle As ObjectHandle
  oHandle = Activator.CreateInstance(sFQName, _
                      sClassPrefix & "Connection")

  ' unwrap object and assign to local variable
  Dim oCon As Object = oHandle.Unwrap()

  ' create instance of provider-specific Command class
  oHandle = Activator.CreateInstance(sFQName, _
                      sClassPrefix & "Command")

  ' unwrap object and assign to local variable
  Dim oCmd As Object = oHandle.Unwrap()

  Try

    ' assign values to properties of new objects and execute
    oCon.ConnectionString = sCon
    oCmd.Connection = oCon
    oCmd.CommandText = sSQL
    oCon.Open()
    Return oCmd.ExecuteReader(CommandBehavior.CloseConnection)

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    oCon.Close()
    lblResult.Text &= "ERROR: " & oErr.Message & "<br />"
```

LISTING 10.10 Continued

```
  End Try

End Function
```

When you have the `Connection` and `Command` objects, you can open the connection and return the `DataReader` instance that you get back from executing the command. You use the `CommandBehavior.CloseConnection` parameter to ensure that the `Connection` instance is closed when the `DataReader` instance goes out of scope after being bound to the `DataGrid` control in the page.

Updating Multiple Rows by Using Changed Events

The rest of this chapter looks at an interesting situation with regard to updating data with a `DataGrid` control (or, in fact, any other list control that supports templates—such as the `Repeater` and `DataList` controls). We include this example after seeing a question on the ASP.NET groups as to whether it is possible to handle the changed events for controls located within the output of a `DataGrid` control and do anything useful with them.

When you use a `DataGrid` control or `DataList` control for inline editing of the data rows in ASP.NET, as demonstrated in Chapter 4, the usual technique for updating the data source is to fire off individual SQL statements (or execute stored procedures) on each postback. This means that only one row can be edited at a time because a postback is required to change the `EditItemIndex` property of the `DataGrid` control to show a different row in edit mode, and the values in the controls in the previous row are lost.

However, the change event for a control such as the `TextBox` control does not cause a postback (unless you set the `AutoPostback` property to `True`). All the `TextChanged` events for all the constituent members of the `DataGrid` control are raised one after the other when the next postback occurs. This means that you can take advantage of this feature to allow users to edit multiple rows in a `DataGrid` control or `DataList` control and then perform all the updates in one go on the server afterward.

Figure 10.17 shows the sample page we provide with the samples for this book (which you can find at `www.daveandal.net/books/6744/`). It displays six rows from the `Products` table in the Northwind sample database. Each value except the row key is displayed in an edit control. There are text boxes for the product name and price, and there is a check box for the Discontinued column.

Figure 10.18 shows what happens after some of the values are changed and the Update button is clicked. You can see that the reduced-sugar aniseed syrup is more expensive than the standard variety, that Chai is now discontinued, that Chang has suffered more than usual price inflation (perhaps due to a bad harvest in Indonesia), and that Chef Anton has really gone to town in naming his latest gumbo mix flavor.

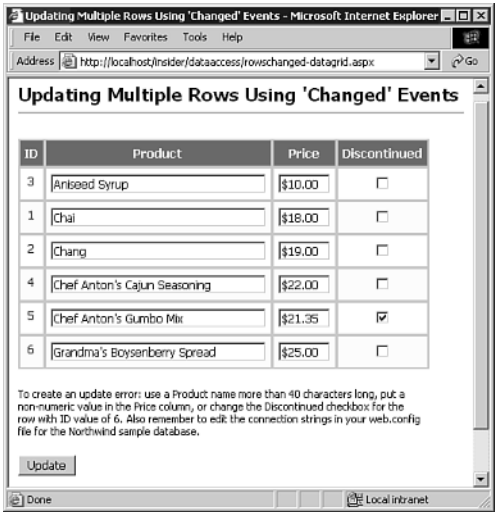


FIGURE 10.17 A sample page that demonstrates editing multiple rows in a `DataGrid` control.

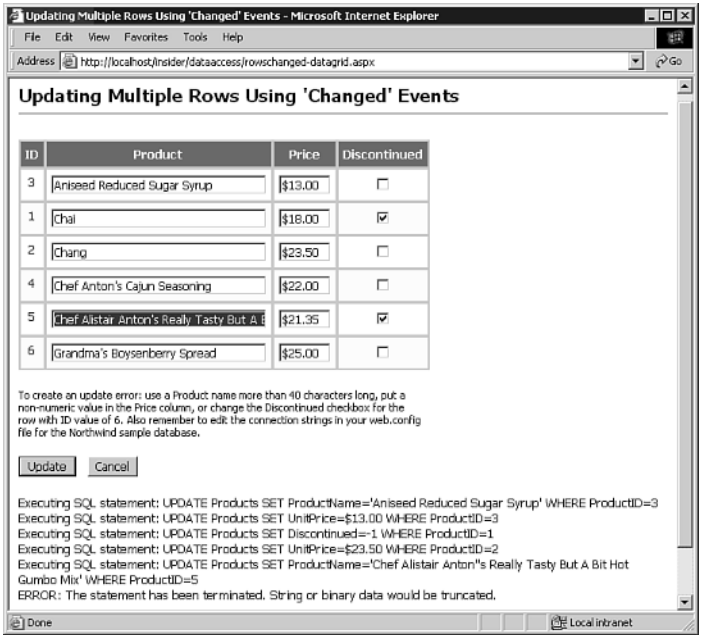


FIGURE 10.18 Highlighting the control where an error occurred after a postback.

At the bottom of the page, you can see that the single postback resulted in five SQL `UPDATE` statements being generated and executed against the database—one for each change to the controls in the page. These updates occur in response to the four `TextChanged` events for the text boxes that were edited and the single `CheckedChanged` event for the check box in the Discontinued column.

The important point to note is that the last of these statements failed because the value provided for the ProductName column was too long for the column in the database. Also note—and here's the clever part—that the control containing the error is selected (that is, receives the input focus). Moreover, if this book were in color, you'd be able to see that the text within the text box is now colored red; if there is more than one error, the focus moves to the first one, but all the fields that generated errors are highlighted in red.

If an error is detected during an update, a Cancel button appears on the page (refer to Figure 10.18). You can use the Cancel button to abandon the changes that resulted in an error and that were not applied. All it does is repopulate the `DataGrid` control from the database to restore the current values and remove any highlighting and color changes from the controls.

The declaration of the `DataGrid` control is reasonably simple. You need to handle the `ItemDataBound` event for every row, as you'll see later in this chapter, so you specify a routine that will be executed when the event is raised. You obviously don't want the `DataGrid` control to autogenerate the columns either because you'll declare them yourself so that you get edit controls in every row:

```
OnItemDataBound="BindRowData"
AutoGenerateColumns="False"
```

Listing 10.11 shows the `<Columns>` section of the `DataGrid` control, which contains just an `<ItemTemplate>` section for each column. You won't be switching into edit mode, so this is all you need. The first column is the product ID, and it is read-only, so it's just displayed as text. Each of the next two columns contains a `TextBox` control, with the `Text` property bound to the value in the source data row. The third column contains a `CheckBox` control, with its `Checked` property bound to the value in the source data row.

Notice that you specify event handlers for the `TextChanged` events of the `TextBox` controls and the `CheckedChanged` event of the `CheckBox` control. These event handlers will push the changed values into the database following a postback.

LISTING 10.11 The Column Declarations for the `DataGrid` Control

```
<Columns>

  <asp:TemplateColumn HeaderText="ID"
       HeaderStyle-HorizontalAlign="Center"
       ItemStyle-HorizontalAlign="Center">
    <ItemTemplate>
      <%# Container.DataItem("ProductID") %>
    </ItemTemplate>
  </asp:TemplateColumn>

  <asp:TemplateColumn HeaderText="Product"
       HeaderStyle-HorizontalAlign="Center"
       ItemStyle-HorizontalAlign="Right">
    <ItemTemplate>
```

LISTING 10.11 Continued

```
      <asp:Textbox id="txtProductName" Columns="40"
           Text='<%# Container.DataItem("ProductName") %>'
           OnTextChanged="ProductNameChanged"
           runat="server" />
    </ItemTemplate>
  </asp:TemplateColumn>

  <asp:TemplateColumn HeaderText="Price"
       HeaderStyle-HorizontalAlign="Center"
       ItemStyle-HorizontalAlign="Right">
    <ItemTemplate>
      <asp:Textbox id="txtPrice" Columns="6"
           Text='<%# DataBinder.Eval(Container.DataItem, _
                     "UnitPrice", "${0:F2}") %>'
           OnTextChanged="PriceChanged"
           runat="server" />
    </ItemTemplate>
  </asp:TemplateColumn>

  <asp:TemplateColumn HeaderText="Discontinued"
       ItemStyle-HorizontalAlign="Center">
    <ItemTemplate>
      <asp:Checkbox id="chkDiscontinued"
           Checked='<%# Container.DataItem("Discontinued") %>'
           OnCheckedChanged="DiscontinuedChanged"
           runat="server" />
    </ItemTemplate>
  </asp:TemplateColumn>

</Columns>
```

The Edit and Cancel Buttons

No buttons or links are required within the `DataGrid` control to initiate postbacks. Normally, you'd use an `EditCommandColumn` column or a `ButtonColumn` column that adds an edit link or button to each row to switch that row into edit mode. However, all the rows are already effectively in edit mode in this example, and you just need a single button that will cause a postback—whereupon you perform all the updates in one go. So you just add an ordinary `Button` control to the foot of the page:

```
<asp:Button Text="Update" id="btnUpdate" runat="server" />
```

The Cancel button shown in Figure 10.18 is visible only when an update error is detected. This is another `Button` control, with its `Visible` property set to `False` so that it does not appear in the page by default. If you turn off viewstate for the button, it will appear only when you set its `Visible` property to `True` during a postback, and then it will automatically disappear again on the next postback:

```
<asp:Button Text="Cancel" id="btnRefresh" Visible="False"
     OnClick="DoRefresh" EnableViewState="False" runat="server" />
```

Populating the `DataGrid` Control

The data to populate the `DataGrid` control is extracted from the database by using a simple SQL statement (you would, of course, use a stored procedure here in the real world), stored in a `DataSet` instance, and then bound to the `DataGrid` control. However, you could equally well use the `DataReader` class to populate the control instead.

In this example, the data access code is in a routine named `BindDataGrid`, which you call from the `Page_Load` event the first time the page is loaded. You also call this routine from the Cancel button via an event handler named `DoRefresh`. The `BindDataGrid` routine simply repopulates the `DataGrid` control with the values in the underlying table in the database. Listing 10.12 shows the `Page_Load` and `DoRefresh` event handlers and the `BindDataGrid` routine.

LISTING 10.12 Populating the `DataGrid` Control

```
Sub Page_Load()

  If Not Page.IsPostback Then
    BindDataGrid()
  End If

End Sub


Sub DoRefresh(sender As Object, e As EventArgs)

  ' refill grid from original data source
  BindDataGrid()

End Sub


Sub BindDataGrid()

  ' declare SQL statement to fill table
  Dim sProducts As String _
    = "SELECT ProductID, ProductName, UnitPrice, Discontinued " _
```

LISTING 10.12 Continued

```
    & "FROM Products " _
    & "WHERE ProductID <= 6 ORDER BY ProductName"

  Dim sConnect As String = ConfigurationSettings.AppSettings( _
                           "NorthwindOleDbConnectString")

  Dim oConnect As New OleDbConnection(sConnect)

  ' create new DataSet and fill from database
  Dim oDS As New DataSet()

  Try

    Dim oDA As New OleDbDataAdapter(sProducts, oConnect)
    oDA.Fill(oDS, "Products")

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text &= oErr.Message & "<br />"

  End Try

  ' bind DataGrid to Products table
  dgr1.DataSource = oDS
  dgr1.DataMember = "Products"
  dgr1.DataBind()

End Sub
```

Handling the `ItemDataBound` Event

You include the attribute `OnItemDataBound="BindRowData"` in the declaration of the `DataGrid` control so that the event handler named `BindRowData` will be executed for each row in the `DataGrid` control as it is bound to the source data. You do this because you need to store the primary key value of the current row someplace where you can retrieve it following a postback.

When you use the `DataGrid` control in its default manner, a postback occurs whenever you put a row into edit mode, and another postback is initiated by the update link. So when you update the data source in response to the update postback, you can access the data in the current row and get the primary key of the row from the `DataKeys` collection. You used this technique in most of the examples in Chapter 4.

However, in this example, you only get a postback after the user has completed all of his or her edits, at which point there is no "current" row. You have to be able to discover the primary key value for each row because you need to handle the changed events and perform the updates for each row.

One of the great things about HTML is that it ignores anything it doesn't understand. Therefore, you can hide the row key within the `style` attribute of every control in each row as a selector that the browser will not recognize (and therefore will ignore). But because it is part of the `style` attribute of the HTML element, it is also part of the `Style` property of the ASP.NET control that implements the element. You can set and read this value in your server-side code and have it persisted in the page, and it will be available following a postback.

An Alternative Approach to Storing Key Values

An alternative approach would be to generate a control-specific key (probably the full control ID, including the prefix generated by the `DataGrid` control—as available from the control's `UniqueID` property) and use that to store the value in the viewstate of the page or in the user's session. However, the technique used in this chapter demonstrates an interesting approach that might be useful in other applications and pages you build.

Listing 10.13 shows the `BindRowData` event handler. You must check that the current row is an `Item` row or an `AlternatingItem` row to prevent runtime errors. Then you can get a reference to each of the edit controls in the current row in turn and add the new style selector to the `Style` property of each one. In this example, you use `"rowval"` as the selector name and the product ID from the current row as the selector value.

LISTING 10.13 Handling the `ItemDataBound` Event to Add the Style Selectors

```
Sub BindRowData(sender As Object, e As DataGridItemEventArgs)

  ' see what type of row (header, footer, item, etc.) caused event
  Dim oType As ListItemType = CType(e.Item.ItemType, ListItemType)

  If oType = ListItemType.Item _
  Or oType = ListItemType.AlternatingItem Then

    ' add primary key of current row to each control in row
    ' required to perform multiple-row updates after a postback
    ' hide it in the style attribute as "rowval:key"
    Dim oTextbox As Textbox _
      = CType(e.Item.FindControl("txtProductName"), Textbox)
    oTextbox.Style("rowval") = e.Item.DataItem("ProductID")

    oTextbox = CType(e.Item.FindControl("txtPrice"), Textbox)
    oTextbox.Style("rowval") = e.Item.DataItem("ProductID")
```

LISTING 10.13 Continued

```
    Dim oCheckbox As Checkbox _
      = CType(e.Item.FindControl("chkDiscontinued"), Checkbox)
    oCheckbox.Style("rowval") = e.Item.DataItem("ProductID")

  End If

End Sub
```

Handling the Changed Events

When the user changes the value in any of the edit controls within the `DataGrid` control and clicks the Update button to submit the page, a changed event occurs on the server for every control where the value has changed since the page was created. A separate event handler is attached to each of the three controls, which appear in every row in the `DataGrid` control. The three event handlers are shown in Listing 10.14.

LISTING 10.14 The Event Handlers for the Changed Events

```
Sub ProductNameChanged(sender As Object, e As EventArgs)

  ' remove any previous formatting
  sender.Style.Remove("color")

  ' execute update against data store
  ' DoItemUpdate returns false if update is not performed
  If Not DoItemUpdate(sender.Style("rowval"), "ProductName", _
         "'" & sender.Text.Replace("'", "''") & "'") Then

    ' change text to red
    sender.Style.Add("color", "Red")
    WriteClientScript(sender.UniqueID)

  End If

End Sub


Sub PriceChanged(sender As Object, e As EventArgs)

  sender.Style.Remove("color")
  If Not DoItemUpdate(sender.Style("rowval"), "UnitPrice", _
         sender.Text.Replace("'", "''")) Then
    sender.Style.Add("color", "Red")
    WriteClientScript(sender.UniqueID)
```

LISTING 10.14 Continued

```
  End If

End Sub


Sub DiscontinuedChanged(sender As Object, e As EventArgs)

  ' remove any previous formatting
  sender.Style.Remove("background-color")

  ' get value of Checkbox as an integer (not True/False)
  Dim iChecked As Integer = (sender.Checked = True)

  ' for demonstration only, cause an update error if
  ' ProductID is 6 by applying invalid value
  Dim sForceError As String = ""
  If sender.Style("rowval") = "6" Then
    sForceError = "*NaN*"
  End If

  If Not DoItemUpdate(sender.Style("rowval"), "Discontinued", _
         iChecked.ToString() & sForceError) Then

    ' change background to red
    sender.Style.Add("background-color", "Red")
    WriteClientScript(sender.UniqueID)

  End If

End Sub
```

The `ProductNameChanged` event handler runs for each `TextBox` control in the Product column of the `DataGrid` control (which displays the product name) where the value has been changed. It first removes any `color` style selector from this element. (It will still contain `background-color:Red` if there was an error last time.) Next, the code calls a separate routine named `DoItemUpdate`, passing it three parameters: the primary key value for this row, the name of the column in the source table, and the new value for that column in this row. You can see that the primary key is extracted from the `Style` property of the text box where it was stored in the `BindRowData` routine when the page was created. Any single quotes in the new value for the column are replaced with two single quotes, and then the value is wrapped in single quotes. This is required to conform to the syntax of a SQL statement. The resulting call to the `DoItemUpdate` routine could look something like this:

```
DoItemUpdate(3, "ProductName", "'Grandma''s Boysenberry Spread'")
```

The DoItemUpdate routine returns True if it can perform the update or False if there is an error of any kind. If it returns False, you change the foreground color of the text box to red and call the WriteClientScript routine. You pass the value of the UniqueID property of the control that raised the event (the text box) to this routine, which (as you'll see later in this chapter) is responsible for highlighting the control and making the Cancel button visible.

The other two event handlers shown in Listing 10.14 work in much the same way as the ones just described. In fact, the PriceChanged event handler for the TextBox controls in the Price column is identical except for the second parameter to the DoItemUpdate method (the column name).

The DiscontinuedChanged event handler is slightly different from the other event handlers in that you set the background color to red to indicate an error because setting the foreground color for a CheckBox control has no visible effect. You also have to convert the Boolean value of a CheckBox control to an Integer value to match the column type in the database.

The other issue here is that it's not easy to demonstrate the result of an update error where a check box is used because the only possible values are True and False, and they are both valid in the database. Therefore, the sample code adds an artificial constraint: Changing the Discontinued value for the row with the product ID of 6 will be classified as an error. To achieve this, you add the string "*NaN*" to the value of the control when you call the DoItemUpdate method, which will subsequently fail to apply the update. If you try to update the Discontinued columns for this row, you'll see an error reported, the background of the check box turn red, and the focus move to the check box (see Figure 10.19).

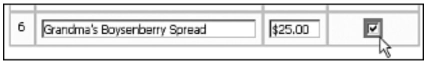


FIGURE 10.19 Highlighting the check box for the product with an ID of 6.

Updating the Source Data

You've just seen how the three event handlers in Listing 10.14 call a routine named DoItemUpdate to push the changed values back into the database. Listing 10.15 shows this routine in full. There is nothing really unusual here; you just generate a suitable SQL statement and then execute it against the database. You return True if one row is updated or False otherwise.

LISTING 10.15 A Routine That Performs Updates to the Data Source

```
Function DoItemUpdate(sRowKey As String, _
                      sColumnName As String, _
                      sValue As String) As Boolean

  ' create a suitable SQL statement and execute it
  Dim sSQL As String
  sSQL = "UPDATE Products SET " & sColumnName & "=" _
       & sValue & " WHERE ProductID=" & sRowKey
  lblErr.Text &= "Executing SQL statement: " & sSQL & "<br />"
```

LISTING 10.15 Continued

```
  Dim sConnect As String = ConfigurationSettings.AppSettings( _
                           "NorthwindOleDbConnectString")
  Dim oConnect As New OleDbConnection(sConnect)

  Try

    oConnect.Open()
    Dim oCommand As New OleDbCommand(sSQL, oConnect)
    If oCommand.ExecuteNonQuery() <> 1 Then
      lblErr.Text &= "ERROR: Could not update the selected row"
      Return False
    Else
      Return True
    End If
    oConnect.Close()

  Catch oErr As Exception

    ' be sure to close connection if error occurs
    If oConnect.State <> ConnectionState.Closed  Then
      oConnect.Close()
    End If

    ' display error message in page
    lblErr.Text &= "ERROR: " & oErr.Message & "<br />"
    Return False

  End Try

End Function
```

One thing you aren't doing here is managing concurrency errors (in which more than one person tries to update the database at the same time). You can protect against these errors by saving the existing values of the source data rows in the page and then checking that they haven't changed in the database when you perform the update.

One way to do this would be to store the current values for columns that the user can edit in hidden controls in each row of the `DataGrid` control. Then you could extract

> **Remember to Use Parameters in Your SQL Statement**
>
> **Notice that in Listing 10.15 we have broken the rule we suggested earlier in this chapter about protecting your pages from malicious users: We have built our SQL statements as literal text rather than by including parameters. We've done that here only so that you can see the actual SQL statement that is executed. If we used replaceable parameters in the SQL statement, you wouldn't see the values and the full syntax. You would only see the parameter names within the statements.**

Using SQL Statements That Update More Than One Column

As shown in the code in Listing 10.15, if the user edits more than one control value in the same row, you actually end up executing a separate `UPDATE` action on the database for each edited control. Following a postback, the changed events are raised on the server in the order in which the controls are declared on the page. This means they will occur in turn for all the controls in each row that have been changed. You could, therefore, use the control names or their `UniqueID` values to determine whether they came from the same row, and if so build compound `UPDATE` statements, but the code required for that is certainly not trivial.

them after a postback and use them in the `WHERE` clause of the SQL statement so that it becomes the following:

```
UPDATE Products SET ProductName='New Chai'
WHERE ProductID=1 AND ProductName='Chai'
      AND UnitPrice=18.00 AND Discontinued=0
```

Creating the Client-Side Script to Highlight a Control

The final part of the code in this example demonstrates how you can interact with the controls on the page by using client-side script. Chapters 5, "Creating Reusable Content," and 6, "Client-Side Script Integration," show plenty of this kind of code. All you do is create some client-side JavaScript that runs as the page loads (instead of being located within an event handler), and then you insert it at the end of the `<form>` section of the page, using the `RegisterStartupScript` method of the `Page` object. (This method is described and used at the end of Chapter 7, "Design Issues for User Controls.") The code required for this simply has to move the input focus to the required control on the page and select any text it might contain:

```
var ctrl = document.getElementById('control-id');
ctrl.focus();
ctrl.select();
```

When the `WriteClientScript` routine (shown in Listing 10.16) is called, the full ID of the control that caused the error (its `UniqueID` property) is passed as a parameter. This is used in the client-side script to get a reference to the appropriate element. And because you test whether the script block has been inserted already, this block will be added to the page only once. This means that the focus will move to the first element within the `DataGrid` control that contains an error—just what you want!

LISTING 10.16 Creating a Client-Side Script Block to Highlight Input Errors

```
Sub WriteClientScript(sCtrlID As String)

  ' create script to set focus to first control with error
  ' see if previous instance of this control has already
  ' added the required JavaScript code to the page
  If Not Page.IsClientScriptBlockRegistered("AHHGridFocus") Then

    Dim sScript As String = vbCrlf _
    & "<script language='javascript'>" & vbCrlf _
```

LISTING 10.16 Continued

```
    & "<!--" & VbCrlf _
    & "var ctrl = document.getElementById('" & sCtrlID & "');" _
    & vbCrlf _
    & "ctrl.focus();" & vbCrlf _
    & "ctrl.select();" & vbCrlf _
    & "// -->" & vbCrlf _
    & "<" & "/script>" & vbCrlf

    ' add this JavaScript code to the page
    Page.RegisterStartupScript("AHHGridFocus", sScript)

  End If

  ' make Cancel button visible. Will disappear again on
  ' next postback because EnableViewState in False
  btnRefresh.Visible = True

End Sub
```

As you can see, you now have a responsive and intuitive page that allows multiple row updates to be performed without requiring intermediate postbacks. Although it might not suit your requirements in every way, you could easily adapt the techniques for use in other situations.

Summary

This chapter is concerned with issues that can arise when you have to manage and interact with data using ADO.NET. It also looks at a nonstandard approach to using the ASP.NET list controls. (Some useful techniques with XML data are described in Chapter 11, "Working with XML Data.")

This chapter starts by looking at the risks you face when accepting input from users and constructing SQL statements from it. You have seen the problem demonstrated in a simple sample page and know that, to protect your pages and data, you should consider using parameters in all cases.

This chapter then looks at how you can use stored procedures that contain default parameter values and how this approach can make it easier to write data access code. The example in this chapter shows how you can create sensible default values for data rows and how you can report errors via the Windows event log. This chapter also investigates the passing of parameters by name for the `SqlClient` data provider (whereas other providers pass parameters by position and ignore parameter names).

This chapter also examines how the structure of tables, columns, and other metadata can be generated from a schema or by simply filling the `DataSet` instance from a database. You have seen how the schema affects the result and how the values for the `MissingSchemaAction` property of the `DataAdapter` instance affect the outcome. This chapter also looks at a performance comparison involving loading a schema into a `DataSet` instance to create the internal structure first, before loading the data.

Finally, this chapter shows an alternative approach to editing data in an ASP.NET list control—in this case a `DataGrid` control. By handling the changed events for the controls within the `DataGrid` control, you can allow the user to perform multiple changes and submit them all to the server for updating in one go, rather than using the default individual postback approach.

11

Working with XML Data

IN THIS CHAPTER

In today's distributed world, data comes in a wide variety of shapes and sizes. As a result, exchanging data between different entities is often a challenging task. Although several different data exchange formats have been created—such as flat files, binary structures, and electronic data interchange (EDI)—few have proven to be as versatile and easy to use as Extensible Markup Language (XML). XML has the advantage of being readable by both humans and computers, plus it has support for validation, parsing, and transformation. These strengths have made it one of the most popular technologies for exchanging, storing, and manipulating data.

This chapter discusses several different ways that XML data can be integrated into ASP.NET applications and demonstrates how native .NET Framework XML-parsing application programming interfaces (APIs) can be used to read, write, and manipulate both relational and XML data. The chapter starts off with a quick refresher on how XML can be used in ASP.NET applications and then moves into a discussion of the pros and cons of different .NET Framework XML APIs. The chapter then covers topics such as transforming data with the `XmlTextReader` and `XmlTextWriter` classes, searching XML documents, working with default and local namespaces, converting relational data to XML, leveraging XML serialization, and much more.

The Role of XML in ASP.NET

One of the most popular ways XML data has been used in Web applications over the past few years has been to transform it into HTML by using Extensible Stylesheet Language Transformations (XSLT). People with a variety of programming skills can use XML and XSLT to quickly and easily build dynamic applications capable of targeting multiple devices. You can easily skin and personalize Web sites by changing the XSLT stylesheet used to transform XML data and generate the HTML output.

With the release of the .NET Framework in 2002, a new way of developing Web applications was introduced that allowed developers to build object-oriented pages capable of utilizing specialized HTML generators known as *server controls*. As a result of this substantial shift in Microsoft's Web technology, one could argue that the need for XML and XSLT has been minimized. After all, ASP.NET allows data to be bound to many different types of controls with a minimal amount of code.

Although it's true that ASP.NET makes the display of data much more straightforward than in the past, there are still many ways that XML (and other XML technologies, such as XSLT) can be used productively in ASP.NET applications. Examples include grabbing XML/RSS news feeds from remote sites, blending relational and XML data together for reporting purposes, generating read-only output, exchanging data between disparate applications, aggregating data from XML Web services, and transforming relational data into hierarchical structures (to name only a few). Because the .NET Framework has built-in support for validation, parsing, and transforming XML, as well as mapping relational data to XML, the possibilities of leveraging XML in ASP.NET applications are virtually endless.

The bottom line is that XML provides a platform-neutral way to work with data that doesn't require a specific database, programming language, or operating system. As more and more products support XML and as new XML technologies continue to be released (such as the upcoming XQuery language), it becomes easier and easier for different types of data stored in distributed locations to be integrated into ASP.NET Web applications.

XML API Pros and Cons

Understanding the different ways that XML can be used in ASP.NET applications is important, but when you're ready to design and build an application, it's crucial that you know the pros and cons associated with the .NET Framework's XML-parsing APIs up front. Not knowing the pros and cons can lead to the creation of inefficient and nonscalable applications because some APIs are better suited for specific activities than others. Table 11.1 shows the classes that provide functionality for the four main XML-parsing APIs found in the .NET Framework.

TABLE 11.1

The .NET Framework's XML-Parsing API Classes

Class	API Functionality
`XmlTextReader`	A forward-only, noncached XML reader that is capable of reading large XML documents quickly and efficiently.
`XmlDocument`	An editable in-memory object model, referred to as the Document Object Model (DOM), that allows for node selection using the XPath language.
`XPathNavigator`	A non-editable, in-memory, random movement, cursor-style model. Like the DOM, this API allows nodes to be selected by using XPath.
`XmlSerializer`	A serialization/deserialization API that converts objects to XML and back.

Some of the XML APIs listed in Table 11.1 provide flexible object models that are more memory intensive; others aren't as flexible but provide the ultimate in speed and performance. Although you likely have experience working with one or more of the XML APIs available in the .NET Framework, it's worthwhile to examine the pros and cons, starting with the forward-only API exposed by the `XmlTextReader` class.

The Forward-Only API: `XmlTextReader`

The pros of forward-only API include the following:

- The `XmlTextReader` class provides a forward-only, noncached reader that is memory efficient and works well when XML data needs to be read quickly and efficiently.
- XML tokens in the stream created by the `XmlTextReader` class can be pulled out and analyzed as desired. Unwanted tokens can be skipped.
- The `XmlTextReader` class is the fastest and most efficient API in the .NET Framework for parsing XML documents.

The cons of forward-only API include the following:

- The `XmlTextReader` class does not contain any editing functionality.
- The XML parsing process is forward only.
- The `XmlTextReader` class can arguably be more difficult to use than other XML-parsing APIs, such as the DOM API.

The DOM API: `XmlDocument`

The pros of the DOM API include the following:

- When you use the DOM API, nodes (elements, attributes, text nodes, and so on) can be updated, deleted, inserted, and moved.
- XPath expressions can be used to query an XML document and locate specific nodes. Unwanted nodes can be ignored.

- Recursion techniques can be used to walk through the DOM structure, which can result in less code.
- The DOM structure can be traversed in any direction, from parent to ancestor to siblings.

The cons of the DOM API include the following:

- The DOM API works by loading the entire XML document into memory. This may cause performance or scalability problems when working with large XML documents.
- The DOM API uses the forward-only API exposed by the `XmlTextReader` class behind the scenes to initially read the XML document and load it into memory. Although this is not a con in and of itself, this extra loading step causes the performance of the DOM API to be slower than the parsing speed of the `XmlTextReader` class.

The Cursor-Style API: `XPathNavigator`

The pros of the cursor-style API include the following:

- XML can be navigated randomly, using a more efficient mechanism than that associated with DOM objects in the .NET Framework.
- XPath expressions can be used to query an XML document and access specific nodes. XPath expressions can be compiled in order to add additional functionality, such as sorting.
- Recursion techniques can be used to walk through the XML tree in a cursor-style manner, which can result in less code.
- XML data can be traversed in any direction, from parent to ancestor to siblings.
- Non-XML data stores that expose the `IXPathNavigable` interface—including INI files, directory structures, and more—can be navigated by using `XPathNavigator`.

The cons of the cursor-style API include the following:

- The `XPathNavigator` class does not allow nodes to be edited (this will change in version 2 of the .NET Framework, which adds new editing classes, such as `XPathEditor`).
- Although more efficient than the DOM API, `XPathNavigator` still involves working with XML documents that have been loaded into memory and is therefore not as fast or efficient as the forward-only, noncached API found in the `XmlTextReader` class.

The XML Serialization API: `XmlSerializer`

The pros of the XML serialization API include the following:

- XML documents can be manipulated without an in-depth knowledge of different XML-parsing APIs.

- XML documents can be created and edited by using real-world objects rather than DOM-specific classes.
- An object's state can be stored (serialized) and restored (deserialized) by using XML.
- XML schema definition (XSD) schemas can be converted to .NET Framework classes by using the `xsd.exe` tool.

The cons of the XML serialization API include the following:

- When done by hand, XML serialization/deserialization may involve more up-front development work than using other APIs, such as the one exposed by the `XPathNavigator` class.
- Although it is very convenient, XML serialization is not as efficient as the forward-only, noncached API exposed by the `XmlTextReader` class.
- The `XmlSerializer` class serializes only public properties/fields of an object. Private members are ignored.

Combining the XmlTextReader and XmlTextWriter Classes

There are a variety of ways that you can convert XML data into HTML by using the .NET Framework and ASP.NET. Although using XSLT is a popular way to transform XML into HTML, XSLT is memory intensive and doesn't always perform well when large amounts of data are involved. When maximum performance is needed, you should consider combining functionality found in the `XmlTextReader` and `XmlTextWriter` classes instead of using XSLT. If you use these two classes, you'll see better performance, especially when data changes frequently and can't be cached.

To illustrate one of the many ways you can use the `XmlTextReader` class along with the `XmlTextWriter` class to generate HTML, let's examine a simple XML document published by MoreOver.com, found at `http://p.moreover.com/cgi-local/page?c=XML%20and%20metadata%20news&o=xml`. (As with any hyperlink on the Web, the MoreOver.com link listed here is subject to change.) Listing 11.1 shows a portion of the XML document that contains news articles. Each news article is marked up by using an `article` element that has several children. Two of the children (`url` and `headline_text`) are used in the examples that follow to dynamically add news headlines to an ASP.NET page.

> **Tips for Enhancing XSLT Performance**
>
> **Although the `XmlTextReader` and `XmlTextWriter` classes are being highlighted in this section, XSLT is still a viable solution for transforming XML into different output structures. However, if you use XSLT in the .NET Framework version 1.1, it is highly recommended that you do not use the `XmlDataDocument` class when performing a transformation due to performance issues. Instead, you should use the `XPathDocument` class located in the `System.Xml.XPath` namespace.**

LISTING 11.1 MoreOver.com XML News Feed

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE moreovernews
  SYSTEM "http://p.moreover.com/xml_dtds/moreovernews.dtd">
<moreovernews>
  <article id="_116273395">
    <url>http://c.moreover.com/click/here.pl?x116273395</url>
    <headline_text>oXygen XML Editor 3.0 released</headline_text>
    <source>MacMinute</source>
    <media_type>text</media_type>
    <cluster>XML and metadata news</cluster>
    <tagline></tagline>
    <document_url>http://www.macminute.com</document_url>
    <harvest_time>Jan 14 2004 8:57PM</harvest_time>
    <access_registration></access_registration>
    <access_status></access_status>
  </article>
  <!-- More article nodes follow -->
</moreovernews>
```

Because the `XmlTextReader` class creates a stream of XML tokens, the MoreOver.com news feed can be parsed quickly and efficiently, without tying up a lot of memory. Listing 11.2 shows how to parse data from the XML document by using the `XmlTextReader` class and generate HTML output by using the `XmlTextWriter` class.

LISTING 11.2 Parsing XML with `XmlTextReader` and Creating HTML with `XmlTextWriter`

```
Dim url As String = _
  "http://p.moreover.com/cgi-local/page?" + _
  "c=XML%20and%20metadata%20news&o=xml"
'Create backing store where XmlWriter will write to
Dim sw As New StringWriter
Dim writer As New XmlTextWriter(sw)
writer.Formatting = Formatting.Indented
Dim reader As XmlTextReader = Nothing
Try
  writer.WriteStartElement("ul")
  'Parse XML using the XmlReader stream API
  reader = New XmlTextReader(url)
  reader.XmlResolver = Nothing 'Prevent DTD resolution
  Dim headline_url As String = Nothing
  While reader.Read()
    'Locate only start elements
    If reader.NodeType = XmlNodeType.Element Then
      Select Case reader.Name
```

LISTING 11.2 Continued

```
        Case "url"
          headline_url = _
            reader.ReadString() 'Store URL
        Case "headline_text"
          Dim headline As String = reader.ReadString()
          'Filter out headlines that don't have xml
          'keyword in them
          If headline.ToLower().IndexOf("xml") <> -1 Then
            writer.WriteStartElement("li")
            writer.WriteStartElement("a")
            writer.WriteAttributeString("href", headline_url)
            writer.WriteAttributeString("target", "_blank")
            writer.WriteString(headline)
            writer.WriteEndElement() '</a>
            writer.WriteEndElement() '</li>
          End If
       End Select
    End If
  End While
  writer.WriteEndElement() '</ul> tag
  'Update Label
  Me.lblNews.Text = sw.ToString()
Catch exp As Exception
  Me.lblNews.Text = exp.Message
Finally
  reader.Close()
  writer.Close()
End Try
```

The code in Listing 11.2 starts out by creating a new `StringWriter` instance (which internally creates a `StringBuilder` instance) that is passed into the `XmlTextWriter` class's constructor. Because the output will ultimately be written to an ASP.NET `Label` control, the `StringWriter` instance provides a convenient way to capture the data generated by the writer. Next, the `XmlTextReader` class is instantiated and used to parse the remote XML document. As the stream of XML tokens begins flowing, node types are checked, and element names are inspected, using the `XmlTextReader` class's `NodeType` and `Name` properties, respectively. When the `url` and `headline_text` nodes are found, their text nodes are read, using `ReadString()`, and are passed to the `XmlTextWriter` class's `WriteString()` method for inclusion in the output. Headlines that do not contain the text `xml` are filtered out and do not show up in the output data. Finally, the `StringWriter` class's `ToString()` method is called to access the output data and assign it to the `Label` control.

Although the `XmlTextWriter` class isn't absolutely necessary in this situation, using it prevents messy HTML string concatenations from being added to the code. The `XmlTextWriter` class also has the added benefit of making the output more human readable by setting its `Formatting` property to `Formatting.Indented` and generating well-formed HTML.

Although not shown in Listing 11.2, the `XmlTextWriter` class has a nice convenience method that can be used in situations where attributes encountered by the `XmlTextReader` class need to be automatically added to the output generated by the writer. This is accomplished using the `XmlTextWriter` class's `WriteAttributes()` method, which accepts an `XmlReader` instance as a parameter. `WriteAttributes()` automatically walks through all attributes on the current element and positions the reader back on the element when it finishes writing the last attribute. Although it's easy enough to write this functionality by hand, knowing how methods such as `WriteAttributes()` can be used results in less coding.

To this point, you've seen that the `XmlTextReader` and `XmlTextWriter` classes work well together. However, after looking through the code in Listing 11.2, you might be wondering if it's worth the effort. After all, other .NET Framework classes, such as `DataSet`, can be utilized to parse and bind XML data to ASP.NET Web server controls such as the `DataList` control with a minimal amount of code, as shown in Listing 11.3.

LISTING 11.3 Parsing and Binding XML Data with the `DataSet` Class

```
Dim url As String = _
  "http://p.moreover.com/cgi-local/page?" + _
  "c=XML%20and%20metadata%20news&o=xml"
Dim ds As New DataSet
Try
  ds.ReadXml(url) 'Load XML into DataSet
  Dim dv As DataView = ds.Tables(0).DefaultView
  dv.RowFilter = "headline_text LIKE '%xml%'"
  Me.dlNews.DataSource = dv
  Me.dlNews.DataBind()
Catch exp As Exception
  Me.lblError.Text = exp.Message
End Try
```

Using the `DataSet` instance's `ReadXml()` method, the MoreOver.com XML document can be loaded and parsed with a single line of code. The data can then be filtered and bound to a `DataList` control with only four more lines of code; this is quite impressive. Given the minimal amount of code in Listing 11.3, it may be tempting to always use the `DataSet` class and ignore some of the other XML parsing options in the .NET Framework.

The important factor to take into consideration when deciding between using the `XmlTextReader` and `XmlTextWriter` classes and the `DataSet` class is memory. Although both solutions can filter out undesirable data (such as headlines that don't have the text `xml` in them), the `DataSet` class loads the entire XML document into memory, whereas the `XmlTextReader` class streams the XML data and has a small memory footprint.

With the DataSet class, small to medium-sized XML documents may not present a problem. Large documents will likely cause available memory to decrease, depending on how often the data is loaded. ASP.NET caching can be used to prevent loading the XML document for each request to the server, but caching may not be practical in cases where the XML data changes frequently. Such cases may warrant the more streamlined XmlTextReader and XmlTextWriter approach.

Parsing XML Strings

A question that shows up quite frequently in different .NET Framework XML newsgroups and listservs is "How do I parse XML data within a string?" Many people find it easy to parse XML strings by using the XmlDocument class's LoadXml() method but have a harder time figuring out how to parse XML strings by using the XmlTextReader class. The answer to this question is surprisingly simple, but it involves one extra step in order to work properly. If you look at the .NET Framework Software Developer's Kit (SDK), you'll see that the XmlTextReader class has the following overloaded constructor that accepts a TextReader instance:

```
Public Sub New(TextReader)
```

The StringReader class (located in the System.IO namespace) derives from the abstract TextReader class and reads strings that are passed into its constructor. As a result, it can be passed into the XmlTextReader class's constructor, as shown in Listing 11.4.

LISTING 11.4 Parsing an XML String with the XmlTextReader Class

```
Dim sb As New StringBuilder
Dim xml As String = _
  "<customers><customer id=""2"" " + _
  "name=""John Doe"" /></customers>"
Dim reader As XmlTextReader = New XmlTextReader(New StringReader(xml))
While (reader.Read())
  If (reader.Name = "customer" And _
    reader.NodeType = XmlNodeType.Element) Then
    While (reader.MoveToNextAttribute())
      sb.Append(reader.Name)
      sb.Append(" = ")
      sb.Append(reader.Value)
      sb.Append("<br />")
    End While
  End If
  Me.lblOutput.Text = sb.ToString()
End While
reader.Close()
```

When the XML within the string is loaded into the `StringReader` instance (which is in turn passed to the `XmlTextReader` constructor), it can be parsed like any other XML data. Because the `XmlTextReader` instance is closed, the `StringReader` will automatically be closed as well.

Accessing XML Resources by Using the `XmlResolver` Class

The `System.Xml` namespace contains an abstract class named `XmlResolver` that is responsible for resolving external resources, including items such as document type definitions (DTDs) and schemas. Although a concrete instance of the `XmlResolver` class can't be created, `XmlResolver` has two child classes that derive from it—`XmlUrlResolver` and `XmlSecureResolver`—that can be instantiated and used. These classes are used under the covers by different .NET Framework classes, such as `XmlDocument`, `XmlDataDocument`, and `XmlTextReader`, and they can be accessed through their respective `XmlResolver` properties.

`XmlUrlResolver` is typically used when an external resource such as a DTD needs to be ignored, when security credentials need to be passed to a remote resource, or with XSLT stylesheets. Ignoring external resources is accomplished by setting the `XmlResolver` property of XML classes such as `XmlTextReader` and `XmlDocument` to `Nothing` (`null` in C#). This can be useful when the XML data needs to be parsed but a referenced DTD or schema doesn't need to be resolved.

An example of setting the `XmlResolver` property to `Nothing` is shown in Listing 11.2, where the MoreOver.com news feed is parsed to extract news headlines. Because the DTD referenced in the document isn't of use to the application, the `XmlTextReader` class's `XmlResolver` property is set to `Nothing`. If the `XmlResolver` property were left in its default state, the DTD uniform resource identifier (URI) would be resolved by an underlying `XmlResolver` property, assuming that access to the Internet is available. However, if a proxy server blocked outside access to the DTD or if the network were temporarily unavailable, the following error would occur:

```
The underlying connection was closed:
The remote name could not be resolved.
```

You can also use the `XmlUrlResolver` class to pass security credentials to a remote resource that requires authentication by using its `Credentials` property. `Credentials` represents a write-only property of type `ICredentials`. Listing 11.5 shows how you can create an `XmlUrlResolver` instance and assign it authentication credentials by using the `NetworkCredential` class (found in the `System.Net` namespace). After you define the necessary credentials, you assign the `XmlUrlResolver` instance to the `XmlTextReader` class's `XmlResolver` property so that the secured XML document can be parsed.

LISTING 11.5 Passing Security Credentials to a Remotely Secured XML Document

```
Dim reader As XmlTextReader = Nothing
Dim xmlUri As String = "http://localhost/XMLChapterVB/Listing1.xml"
'Get login credentials
Dim uid As String = ConfigurationSettings.AppSettings("UID")
```

LISTING 11.5 Continued

```
Dim pwd As String = ConfigurationSettings.AppSettings("Password")
Dim domain As String = ConfigurationSettings.AppSettings("Domain")
Dim resolver As New XmlUrlResolver
resolver.Credentials = New NetworkCredential(uid, pwd, domain)
Try
  reader = New XmlTextReader(xmlUri)
  'Hook resolver to XmlTextReader
  reader.XmlResolver = resolver
  While reader.Read() 'Try to parse document
  End While
  Me.lblOutput.Text = "Parsed secured document."
Catch exp As Exception
  Me.lblOutput.Text = "Did NOT parse secured document: " + exp.Message
Finally
  If Not (reader Is Nothing) Then
    reader.Close()
  End If
End Try
```

`XmlResolver`, `Evidence`, and `XslTransform`

Version 1.1 of the .NET Framework enhances security in the `XslTransform` class by marking several overloaded versions of the `XslTransform` class's `Load()` and `Transform()` methods as obsolete while adding new, more secure overloaded methods. The `XslTransform` class's `XmlResolver` property has also been made obsolete in version 1.1. These changes prohibit XSLT scripts and extension objects, `xsl:import` and `xsl:include` statements, and XSLT `document()` functions from being processed without supplying security evidence and/or an `XmlResolver` instance when calling the `Load()` and `Transform()` methods. The following sections analyze changes to the `XslTransform` class and explain the roles of the `XmlResolver` and `Evidence` classes.

The `Load()` Method

The `XslTransform` class's `Load()` method found in version 1.1 of the .NET Framework has several overloads that allow fine-grained control over whether XSLT scripts, extension objects, and `xsl:import`/`xsl:include` statements are ignored during an XSLT transformation. When local XSLT stylesheets are used in a transformation, the `Load()` method still has an overload that accepts a `String`-type parameter containing the path to the stylesheet. Using this overload is the easiest way to transform XML documents via XSLT because it automatically handles included or imported stylesheets as well as compiling embedded script within the stylesheet.

> **Loading Local XSLT Stylesheets**
>
> **Using the overloaded `Load()` method and passing the physical location of the XSLT stylesheet is fairly straightforward:**
>
> ```
> Dim xslPath as String = _
> Server.MapPath("XSLT/Output.xslt")
> Dim trans as New XslTransform
> trans.Load(xslPath)
> 'Perform transformation
> 'with Transform() method
> ```

In addition to the overload mentioned previously, there are several new overloads for the `Load()` method in version 1.1 of the .NET Framework. The following is one example:

```
Overloads Public Sub Load( _
   ByVal stylesheet As IXPathNavigable, _
   ByVal resolver As XmlResolver, _
   ByVal evidence As Evidence _
)
```

The `IXPathNavigable` parameter represents the XSLT stylesheet used in the transformation. This parameter accepts any object that implements the `IXPathNavigable` interface, such as `XmlDocument`, `XPathNavigator`, or `XPathDocument`.

The `XmlResolver` parameter is used to resolve XSLT documents imported into or included in a master stylesheet. When the parameter value is `Nothing`, imported or included documents are ignored. When a new `XmlUrlResolver` instance is passed into the `Load()` method, documents referenced by `xsl:import` or `xsl:include` statements are resolved and used in the transformation. The `XmlUrlResolver` class's `Credentials` property can be used in cases where included or imported stylesheets require authentication in order to be accessed. (Refer to Listing 11.5 for an example of using the `Credentials` property.)

The `evidence` parameter determines whether XSLT script blocks and extension objects are processed based on whether they are from trusted sources. The parameter is based on the `Evidence` type, located in the `System.Security.Policy` namespace. The .NET Framework SDK provides the following insight into how `Evidence` is used:

> Security policy is composed of code groups; a particular assembly (the basic unit of code for granting security permissions) is a member of a code group if it satisfies the code group's membership condition. Evidence is the set of inputs to policy that membership conditions use to determine to which code groups an assembly belongs.

That is, by supplying a proper `Evidence` object to the `Load()` method, script code contained within potentially untrusted XSLT stylesheets can be compiled and used in an XML document transformation because the assembly that is generated can be assigned to the proper .NET Framework code group. If no `Evidence` type is supplied, assemblies created during the compilation of XSLT script code cannot be used successfully due to their inherit security risks. For example, when `Nothing` is passed for the `Evidence` parameter, XSLT scripting, extension objects, and `document()` functions are ignored.

In cases where a *local* XSLT stylesheet has embedded script, uses extension objects, or references the `document()` function, the following code can be used to create the proper `Evidence` object for the assembly:

```
Me.GetType().Assembly.Evidence
```

When a remote XSLT stylesheet containing script or extension object references is used in a transformation, the caller of the `Load()` method must supply evidence in order for script or extension objects to be executed properly. To supply evidence, the XmlSecureResolver class's CreateEvidenceForUrl() method can be used. The CreateEvidenceForUrl() method accepts a single String-type parameter that contains the URL for which to create evidence, as shown here:

```
Dim uri as String = "some uri"
Dim xslDoc as new XPathDocument(uri)
'Create Evidence
Dim e as Evidence = _
  XmlSecureResolver.CreateEvidenceForUrl(uri)
Dim trans as new XslTransform
trans.Load(xslDoc,new XmlUrlResolver(),e)
'XSLT script, extension objects, etc. can be used
'since evidence was supplied
```

The Transform() Method

In addition to new overloaded Load() methods, the Transform() method has several new overloads that expect an instance of an XmlResolver to be passed as a parameter. The following is an example of one of these overloads:

```
public void Transform(XPathNavigator, XsltArgumentList, _
  TextWriter, XmlResolver);
```

In cases where simple XSLT transformations (that is, transformations that involve a single XML document and a single XSLT stylesheet stored locally) are performed, Nothing (null in C#) can be passed for the XmlResolver parameter:

```
Dim writer as new StringWriter
Dim xsl As New XslTransform
xsl.Load(xslPath)
xsl.Transform(doc, Nothing, writer, Nothing)
```

Passing Nothing for the XmlResolver parameter when more than one XML document is involved in the transformation presents a problem. For example, when the document() function is used within the XSLT stylesheet to transform multiple XML documents simultaneously, passing Nothing causes any additional XML documents to be ignored. In order to perform this type of transformation successfully, a new XmlUrlResolver instance must be passed to the Transform() method. Listing 11.6 shows how this is done and highlights how evidence can be passed to the Load() method in cases where a local XSLT stylesheet is used.

LISTING 11.6 Using the XslTransform Class's Load() and Transform() Methods

```
Dim sw As New StringWriter
'Load XML Doc and master XSLT Stylesheet
Dim xmlDoc As New XPathDocument(Server.MapPath("XML/Form.xml"))
Dim xslDoc As New XPathDocument(Server.MapPath("XSLT/Form.xslt"))
```

LISTING 11.6 Continued

```
'Create XslTransform and load stylesheet
Dim trans1 As New XslTransform
Dim resolver As New XmlUrlResolver
trans1.Load(xslDoc, resolver, Me.GetType().Assembly.Evidence)
'Transform XML
trans1.Transform(xmlDoc, Nothing, sw, resolver)
Response.Write(sw.ToString())
sw.Close()
```

Searching, Filtering, and Sorting XML Data

A little over a year after the XML 1.0 specification was released by the World Wide Web Consortium (W3C), the XPath language emerged on the scene to fill a void created by the inability to effectively search and filter XML data. Since its release, XPath has become increasingly important in the world of XML and is used in DOM APIs, XSLT stylesheets, XSD schemas, and other XML-specific languages.

XPath is a path-based language (it resembles DOS paths in some regards) that allows specialized statements capable of searching and filtering nodes to be executed against XML documents. This chapter does not provide a complete explanation of the XPath language; for more details on the XPath language, see the book *XML for ASP.NET Developers* from Sams Publishing. The following is a sample XPath statement that uses axes, node tests, and a predicate:

```
/customers/customer[@id='ALFKI']
```

This XPath statement uses the `Child` and `Attribute` axes, along with node tests and a predicate (the text within the square brackets) to search for an element named `customer` that has an `id` attribute value equal to `ALFKI`. When the statement is executed, unwanted nodes are automatically filtered out, and the desired node is returned (assuming that it is found). Although quite simple, this XPath statement demonstrates the power of searching and filtering data located in an XML document. The following sections show how different .NET Framework classes can be used along with XPath to search, filter, and sort data.

> **Using ADO.NET to Search, Filter, and Sort XML Data**
>
> **You can also search, filter, and sort XML data by using the `DataSet` class and its related classes. After loading XML data into a `DataSet` instance by using the `ReadXml()` method, you can use properties and methods of the `DataTable` and `DataView` classes to accomplish tasks similar to those that the XPath language handles.**

Searching and Filtering XML Data

The .NET Framework contains several classes that are capable of searching and filtering XML data using the XPath language. Each class has unique pros and cons, as outlined earlier in this chapter, and offers different levels of efficiency. The `XPathNavigator` class is designed to work hand-in-hand with the XPath language to provide a read-only cursor-style model for navigating XML nodes. Other classes, such as `XmlDocument` and `XmlNode`, provide XPath support through their `SelectNodes()` and `SelectSingleNode()` methods.

When designing applications that consume XML data, you should first look to the `XPathNavigator` class (located in the `System.Xml.XPath` namespace) when you need to search XML data. Although `XPathNavigator` isn't as fast as the forward-only API provided by the `XmlTextReader` class and doesn't provide the editing capabilities found in the DOM API (this will change in version 2.0 of the .NET Framework when classes such as `XPathEditor` are introduced), it can be useful in applications that need the ability to traverse an XML document's hierarchy along a variety of axes. The `XPathNavigator` class offers numerous benefits, such as compiled XPath statements and the ability to leverage the `IXPathNavigable` interface to search non-XML data stores.

The `XPathNavigator` class is abstract, so it can't be created directly. However, you can use classes that implement the `IXPathNavigable` interface (`XmlDocument`, `XmlDataDocument`, `XmlNode`, and `XPathDocument`) to create a concrete instance of the `XPathNavigator` class by using `CreateNavigator()`. After the `XPathNavigator` instance is created, you can navigate through the XML document one node at a time. When you are positioned on a node, you can reach other nodes located before or after the current node by calling a variety of methods, such as `MoveToNext()`, `MoveToParent()`, and `MoveToFirstChild()`.

You can also use `XPathNavigator` to search and filter nodes within an XML document by using XPath statements. By leveraging XPath, you can greatly reduce the amount of code that needs to be written to gather data, thus making applications easier to maintain. Nodes returned from executing an XPath statement are placed in an `XPathNodeIterator` instance that can be iterated through easily. Before looking at an example of using `XPathNavigator`'s methods, you should examine the XML document in Listing 11.7, which contains book and author data.

LISTING 11.7 An XML Document That Contains Book and Author Data

```
<?xml version="1.0"?>
<bookstore>
  <book genre="novel" style="hardcover">
    <title>The Handmaid's Tale</title>
    <author>
      <first-name>Margaret</first-name>
      <last-name>Atwood</last-name>
    </author>
    <price>19.95</price>
  </book>
  <book genre="novel" style="hardcover">
    <title>The Worker's Tale</title>
    <author>
      <first-name>Margaret</first-name>
      <last-name>Atwood</last-name>
    </author>
```

LISTING 11.7 Continued

```
    <price>49.95</price>
  </book>
  <!-- Additional book nodes removed for brevity -->
</bookstore>
```

Listing 11.8 demonstrates how to walk through the XML data shown in Listing 11.7 and write out book and author details. The code in Listing 11.8 uses different methods to navigate from node to node, such as `MoveToFirstChild()`, `MoveToNext()`, and `SelectChildren()`. The code also searches for other books that a specific author has written by passing an XPath statement to the `Select()` method. Several comments have been added to the code in Listing 11.8 to provide additional details about what it is doing. Figure 11.1 shows the output generated by executing the code.

LISTING 11.8 Navigating XML Data by Using `XPathNavigator`

```
Dim sb as New StringBuilder
Private Sub NavigateBooks()
  Dim xmlPath As String = Server.MapPath("Listing7.xml")
  'Load XML into a non-editable structure
  'This is more efficient than the DOM
  Dim doc As New XPathDocument(xmlPath)

  'Create XPathNavigator by calling CreateNavigator() method
  Dim nav As XPathNavigator = doc.CreateNavigator()
  'Move to document
  nav.MoveToRoot()
  'Move to root element - bookstore
  nav.MoveToFirstChild()

  'Move to first book child element
  If nav.MoveToFirstChild() Then
    Do 'Walk through book elements
      WalkSiblings(nav)
    Loop While nav.MoveToNext()
  End If
  'Write out data found while navigating doc
  lblOutput.Text = sb.ToString()
End Sub



Private Sub WalkSiblings(ByVal nav As XPathNavigator)
  'Move to "title" element and get value
  Dim firstName As String = String.Empty
```

LISTING 11.8 Continued

```
  Dim lastName As String = String.Empty
  nav.MoveToFirstChild()
  Dim title As String = nav.Value
  sb.Append((title + "<br />"))

  'Move back to book element
  nav.MoveToParent()
  'access author element under book
  Dim authorNode As XPathNodeIterator = _
    nav.SelectChildren("author", "")
  While authorNode.MoveNext()
    'Move to first-name element
    authorNode.Current.MoveToFirstChild()
    firstName = authorNode.Current.Value
    'Move to last-name element
    authorNode.Current.MoveToNext()
    lastName = authorNode.Current.Value
    sb.Append((firstName + " " + lastName + "<br />"))
  End While

  'Now move to price element
  Dim priceNode As XPathNodeIterator = _
    nav.SelectChildren("price", "")
  priceNode.MoveNext()
  'Write out value of price element
  sb.Append(("$" + priceNode.Current.Value + "<br />"))

  'Search books by author and filter out unwanted books
  Dim otherBookNodes As XPathNodeIterator = _
    nav.Select(("//book[author/first-name='" + firstName + _
    "' and author/last-name='" + lastName + _
    "' and title != """ + title + """]/title"))
  sb.Append("<i>Other Books:</i> <br />")
  'Add other books to output
  If otherBookNodes.Count > 0 Then
    While otherBookNodes.MoveNext()
      sb.Append((otherBookNodes.Current.Value + "<br />"))
    End While
  Else
    sb.Append("None")
  End If
  sb.Append("<p />")
End Sub
```

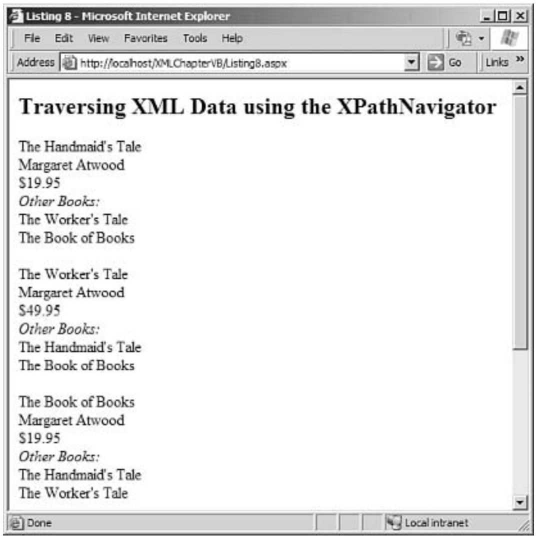


FIGURE 11.1 Accessing `book` and `author` nodes by using the `XPathNavigator` API.

> **Alternatives to `XPathNavigator`**
>
> **The `XmlDocument` and `DataSet` classes could also be used to search and filter the XML document shown in Listing 11.7. However, because no editing operations were performed, the `XPathDocument` and `XPathNavigator` combination provides a more efficient solution.**

Sorting XML Data

In the past, applications that required XML data to be sorted have typically relied on XSLT and the `xsl:sort` element due to XPath's lack of native support for sorting data. Although using XSLT to sort can get the job done, writing stylesheets and templates is often overkill and doesn't work for all types of data sorts. In fact, the XSLT 1.0 specification only supports text and numeric sorts "out of the box."

Fortunately, the reliance on XSLT to sort XML data is minimized in the .NET Framework due to native XML sorting capabilities found in the `XPathExpression` and `DataView` classes. In addition to being able to perform textual and numeric sorts, you can use the `XPathExpression` class to perform custom sorts, using objects that implement the `IComparer` interface. You can also use the `DataView` class to sort data loaded into a `DataTable` instance. The following sections demonstrate how to sort XML data by using the `XPathNavigator` and `XPathExpression` classes and provide details on how to leverage the `IComparer` interface. They also demonstrate how to sort XML data by using XSD schemas, along with `DataTable` and `DataView` instances.

Sorting with the `XPathExpression` Class

You can sort XML nodes by first compiling an XPath statement into an `XPathExpression` object. This is accomplished by calling the `XPathNavigator` class's `Compile()` method. Then you can add a text or numeric sort to the `XPathExpression` object by calling its `AddSort()` method. `AddSort()` has two overloads:

```
Overloads Public MustOverride Sub AddSort( _
   ByVal expr As Object, _
   ByVal comparer As IComparer _
)

Overloads Public MustOverride Sub AddSort( _
   ByVal expr As Object, _
   ByVal order As XmlSortOrder, _
   ByVal caseOrder As XmlCaseOrder, _
   ByVal lang As String, _
   ByVal dataType As XmlDataType _
)
```

The first of these overloads allows a custom object implementing `IComparer` to be used to perform sorts. This is useful when more advanced sorts need to take place. The second overload accepts a sort key, the sort order (ascending or descending), a value indicating how to sort uppercase and lowercase text, a language value, and the type of sort to perform (text or numeric). Listing 11.9 shows how to use the `Compile()` and `AddSort()` methods to sort the news headlines shown in Listing 11.1. The code sorts the headlines based on the `title` element, in ascending order.

LISTING 11.9 Sorting XML Data by Using the `XPathNavigator` and `XPathExpression` Classes

```
Dim sorted As New StringBuilder
'XPath statement
Dim xpath As String = "/moreovernews/article/headline_text"

'Create XPathDocument class so we can get a navigator
Dim doc As New XPathDocument(Server.MapPath("Listing1.xml"))
Dim nav As XPathNavigator = doc.CreateNavigator()

'Compile xpath expression
Dim exp As XPathExpression = nav.Compile(xpath)
'Add a sort based upon the headline_text child text node
exp.AddSort("text()", XmlSortOrder.Ascending, XmlCaseOrder.None, _
  "", XmlDataType.Text)

'select nodes so we can see the sort
Dim it As XPathNodeIterator = nav.Select(exp)
While it.MoveNext()
  'Grab headline_text value
  Dim headline As String = it.Current.Value
  'Move to article
  it.Current.MoveToParent()
  'Move to url
  it.Current.MoveToFirstChild()
```

LISTING 11.9 Continued

```
  'Grab url
  Dim url As String = it.Current.Value
  sorted.Append("<tr><td><a href=""")
  sorted.Append(url)
  sorted.Append(""">")
  sorted.Append(headline)
  sorted.Append("</a></td></tr>")
End While
Me.lblNews.Text = sorted.ToString()
```

Although this type of sorting works well for basic text or numeric sorts, what if you need to sort a set of nodes based on a `Date` data type? Fortunately, one of the `AddSort()` overloads shown earlier in this section allows a custom object that implements the `IComparer` interface to be passed to it. `IComparer` has a single method, named `Compare()`, that you can use to perform a variety of object comparisons. Listing 11.10 shows a simple class named `DateComparer` that implements the `Compare()` method.

LISTING 11.10 Creating a Custom Sort Class That Implements `IComparer`

```
Imports System.Collections
Public Class DateComparer : Implements IComparer

  Public Function Compare(ByVal date1 As Object, _
    ByVal date2 As Object) As Integer Implements IComparer.Compare
    Dim intResult As Integer
    Dim d1 As DateTime = Convert.ToDateTime(date1)
    Dim d2 As DateTime = Convert.ToDateTime(date2)
    intResult = DateTime.Compare(d1, d2)
    Return intResult * -1
  End Function

End Class
```

The `DateComparer` class works by accepting two objects that are converted to `DateTime` types. Upon conversion, the objects are compared to each other, using the `DateTime` object's `Compare()` method. `Compare()` returns an integer value from `-1` to `1`, depending on how the dates compare. A value of `0` means that the two dates are equal, and a value of `-1` or `1` means that one of the dates is greater than the other. (See the .NET Framework SDK for more details.) The integer created by calling `DateTime.Compare()` is returned from the `DateComparer` class's `Compare()` method and used by the `XPathExpression` class to perform the sorting. Listing 11.11 shows an example of using `DateComparer` in conjunction with the `XPathExpression` class.

LISTING 11.11 Performing Custom Sorts with the `XPathExpression` Class

```
Dim sorted As New StringBuilder
Dim xpath As String = "/moreovernews/article/harvest_time"

'Create XPathDocument class so we can get a navigator
Dim doc As New XPathDocument(Server.MapPath("Listing1.xml"))
Dim nav As XPathNavigator = doc.CreateNavigator()

'Compile xpath expression so we can add a sort to it
Dim exp As XPathExpression = nav.Compile(xpath)
'Create IComparer object
Dim dc As New DateComparer
'Pass IComparer object to AddSort()
exp.AddSort("text()", dc)

'select nodes so we can see the sort
Dim it As XPathNodeIterator = nav.Select(exp)
While it.MoveNext()
  'Grab harvest_time value
  Dim [date] As String = it.Current.Value
  'Move to article parent
  it.Current.MoveToParent()
  'Move to url
  it.Current.MoveToFirstChild()
  'Grab url
  Dim url As String = it.Current.Value
  'Move to headline
  it.Current.MoveToParent()
  Dim headlineIt As XPathNodeIterator = _
    it.Current.SelectChildren("headline_text", String.Empty)
  headlineIt.MoveNext()
  Dim headline As String = headlineIt.Current.Value

  sorted.Append("<tr><td><a href=""")
  sorted.Append(url)
  sorted.Append(""">")
  sorted.Append(headline)
  sorted.Append("</a></td><td>")
  sorted.Append([date])
  sorted.Append("</td></tr>")
End While
'Add data to a PlaceHolder server control named phNews
Me.phNews.Controls.Add(New LiteralControl(sorted.ToString()))
```

Figure 11.2 shows the HTML output generated after running the code shown in Listing 11.11. Notice that the news headlines are properly sorted by date and time.

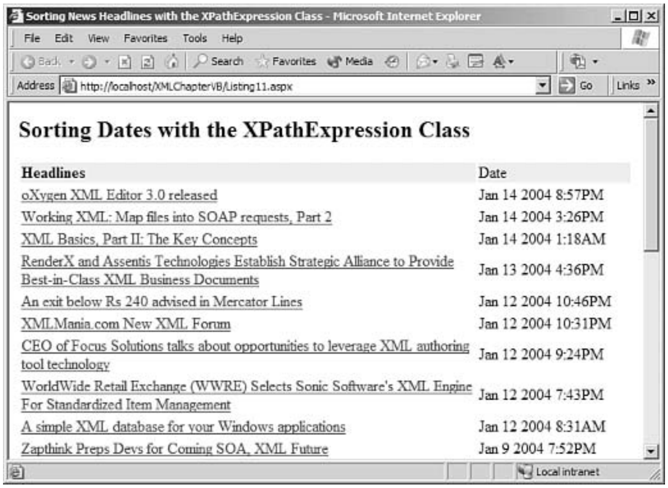


FIGURE 11.2
The result of sorting XML nodes based on date and time.

Sorting with the `DataView` Class

You can use ADO.NET's `DataView` class in combination with the `DataSet` and `DataTable` classes to sort XML data. Using the `DataView` class to sort XML is generally attractive to ASP.NET developers because they can use it to bind views to a variety of ASP.NET controls, including the `DataGrid` control. When you set it up properly, you can even use the `DataView` class to sort dates found within XML data, without resorting to using a custom class that implements the `IComparer` interface discussed earlier.

To sort by using the `DataView` class, you must first load the XML data into a `DataSet` instance by calling the `ReadXml()` method. You can then create a `DataView` object based on the appropriate `DataTable` within the `DataSet` instance. To sort based on a specific column, you assign the `DataColumn` name to the `DataView` object's `Sort` property. You can then assign the `DataView` object to the `DataSource` property of a variety of ASP.NET Web controls.

Following these steps works well for text-based sorts, but what happens if you want to sort numerically or by date? To sort based on dates contained within an XML document, the column representing the date data must be defined as a `DateTime` type within the `DataTable` object. Although you can do this programmatically, a more flexible solution is to preload an XSD schema that describes the XML document structure and its types into the `DataSet` object. The XSD schema can be loaded by calling the `DataSet` object's `ReadXmlSchema()` method. When you load the schema into the `DataSet` instance, all the `DataColumn` instances will be properly typed so that sorting can occur on different types (such as `DateTime`) by using the `DataView` object.

Before showing the code to sort XML data by date using a `DataView` instance, we need to mention a gotcha. XSD schema date types format dates differently than do Common Language

Runtime (CLR) `DateTime` types. For example, you can load the `harvest_time` element shown in Listing 11.1 into a `DateTime` structure by using its `Parse()` method:

```
Dim date as DateTime = DateTime.Parse("Jan 14 2004 8:57PM")
```

However, this date is not valid, according to the XSD schema specification. As a result, it will cause an error when it is loaded into a `DataSet` instance that has been preloaded with an XSD schema defining the `harvest_time` element as a `Date` data type. To make the data conform to the `Date` data type defined in the schema specification, you need to change it to the following format:

```
2004-01-14T20:57:00.0000000-07:00
```

Although you could potentially do this conversion by hand, the `XmlConvert` class can handle it with a single line of code (see Listing 11.12). Failure to properly perform this conversion will result in an error when the XML is loaded into the `DataSet` instance:

```
String was not recognized as a valid DateTime.
```

Although this gotcha causes a minor inconvenience when you're trying to sort the news data in Listing 11.1 by `harvest_time`, you can easily overcome it by using the `XmlDocument` and `XmlConvert` classes to manipulate the date values. The code in Listing 11.12 shows how to use these classes as well as perform several other tasks, including the following:

- Loading the news XML data into the DOM
- Converting all `harvest_time` text node values to valid schema `Date` data types by using the `XmlConvert` class's `ToString()` method
- Serializing the DOM structure to a `MemoryStream` object
- Loading the `MemoryStream` object into a `DataSet` instance that is preloaded with an XSD schema to properly type the different `DataTable` columns
- Creating a `DataView` object based on the `DataSet` object's first `DataTable`
- Identifying a sort column by using the `DataView` object's `Sort` property
- Binding the `DataView` to an ASP.NET `DataGrid` server control

LISTING 11.12 Sorting XML Data by Using the `DataView` Class

```
'Fix Listing1.xml dates to be schema "compatible" using DOM
Dim doc As New XmlDocument
doc.Load(Server.MapPath("Listing1.xml"))
'Find all harvest_time nodes
Dim dateNodes As XmlNodeList = doc.SelectNodes("//harvest_time")
For Each dateNode as XmlNode In dateNodes
  Dim newDate As DateTime = DateTime.Parse(dateNode.InnerText)
  'Convert harvest_time string to XSD Schema data type string
```

LISTING 11.12 Continued

```
  dateNode.InnerText = XmlConvert.ToString(newDate)
Next dateNode

'Save updated harvest_time XML to a Stream
Dim ms As New MemoryStream
doc.Save(ms)
ms.Position = 0

Dim ds As New DataSet
'Load schema
ds.ReadXmlSchema(Server.MapPath("Listing12.xsd"))
'Load XML data into DataSet
ds.ReadXml(ms)

'Create DataView
Dim view As DataView = ds.Tables(0).DefaultView
'Sort on date column
view.Sort = "harvest_time DESC"
Me.dgNews.DataSource = view
Me.dgNews.DataBind()
ms.Close()
```

Figure 11.3 shows the result of sorting the XML headlines based on `harvest_time`.

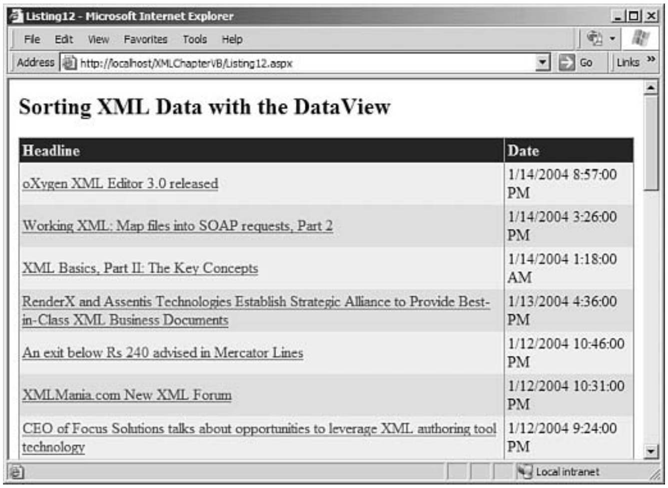


FIGURE 11.3 The result of sorting XML nodes based on date and time with a `DataView` instance.

Searching Namespace Qualified Nodes

The .NET Framework prevents naming collisions by logically organizing classes into namespaces. XML documents also prevent naming collisions by using namespaces, although the way they are defined is quite different. Namespace qualified nodes are logically separated from other nodes (think of XML nodes as being organized into different rooms in a building based on their namespace URIs) to make them easy to locate and to avoid collisions. Two different types of XML namespaces exist: default and local.

The following is an example of defining a default namespace:

```
<?xml version="1.0"?>
<moreovernews xmlns="http://www.moreover.com">
   <!-- Article nodes go here -->
</moreovernews>
```

Because the default namespace shown here is defined at the root level, all children of the `moreovernews` element are members of this namespace.

You create a local namespace by defining a namespace prefix along with a unique URI, as in this example:

```
<?xml version="1.0" ?>
<news:moreovernews xmlns:news="http://www.moreover.com">
   <!-- Article nodes go here -->
</news:moreovernews>
```

Nodes that have the `news` prefix prepended to their names are placed in the local `www.moreover.com` namespace. Nodes that do not have this prefix are in a separate namespace.

When XML namespaces are added to an XML document, you must take them into account when searching or filtering nodes by using XPath. Failure to account for namespaces results in no matches being returned. You search for nodes in a default or local namespace by using the `XmlNamespaceManager` class. `XmlNamespaceManager` has an `AddNamespace()` method that accepts a namespace prefix and URI, as in this example:

```
Public Overridable Sub AddNamespace( _
   ByVal prefix As String, _
   ByVal uri As String _
)
```

Although `XmlNamespaceManager` is often used when namespaces need to be dynamically added into XML fragments, it can also be used when executing XPath statements. To query article nodes located in a default namespace (such as the one shown earlier in this section), you can add the default namespace to the `XmlNamespaceManager` instance and then use it in the XPath statement. The code shown in Listing 11.13 illustrates this process. Adding the `XmlNamespaceManager` namespace data into the context of the XPath statement is accomplished by using the `XPathExpression` class's `SetContext()` method.

LISTING 11.13 Searching for Nodes in a Default Namespace by Using `XpathNavigator`

```
Dim xmlPath As String = Server.MapPath("Listing13.xml")
'Load XML
Dim doc As New XPathDocument(xmlPath)
'Create navigator
Dim nav As XPathNavigator = doc.CreateNavigator()
Dim ns As New XmlNamespaceManager(nav.NameTable)

'Define default namespace.  Prefix can be any valid XML namespace
'prefix value
ns.AddNamespace("ns", "http://www.moreover.com")
'Add default prefix into xpath statement to account for
'default namespace
Dim xpath As String = "/ns:moreovernews/ns:article/" + "ns:headline_text"

'Create a compiled xpath statement and set context to include
'the namespace manager data.
Dim exp As XPathExpression = nav.Compile(xpath)
exp.SetContext(ns)

'Select nodes and write out the headlines
Dim it As XPathNodeIterator = nav.Select(exp)
While it.MoveNext()
  lblOutput.Text += it.Current.Value + "<br />"
End While
```

Querying local namespace nodes involves the same process shown in Listing 11.13, although the prefix value passed to the `AddNamespace()` method must match the namespace prefix defined in the XML document. Listing 11.14 demonstrates how to use XPath to query nodes in a local namespace.

LISTING 11.14 Searching for Nodes in a Local Namespace by Using `XPathNavigator`

```
Dim xmlPath As String = Server.MapPath("Listing14.xml")
'Load XML
Dim doc As New XPathDocument(xmlPath)
'Create navigator
Dim nav As XPathNavigator = doc.CreateNavigator()
Dim ns As New XmlNamespaceManager(nav.NameTable)
'Define news namespace prefix and URK.
ns.AddNamespace("news", "http://www.moreover.com")
'Add news prefix into xpath statement
Dim xpathNS As String = "/moreovernews/news:article/headline_text"
'Create a compiled xpath statement and set context to include
'the namespace manager data.
```

LISTING 11.14 Continued

```
Dim exp As XPathExpression = nav.Compile(xpathNS)
exp.SetContext(ns)
'Select nodes and write out the headlines
Dim it As XPathNodeIterator = nav.Select(exp)
While it.MoveNext()
  Me.lblLocalNS.Text += it.Current.Value + "<br />"
End While

'Locate articles not in the http://www.moreover.com namespace
Dim xpathDefault As String = "/moreovernews/article/headline_text"
Dim expDefault As XPathExpression = nav.Compile(xpathDefault)
expDefault.SetContext(ns)
'Select nodes and write out the headlines
Dim it2 As XPathNodeIterator = nav.Select(expDefault)
While it2.MoveNext()
  Me.lblNoNamespace.Text += it2.Current.Value + "<br />"
End While
```

You can also use the `XmlNamespaceManager` object to search for namespace qualified nodes, using the `XmlDocument` class, as shown in Listing 11.15. The `XmlDocument` class's `SelectNodes()` method (which is inherited from `XmlNode`) contains an overload that accepts an `XmlNamespaceManager` object as a parameter.

LISTING 11.15 Searching for Nodes in a Local Namespace by Using `XmlDocument`

```
Dim xmlPath As String = Server.MapPath("Listing14.xml")
Dim doc As New XmlDocument
doc.Load(xmlPath)
Dim ns As New XmlNamespaceManager(doc.NameTable)
ns.AddNamespace("news", "http://www.moreover.com")
Dim xpathLocal As String = "/moreovernews/news:article/headline_text"
Dim newsNodes As XmlNodeList = doc.SelectNodes(xpathLocal, ns)
For Each newsNode As XmlNode In newsNodes
  Me.lblLocalNS.Text += newsNode.InnerText + "<br />"
Next newsNode

'Select nodes not in http://www.moreover.com namespace
Dim xpathDefault As String = "/moreovernews/article/headline_text"
Dim nonNSNodes As XmlNodeList = doc.SelectNodes(xpathDefault)
For Each newsNode As XmlNode In nonNSNodes
  Me.lblNoNamespace.Text += newsNode.InnerText + "<br />"
Next newsNode
```

Creating a Reusable XML Validation Class

In addition to creating Web form front-end code, ASP.NET programmers are often charged with developing a variety of back-end processes, such as those that access remote XML data and store it in a database for later retrieval. These types of processes may involve validating the XML data to ensure that it is structured properly and contains valid data types that properly match up with database fields. By validating XML data first, you can catch potential errors ahead of time, before any SQL statements are executed in the database.

The .NET Framework supports validating XML documents using several different types of documents including DTDs, XML Data-Reduced (XDR) schemas, and XSD schemas. XSD schemas offer the most power and flexibility of the three choices, through their support for validating a document's structure as well as the data types it contains. You can find more information about XSD schemas at the W3C Web site: `www.w3.org`.

XML documents can be programmatically validated by using the `XmlValidatingReader` class located in the .NET Framework's `System.Xml` namespace. You can use this class to validate documents against DTD, XDR, or XSD schema documents. Like the `XmlTextReader` class, it provides a fast, forward-only API that can handle large XML documents quickly and efficiently. `XmlValidatingReader` exposes a `ValidationHandler` event that is called when the validation process errors, such as when incorrect element nesting or invalid data types are encountered.

Although you can write validation code from scratch each time you need to validate an XML document, encapsulating validation code into a wrapper class brings many object-oriented coding benefits, including encapsulation and code reuse. If you write a wrapper class, developers with different skill levels can perform XML document validation more easily; also, validation code can be simplified when multiple applications share the same code base.

Listing 11.16 contains the skeleton for a reusable XML validation component that uses the `XmlValidatingReader` class. The `XmlValidator` class relies on a helper structure named `XmlValidationStatus` to report if XML documents are valid to calling applications.

LISTING 11.16 A Skeleton for a Reusable XML Validator Class and Helper Structure

```
Public Class XmlValidator

  Public Function Validate(ByVal xml As Object, _
    ByVal schemaCol As XmlSchemaCollection, _
    ByVal dtdInfo() As String, ByVal logError As Boolean, _
    ByVal logFile As String) As XmlValidationStatus
  End Function


  Private Sub ValidationCallBack(ByVal sender As Object, _
    ByVal args As ValidationEventArgs)
  End Sub
```

LISTING 11.16 Continued

```
End Class

Public Structure XmlValidationStatus
  Public Status As Boolean
  Public ErrorMessages As String
End Structure
```

The `XmlValidator` class has a single public method named `Validate()` that accepts the XML data source to be validated, an `XmlSchemaCollection` object, a `String` array containing DTD information (used when validating against DTDs), a `Boolean` parameter used to turn logging on and off, and the path to the log file that is used when logging is enabled.

The `logError` and `logFile` parameters are self-explanatory, but the others need further explanation. The `xml` parameter is typed as `Object` to allow different types of XML data sources to be validated. Valid XML data source types include `StringReader`, `String`, and `Stream`. Passing any other types for the `xml` parameter value will cause an `ApplicationException` error to be thrown. The `schemaCol` parameter accepts an `XmlSchemaCollection` instance (`XmlSchemaCollection` is located in the `System.Xml.Schema` namespace) that contains one or more schemas used to validate the XML data source. When DTDs are used for validation, the DTD `DocTypeName` (the root element of the XML document) is passed as the first item in the `String` array, followed by the physical path to the DTD document. Listing 11.17 shows the complete code for the `Validate()` method.

LISTING 11.17 The `Validate()` and `ValidationCallBack()` Methods

```
Private _valid As Boolean
Private _logError As Boolean
Private _logFile As String
Private _validationErrors As String = String.Empty
Private xmlReader As XmlTextReader = Nothing
Private vReader As XmlValidatingReader = Nothing

Public Function Validate(ByVal xml As Object, _
  ByVal schemaCol As XmlSchemaCollection, _
  ByVal dtdInfo() As String, ByVal logError As Boolean, _
  ByVal logFile As String) As XmlValidationStatus
    _logError = logError
    _logFile = logFile
    _valid = True

    Try
      'Check what type of XML data source was passed
      If TypeOf xml Is StringReader Then
        xmlReader = New XmlTextReader(CType(xml, StringReader))
      ElseIf TypeOf xml Is String Then
        xmlReader = New XmlTextReader(CType(xml, String))
```

LISTING 11.17 Continued

```
      ElseIf TypeOf xml Is Stream Then
        xmlReader = New XmlTextReader(CType(xml, Stream))
      Else
        Throw New ApplicationException("Invalid XML data " + _
         "source passed.")
      End If
      'Hookup DTD or Schemas
      If Not (dtdInfo Is Nothing) Then
        If dtdInfo.Length > 0 Then
          Dim context As New XmlParserContext(Nothing, Nothing, _
            dtdInfo(0), "", dtdInfo(1), "", dtdInfo(1), "", _
            XmlSpace.Default)
          xmlReader.MoveToContent()
          vReader = _
            New XmlValidatingReader(xmlReader.ReadOuterXml(), _
              XmlNodeType.Element, context)
          vReader.ValidationType = ValidationType.DTD
        End If
      Else
        vReader = New XmlValidatingReader(xmlReader)
        vReader.ValidationType = ValidationType.Auto
        If Not (schemaCol Is Nothing) Then
          vReader.Schemas.Add(schemaCol)
        End If
      End If
      'Associate validating reader with callback method
      'to handle any validation errors
      AddHandler vReader.ValidationEventHandler, _
        AddressOf Me.ValidationCallBack
      ' Parse through XML document
      While vReader.Read()
      End While
    Catch
      _valid = False
    Finally 'Close validating reader
      If Not (vReader Is Nothing) Then
        vReader.Close()
      End If
    End Try

    'Report back to calling application
```

LISTING 11.17 Continued

```
    Dim status As New XmlValidationStatus
    status.Status = _valid
    status.ErrorMessages = _validationErrors
    Return status
  End Function


  Private Sub ValidationCallBack(ByVal sender As Object, _
    ByVal args As ValidationEventArgs)
    _valid = False 'hit callback so document has a problem
    Dim today As DateTime = DateTime.Now
    Dim writer As StreamWriter = Nothing
    Try
      If _logError Then 'Handle logging to logfile
        writer = New StreamWriter(_logFile, True, Encoding.ASCII)
        writer.WriteLine("Validation error in XML: ")
        writer.WriteLine()
        writer.WriteLine((args.Message + " " + today.ToString()))
        writer.WriteLine()
        If xmlReader.LineNumber > 0 Then
          writer.WriteLine(("Line: " + xmlReader.LineNumber + _
          " Position: " + xmlReader.LinePosition))
        End If
        writer.WriteLine()
        writer.Flush()
      Else 'Track error messages
        _validationErrors = args.Message + " Line: " + _
        xmlReader.LineNumber.ToString() + _
        " Column:" + xmlReader.LinePosition.ToString() + _
        ControlChars.Lf + ControlChars.Lf
      End If
    Catch
    Finally 'Ensure StreamWriter gets closed
      If Not (writer Is Nothing) Then
        writer.Close()
      End If
    End Try
  End Sub
```

`Validate()` starts by loading the XML data source into an `XmlTextReader` instance, hooking up schemas or DTDs, and then instantiating the `XmlValidatingReader` instance. Any errors

> **Other Uses for the `XmlValidator` Class**
>
> **Although the example in Listing 11.18 uses the `XmlValidator` class from within an ASP.NET page, a more realistic and useful approach might be to have a Windows service that automatically grabs XML documents from a variety of locations and validates them. Valid XML documents could then be moved into a database or stored on the file system for later retrieval. You can find an example of creating a Windows service for this purpose at `www.xmlforasp.net/codeSection.aspx?csID=77`.**

encountered during the XML validation process cause the `ValidationCallBack` method to be called; this method handles tracking and logging errors. Upon completion, the `Validate()` method creates an `XmlValidationStatus` structure and assigns appropriate values to its fields.

Listing 11.18 provides an example of putting the `XmlValidator` class to use. Any errors found during the validation operation are written back to the page in this example, but they could instead be logged to a file.

LISTING 11.18 Using the `XmlValidator` Class to Validate XML Data

```
'Define logging folder to use when logging is turned on
Dim logFile As String = Server.MapPath("Log.txt")
Dim xmlFilePath As String = Server.MapPath("Listing18.xml")

'Create schema collection object and add schema to it
Dim schemaCol As New XmlSchemaCollection
schemaCol.Add(String.Empty, Server.MapPath("Listing18.xsd"))

'Create XmlValidator and call Validate() method
Dim validator As New XmlValidator
Dim valStatus As XmlValidationStatus = _
  validator.Validate(xmlFilePath, schemaCol, Nothing, _
  False, logFile)
If valStatus.Status = True Then
  Me.lblOutput.Text = "<b>Validation was SUCCESSFUL!</b>"
  'Call method to process XML document for backend process
Else
  Me.lblOutput.Text = "<b>Validation failed!</b><p />"
  Me.lblOutput.Text += valStatus.ErrorMessages
End If
```

Converting Relational Data to XML

Web applications have come a long way since the early days of the Internet. Many of the first applications relied on data stored in local databases or flat files and provided little to no flexibility for accessing data from distributed sources. As the Internet has evolved, more advanced data access technologies have come about that allow data from a variety of locations and sources to be used in Web applications. This has resulted in companies automating business processes and ultimately cutting operational costs.

ADO.NET represents one of the most powerful technologies to have evolved out of the old technologies of the Internet. When you use ADO.NET, only a few lines of code are required to load data from a relational database and convert it to XML for transport between different business entities, binding to hierarchical controls, transformation with XSLT, and many other purposes. The following sections provide several pure .NET Framework techniques for converting relational data to XML and show how you can customize the structure of an XML document.

Customizing XML by Using the `DataSet` Class

The `DataSet` class exposes two methods named `GetXml()` and `WriteXml()` that can be used to easily convert relational data into XML. `GetXml()` returns a string that contains the XML data, and `WriteXml()` can write XML data to a file or to a `TextWriter`, `Stream`, or `XmlWriter` instance. Both methods generate XML documents that are element-centric. The root node of the generated XML is named after the `DataSet` instance, and each child of the root is named based on `DataTable` instances in the `DataSet` instance.

Although the default XML structure generated by the `DataSet` instance might be fine for some applications, others might require the structure to be customized so that the data can be integrated into another application or matched up with a schema. You can customize the XML structure by using the `DataColumn` and `DataRelation` classes. You can use the `DataColumn` class to control whether data is mapped to elements or attributes, and you can use the `DataRelation` class to control nesting.

After a `DataSet` instance is filled with data from a database, each `DataColumn` instance (within the respective `DataTable` instances) can have its `ColumnMapping` property set to one of several `MappingType` enumeration values. Table 11.2 lists these values.

TABLE 11.2

`MappingType` **Enumeration Values**

`MappingType` **Value**	**Functionality**
`Element`	Data is mapped to an element. This is the default behavior.
`Attribute`	Data is mapped to an attribute.
`Hidden`	Data is not output in the generated XML.
`SimpleContent`	Data is mapped to an `XmlText` node.

Changing the `MappingType` value allows you to shape the XML data as desired. Listing 11.19 demonstrates how to load data from the Northwind database's `Customers` table into a `DataSet` instance and use the `ColumnMapping` property of the `DataColumn` class to associate primary key data with an attribute.

LISTING 11.19 Shaping XML Data by Using the `ColumnMapping` Property

```
Dim connStr As String = ConfigurationSettings.AppSettings("ConnStr")
Dim sql As String = "SELECT * FROM Customers " + _
  "WHERE CustomerID = 'ALFKI'"
Dim conn As New SqlConnection(connStr)
```

LISTING 11.19 Continued

```
Dim da As New SqlDataAdapter(sql, conn)

'Provide root name for XML document
Dim ds As New DataSet("Customers")

'Provide name for each child element of root
da.Fill(ds, "Customer")

'Map CustomerID field to an attribute
ds.Tables(0).Columns("CustomerID").ColumnMapping = _
  MappingType.Attribute
Me.txtXml.Text = ds.GetXml()
conn.Close()
```

The following XML is generated after running the code in Listing 11.19 (notice that the document's root node is named after the `DataSet` instance and that the `CustomerID` data is defined as an attribute):

```
<Customers>
  <Customer CustomerID="ALFKI">
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <ContactName>Maria Anders</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Address>Obere Str. 57</Address>
    <City>Berlin</City>
    <PostalCode>12209</PostalCode>
    <Country>Germany</Country>
    <Phone>030-0074321</Phone>
    <Fax>030-0076545</Fax>
  </Customer>
</Customers>
```

In cases in which relational tables have primary and foreign-key relationships, you can further customize XML data to reflect the relationships. For example, all orders placed by a customer can be nested under the proper `Customer` element in the XML document. Listing 11.20 shows how you can programmatically define relationships by using the `DataRelation` class and how you can nest those relationships by using its `Boolean Nested` property. After you define a relationship, you can add it to the `DataSet` object instance through its `Relations` collection.

LISTING 11.20 Nesting XML Based on Primary/Foreign-Key Relationships

```
Dim connStr As String = ConfigurationSettings.AppSettings("ConnStr")
Dim sql As String = "SELECT * FROM " + _
  "Customers WHERE CustomerID = 'ALFKI';"
sql += "SELECT * FROM Orders WHERE CustomerID = 'ALFKI'"
```

LISTING 11.20 Continued

```
Dim conn As New SqlConnection(connStr)
Dim da As New SqlDataAdapter(sql, conn)

'Provide root name for XML document
Dim ds As New DataSet("CustomersOrders")

'Fill DataSet with 2 tables worth of data
da.Fill(ds)

'Provide names for DataTables
ds.Tables(0).TableName = "Customer" '
ds.Tables(1).TableName = "Order" '

'Create primary/foreign-key relationships
Dim pk As DataColumn = ds.Tables(0).Columns("CustomerID")
pk.ColumnMapping = MappingType.Attribute
Dim fk As DataColumn = ds.Tables(1).Columns("CustomerID")
Dim r As New DataRelation("CustOrders", pk, fk)
r.Nested = True

'Add relationship to DataSet
ds.Relations.Add(r)
Me.txtXml.Text = ds.GetXml()
conn.Close()
```

Listing 11.21 shows a portion of the XML data created after you run the code in Listing 11.20.

Defining Relationships in an XSD Schema

You can automatically load relationships between `DataTable` objects into a `DataSet` (as opposed to programmatically defining them) by defining them in an XSD schema. You can then load the schema into the `DataSet` instance by calling its `ReadXmlSchema()` method. Schemas define relationships by using the `key` and `keyref` elements.

LISTING 11.21 Nested XML Data

```
<CustomersOrders>
  <Customer CustomerID="ALFKI">
    <!-- Children omitted for brevity -->
    <Order>
      <OrderID>10643</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <EmployeeID>6</EmployeeID>
      <OrderDate>1997-08-25T00:00:00.0000000-07:00</OrderDate>
      <RequiredDate>1997-09-22T00:00:00.0000000-07:00</RequiredDate>
      <ShippedDate>1997-09-02T00:00:00.0000000-07:00</ShippedDate>
      <ShipVia>1</ShipVia>
      <Freight>29.4600</Freight>
      <ShipName>Alfreds Futterkiste</ShipName>
      <ShipAddress>Obere Str. 57</ShipAddress>
```

LISTING 11.21 Continued

```
      <ShipCity>Berlin</ShipCity>
      <ShipPostalCode>12209</ShipPostalCode>
      <ShipCountry>Germany</ShipCountry>    </Order>
    <!-- Other Order nodes omitted for brevity -->
  </Customer>
</CustomersOrders>
```

Adding CDATA Sections into XML Documents

The DataSet class makes it extremely easy to shape XML data in a variety of ways. However, when data retrieved from a database needs to be wrapped with a CDATA section (`<![CDATA[`*`data goes here`*`]]>`) so that an XML parser does not parse the data, the DataSet instance provides no native CDATA MappingType enumeration value. CDATA sections may be necessary when data retrieved from a relational database contains HTML code or script blocks, as is often the case when you're working with different content management systems.

Although no native CDATA MappingType enumeration value exists, you can add CDATA sections to XML data by taking advantage of native .NET Framework XML APIs. Listing 11.22 shows a custom class named CDataSet that derives from DataSet and overloads the GetXml() method to handle adding CDATA sections.

LISTING 11.22 Extending the DataSet Class to Support CDATA Sections

```
Public Class CDataSet : Inherits DataSet

  Public Overloads Function GetXml(ByVal cdataSections() _
    As String) As String
    Return InsertCDATASections(cdataSections)
  End Function

  Private Function InsertCDATASections(ByVal cdataSections() _
    As String) As String
    'Convert to XML with expanded general entities and CDATA sections
    'as appropriate
    Dim reader As XmlValidatingReader = Nothing
    Dim writer As XmlTextWriter = Nothing
    Dim sw As StringWriter = Nothing
    Array.Sort(cdataSections)
    Try
      reader = New XmlValidatingReader(Me.GetXml(), _
        XmlNodeType.Document, Nothing)
      sw = New StringWriter
      writer = New XmlTextWriter(sw)
      writer.Formatting = Formatting.Indented
```

LISTING 11.22 Continued

```
      reader.ValidationType = ValidationType.None
      'Expand character entities that will be in CDATA sections
      'so that characters such as &lt;script&gt; change to <script>
      reader.EntityHandling = EntityHandling.ExpandCharEntities
      Dim currentElement As String = String.Empty
      While reader.Read()
        Select Case reader.NodeType
          Case XmlNodeType.Element
            currentElement = reader.Name
            writer.WriteStartElement(currentElement)
            writer.WriteAttributes(reader, False)
          Case XmlNodeType.Text
            If Array.BinarySearch(cdataSections, _
              currentElement) < 0 Then
              writer.WriteString(reader.Value)
            Else 'Found CDATA DataColumn
              writer.WriteCData(reader.Value)
            End If
          Case XmlNodeType.EndElement
            writer.WriteEndElement()
          Case Else
        End Select
      End While
    Catch exp As Exception
      Return exp.Message
    Finally
      reader.Close()
      writer.Close()
    End Try
    Return sw.ToString()
  End Function

End Class
```

The `GetXml()` overload shown in Listing 11.22 accepts a string array that contains the names of `DataColumn` instances that need to have their data wrapped in `CDATA` sections when converted to XML. `GetXml()` relies on a private method named `InsertCDATASections()` that uses the `XmlValidatingReader` class to read the XML data created by calling the base class's `GetXml()` method. The `XmlTextWriter` class is then used to write the specialized `CDATA` XML data by calling the writer's `WriteCData()` method. `XmlValidatingReader` is used in this scenario because it is fast, like `XmlTextReader`, and it allows character entities such as `<` and `>` to be expanded by setting its `EntityHandling` property to `EntityHandling.ExpandCharEntities`.

Listing 11.23 provides an example of using the `CDataSet` class, and Figure 11.4 shows the different XML documents generated by calling the `DataSet` instance's `GetXml()` method versus calling the `CDataSet` instance's overloaded `GetXml()` method.

LISTING 11.23 Using the `CDataSet` Class to Generate `CDATA` Sections

```
'Identify elements that need to be wrapped with CDATA sections
Dim cdataArray As String() = {"headElement", "bodyElement"}
Dim ds As New CDataSet
'Code to fill CDataSet goes here

'Call normal GetXml() method
Me.txtXml.Text = ds.GetXml()
'Call overloaded GetXml() method and pass
'CDATA element names
Me.txtCdataXml.Text = ds.GetXml(cdataArray)
```

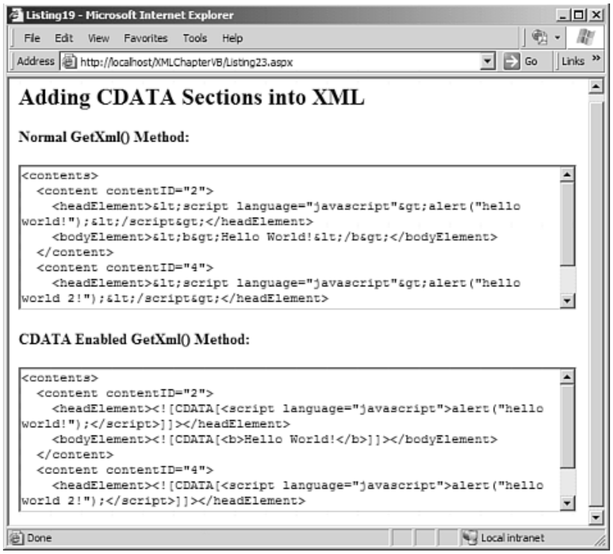


FIGURE 11.4
A comparison of the `DataSet` class's `GetXml()` method and the `CDataSet` class's overloaded `GetXml()` method.

Simplifying Configuration by Using XML

XML provides a convenient format for storing configuration settings due to its inherent support for data retrieval through XML APIs and the XPath language. Many .NET Framework files, including `web.config` and `machine.config`, use XML for storing different types of configuration settings, such as security information, type definitions, and assembly details. These files rely on XML because it is both human and machine readable and can be accessed and updated using a variety of programming techniques.

You can easily store custom ASP.NET configuration data such as database connection strings in files such as `web.config` and access them by using the `ConfigurationSettings` class. In cases where more complex data structures need to be stored, you can use section handlers and name/value pairs. You can also write configuration reader classes that implement the `IConfigurationSectionHandler` interface and use XPath (see `http://support.microsoft.com/default.aspx?scid=kb;EN-US;318457`). Although these customizations to `web.config` work well, they may require more work than they're worth as the configuration structure becomes more complex and more customized. In some cases, it's more straightforward to create a new XML configuration document that is separate from `web.config` and store application-specific configuration data in that file.

Separating custom configuration data from the `web.config` file is certainly not necessary and obviously causes an additional file to be moved when the ASP.NET application is deployed. However, creating a separate XML configuration file when more complex settings need to be stored can have a positive side effect that surfaces when changes to the configuration data must be made. ASP.NET has an in-memory cache that contains `web.config` data, and in the .NET Framework version 1.1, this cache is updated any time the `web.config` file is saved. This update routine causes the Web application to pause momentarily. When you store configuration settings in a separate file, you can avoid this pause when the settings are updated.

Accessing Configuration Settings by Using `XPathNavigator`

You can access data stored in a custom configuration file in several different ways. One of the most straightforward ways is to use the `XPathNavigator` class along with XPath to find nodes. When combined with caching and file dependencies, this approach offers good performance and requires a small amount of code to be written. Listing 11.24 contains an XML configuration document that marks up data related to different types of servers.

LISTING 11.24 A Custom XML Configuration Document with Server Settings

```
<?xml version="1.0" ?>
<ServerConfig>
  <ProxyServer Name="proxy.domain.com" Port="8080"
   UserName="jdoe" Password="password" Domain="myDomain" />
  <SmtpServer Name="localhost" />
  <SqlServer>
    <Prod ConnString="server=prod;uid=uid;pwd=pass;database=db;">
      <Servers>
        <Server>www.xmlforasp.net</Server>
        <Server>www.xml4asp.net</Server>
      </Servers>
    </Prod>
    <Dev ConnString="server=dev;uid=uid;pwd=pass;database=db;">
      <Servers>
        <Server>localhost</Server>
        <Server>127.0.0.1</Server>
      </Servers>
```

LISTING 11.24 Continued

```
    </Dev>
  </SqlServer>
</ServerConfig>
```

You can simplify the configuration data in Listing 11.24 by writing a configuration reader class that wraps functionality exposed by the `XPathDocument` and `XPathNavigator` classes. This class (named `ConfigReader` in the following listings) has several `Shared` methods (`static` methods, in C#) that rely on XPath to locate nodes. Listing 11.25 shows the complete code for the `ConfigReader` class.

LISTING 11.25 The `ConfigReader` Class

```
Public Class ConfigReader

  Public Shared Function GetConfigValue(ByVal xpath As String) _
    As String
    Dim doc As XPathDocument = GetConfigDocument()
    Dim nav As XPathNavigator = doc.CreateNavigator()
    nav.MoveToRoot()
    nav.MoveToFirstChild()
    Dim it As XPathNodeIterator = nav.Select(xpath)
    If Not (it Is Nothing) Then
      it.MoveNext()
      Return it.Current.Value
    Else
      Return Nothing
    End If
  End Function

  Public Shared Function GetConnectionString(ByVal server _
    As String) As String
    Return GetConfigValue(("SqlServer/" + _
      GetServerType(server).ToString() + "/@ConnString"))
  End Function

  Public Shared Function GetServerType(ByVal server As String) _
    As ServerType
    Dim currServer As String = server.ToLower()
    Dim doc As XPathDocument = GetConfigDocument()
    Dim nav As XPathNavigator = doc.CreateNavigator()
    nav.MoveToRoot()
    nav.MoveToFirstChild()
    Dim xpath As String = "//Servers/Server"
    Dim it As XPathNodeIterator = nav.Select(xpath)
```

LISTING 11.25 Continued

```
    While it.MoveNext()
      If it.Current.Value.ToLower() = currServer Then
        it.Current.MoveToParent() 'Move to ServerNames
        it.Current.MoveToParent() 'Move to server type
        Return CType([Enum].Parse(GetType(ServerType), _
          it.Current.Name, True), ServerType)
      End If
    End While
    'Default to dev server
    Return ServerType.Dev
  End Function

  Private Shared Function GetConfigDocument() As XPathDocument
    Dim context As HttpContext = HttpContext.Current
    'Check if config is already loaded into cache
    If context.Cache.Get("ServerConfig") Is Nothing Then
      Try
        Dim configPath As String = _
          context.Server.MapPath( _
          ConfigurationSettings.AppSettings("ServerConfig"))
        Dim doc As New XPathDocument(configPath)
        'Create file dependency for cache
        Dim cd As New CacheDependency(New String() {configPath})
        'Cache XPathDocument instance
        context.Cache.Insert("ServerConfig", doc, cd)
        Return doc
      Catch
      End Try
    Else
      'Return XPathDocument already in cache
      Return CType(context.Cache.Get("ServerConfig"), XPathDocument)
    End If
  End Function
End Class _

Public Enum ServerType
  Prod
  Dev
End Enum
```

You can retrieve a configuration value (such as the SMTP server name used to send email) via the `ConfigReader` class by calling the `GetConfigValue()` method. This method accepts an XPath statement as a parameter and executes it by using the `XPathNavigator` class's `Select()` method. If the XPath statement finds a node, the value (or text node, in the case of elements) is returned.

If no nodes are found, the method returns Nothing. Although the passwords in Listing 11.24 are shown in clear text, they can be encrypted and decrypted during this process for additional security.

The GetConfigValue() method relies on a private method named GetConfigDocument() that accesses the configuration file. ASP.NET caching is used in the GetConfigDocument() method to minimize the number of times the configuration file is accessed from disk. Any time the configuration file changes, the in-memory cache is invalidated and the configuration settings are reloaded into an XPathDocument instance.

The ConfigReader class can also be used to access different database connection strings based on the type of Web server calling the database. The GetConnectionString() and GetServerType() methods make it possible to retrieve development or production database connection strings with only a minimal amount of code:

```
Dim connStr As String = _
  ConfigReader.GetConnectionString(Request.UserHostName)
```

Although using the XPathNavigator API to access configuration data doesn't require you to write a lot of code, this combination of technologies can present a potential maintenance problem when the configuration structure changes or when node names change. These types of changes require updates to the XPath statements in the code, which means you must perform a search and replace. Because XPath statements are quoted, the compiler will never catch errors in the statements. Wouldn't it be nice if the compiler could automatically identify every line of code where a change or update needed to occur? Fortunately, this type of behavior is a reality, as the next section demonstrates.

Using XML Serialization

The .NET Framework contains a handy command-line tool named xsd.exe that you can use to perform a variety of tasks, such as converting XSD schemas to classes and vice versa. Although xsd.exe is best known for its ability to generate strongly typed DataSet classes, you can also use it to generate standard C# or Visual Basic .NET classes. You can then use these classes to access an XML document, using object-oriented techniques, as opposed to using .NET Framework XML APIs. Listing 11.26 shows an XSD schema that describes the XML configuration document shown in Listing 11.24.

LISTING 11.26 An XSD Schema That Describes the XML Configuration Document in Listing 11.24

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" version="1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="ServerConfig">
    <xsd:complexType>
```

LISTING 11.26 Continued

```
      <xsd:sequence>
        <xsd:element name="ProxyServer">
          <xsd:complexType>
            <xsd:attribute name="Name" type="xsd:string" />
            <xsd:attribute name="Port" type="xsd:string" />
            <xsd:attribute name="UserName" type="xsd:string" />
            <xsd:attribute name="Password" type="xsd:string" />
            <xsd:attribute name="Domain" type="xsd:string" />
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="SmtpServer">
          <xsd:complexType>
            <xsd:attribute name="Name" type="xsd:string" />
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="SqlServer">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Prod">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Servers">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element maxOccurs="unbounded"
                             name="Server" type="xsd:string" />
                        </xsd:sequence>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                  <xsd:attribute name="ConnString"
                    type="xsd:string" />
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="Dev">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Servers">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element maxOccurs="unbounded"
                            name="Server" type="xsd:string" />
                        </xsd:sequence>
                      </xsd:complexType>
```

LISTING 11.26 Continued

```
                    </xsd:element>
                  </xsd:sequence>
                  <xsd:attribute name="ConnString"
                    type="xsd:string" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

You can convert this schema into Visual Basic .NET classes by running the following code at the command prompt:

```
xsd.exe /classes /namespace:Configuration
  /language:VB Listing26.xsd
```

After you run this code, a Visual Basic .NET file named `Listing26.vb` that contains the following classes will be created:

- `ServerConfig`
- `ServerConfigProxyServer`
- `ServerConfigSqlServer`
- `ServerConfigSqlServerDev`
- `ServerConfigSqlServerProd`
- `ServerConfigSmtpServer`

To use these classes, you can put the `XmlSerializer` class (located in the `System.Xml.Serialization` namespace) to work. A call to the `XmlSerializer` class's `Deserialize()` method causes XML configuration data to be loaded into instances of the classes shown earlier. When instantiated, data within the classes can be accessed by using standard object-oriented techniques. Listing 11.27 shows an updated version of the `ConfigReader` class that uses the `XmlSerializer` class. This new class (named `ConfigFileReader`) deserializes the XML data into objects as opposed to relying on `XPathNavigator` to parse and extract configuration data.

LISTING 11.27 The `ConfigFileReader` Class

```
Public Class ConfigFileReader

  Public Shared Function GetConnectionString(ByVal server _
    As String) As String
    Dim config As ServerConfig = GetConfig()
    'Check if Web server matches with dev server name or not
    Array.Sort(config.SqlServer.Dev.Servers)
    If Array.BinarySearch(config.SqlServer.Dev.Servers, _
      server) > -1 Then
      Return config.SqlServer.Dev.ConnString
    End If

    'Check if Web server matches with prod server name or not
    Array.Sort(config.SqlServer.Prod.Servers)
    If Array.BinarySearch(config.SqlServer.Prod.Servers, _
      server) > -1 Then
      Return config.SqlServer.Prod.ConnString
    End If

    'Default is Nothing
    Return Nothing
  End Function

  Public Shared Function GetConfig() As ServerConfig
    Dim context As HttpContext = HttpContext.Current
    'Check if config is already loaded into cache
    If context.Cache.Get("ServerConfig") Is Nothing Then
      Dim reader As XmlTextReader = Nothing
      Try
        Dim configPath As String = _
          context.Server.MapPath(_
          ConfigurationSettings.AppSettings("ServerConfig"))
        reader = New XmlTextReader(configPath)
        'Deserialize XML configuration data
        Dim s As New XmlSerializer(GetType(ServerConfig))
        Dim config As ServerConfig = CType(s.Deserialize(reader), _
          ServerConfig)
        'Create file dependency for cache
        Dim cd As New CacheDependency(New String() {configPath})
        context.Cache.Insert("ServerConfig", config, cd)
        Return config
      Catch
        Throw New ApplicationException("Unable to find config file.")
      Finally
```

LISTING 11.27 Continued

```
        If Not (reader Is Nothing) Then
          reader.Close()
        End If
      End Try
    Else
      Return CType(context.Cache.Get("ServerConfig"), ServerConfig)
    End If
  End Function

End Class
```

When you use the `ConfigFileReader` class, you can quickly and easily access configuration data by using object-oriented code rather than XPath statements:

```
Dim config As ServerConfig = ConfigFileReader.GetConfig()
Dim connStr As String = _
  ConfigFileReader.GetConnectionString(Request.UserHostName)
Me.lblOutput.Text = connStr + "<br />"
Me.lblOutput.Text += "SMTP Server: " + config.SmtpServer.Name + "<br />"
Me.lblOutput.Text += "Proxy Server: " + config.ProxyServer.Name
```

XML serialization really shines when structural or naming changes are made to the XML configuration document. Although you must run the `xsd.exe` utility against an updated version of the document's XSD schema, any code that contains invalid object references to old configuration classes or properties will be instantly identified by the compiler. This makes it easier to ensure that changes made in the code mirror configuration file changes.

Summary

This chapter focuses on several ways that you can use XML data in ASP.NET applications. XML's support for marking up data in a platform-neutral manner makes it an excellent choice for sending data between distributed systems and applications. XML is also useful for more basic duties, such as storing configuration settings.

The chapter discusses the pros and cons of the different XML APIs in the .NET Framework. Knowing when and where different APIs should be used is important to ensuring that an ASP.NET application is scalable and efficient. This is especially true as the size of XML documents increases.

This chapter also discusses techniques for combining functionality offered by the `XmlTextReader` and `XmlTextWriter` classes. Using these classes provides a fast and efficient way to parse and generate XML documents. You can also use the `XmlTextWriter` class to format XML data.

Additional topics discussed include the role of `XmlResolver` in accessing XML resource URIs. You can use `XmlResolver` instances to access secured XML resources, and they also play an important role in helping to determine whether specific functions such as `document()` are available to be used securely with XSLT stylesheets.

This chapter also provides information about searching and filtering XML data. As this chapter shows, you can use the XPath language or ADO.NET-specific classes such as `DataView` to perform similar tasks.

The remaining sections of this chapter discuss how to create a reusable XML validation component, shape the structure of XML data emitted from the `DataSet` class, and work with custom XML configuration files by using XPath and XML serialization.

PART IV

Hosting and Security

12

Side-by-Side Execution in ASP.NET

IN THIS CHAPTER

One of the major advantages envisioned for the .NET Framework when it was under development was the ability to run multiple versions concurrently on the same machine and allow applications to execute under whichever version is appropriate. In other words, in terms of ASP.NET, different Web applications, Web services, and Web sites can be running concurrently under different versions of the .NET Framework without interfering with each other.

This also means that components and other resources that are tied to one version of the .NET Framework can continue to be used with the appropriate applications, and installing updated versions will not interfere with existing ones. These components and other resources run side-by-side as separate processes and do not share any resources, assemblies, or other .NET Framework class files.

In Windows Server 2003 the new version of Internet Information Services (IIS)—version 6.0—provides a different core-processing model for Web applications and Web services. Although this isn't the topic of this chapter, it means that Windows Server 2003 can provide better performance, better process separation, and more robust management of errors and deadlocks.

Windows Server 2003 achieves these advantages through the use of a new kernel-level module called `http.sys`, which redirects incoming requests to the appropriate one of multiple separate instances of the Web service. It also handles output caching directly, providing another useful (and considerable) performance boost.

Version 1.1 of the .NET Framework fully supports side-by-side execution, allowing you to run both version 1.0 and version 1.1 of the .NET Framework on the same machine. You can also run the current beta version of the .NET Framework, code-named "Whidbey" and the forthcoming final version 2.0 alongside both version 1.0 and 1.1 installations. You can configure applications to run under any of the installed .NET Framework versions.

However, by default, all applications will run under the most recent version (the highest version number) of the .NET Framework. You have to configure applications and other resources (such as Web services) to force them to run under a specific version. This chapter looks at the following:

- How version 1.1 of the .NET Framework is distributed
- How installing a new version of the .NET Framework affects existing applications
- How ASP.NET selects the .NET Framework version to use at runtime
- How to specify the version that each application will run under
- Web service extensions and application pools in IIS 6.0

How Version 1.1 of the .NET Framework Is Distributed

ASP.NET 1.1 is installed by default on Windows Server 2003, which does not provide version 1.0. You can also install version 1.1 in three other ways:

- By installing Visual Studio .NET 2003. Version 1.0 of the .NET Framework is a prerequisite for this.
- By installing the version 1.1 redistributable file named `Dotnetfx.exe`, which you can download from the Microsoft Web site, at `http://msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx`. Alternatively, you can install it from the Windows Update site, using the link on your Start menu.
- By installing an application that itself contains the .NET Framework redistributable file. To check whether version 1.1 of the .NET Framework is already installed, you can select Start, Control Panel, Add/Remove Programs and then look for the entry "Microsoft .NET Framework [*language*] 1.1."

How Installing a New Version of the .NET Framework Affects Existing Applications

If you install a version of the .NET Framework that is compatible with and more recent than those already installed, the setup program will automatically update IIS so that all applications run under the new version. When you create new applications, they will also run under the new version of the .NET Framework by default—as long the new version is compatible. *Compatible*, in this case, is defined as having the same major version number (for example, versions 1.0 and 1.1 are compatible). However, the final release of version 2.0 will not be compatible with versions 1.0 and 1.1.

What happens when you remove a version of ASP.NET that is currently installed depends on the version you are removing and the other versions (if any) present on the server. If you remove (that is, uninstall) the latest version, all the applications that use this version are automatically converted to run under the next most recent compatible version that is installed. For example, if you remove version 1.1 and have version 1.0 installed, all applications will revert to running under version 1.0.

However, if you uninstall one of the versions that is not the most recent, the applications that run under it will be converted to run under the most recent compatible version. If there are no other compatible versions installed, ASP.NET pages will be served as simple text files. Therefore, it's important to ensure that existing applications are correctly mapped to the new version to prevent users from being able to view the source code of your ASP.NET pages and configuration files.

Configuration Settings in `machine.config`

One important point to note is that installing another version of the .NET Framework will install the default version of `machine.config` and the other policy and security files. If you have modified these files for the existing version and want the same configuration and policies to apply to the new version, you must copy the settings to the newly installed files.

The ASP.NET State Service and SQL Server State Service

If you configure the ASP.NET State Service or SQL Server State Service to handle session state (rather than the default in-process state storage mechanism), all state for all ASP.NET applications is managed by a single instance of the service. This approach is often used in Web farms or multiple server installations, and you may use a separate dedicated server just for this purpose.

When you run applications under different versions of the .NET Framework, all state is held in the same single instance of the ASP.NET State Service or SQL Server State Service—the one installed with the most recent version of the .NET Framework. If you uninstall a version of the .NET Framework, the previous most recent version is used instead.

The ASP.NET Process Account

ASP.NET pages and resources are executed under the context of an account named `ASPNET` by default. Unless you specify otherwise, all access for all pages and resources, regardless of the version of the .NET Framework they are running under, will use this single account. It will be, by default, the account installed by the most recent version of the .NET Framework. If you uninstall a version of the .NET Framework, the previous most recent version is used instead.

Windows Performance Counters

Each version of the .NET Framework installs its own pair of performance counters, named ASP.NET [*version*] and ASP.NET Apps [*version*]. The name of each of these counters contains the version number, which allows you to view the data for each version of the .NET Framework separately. However, the most recent version of the .NET Framework also installs counters that aggregate performance for ASP.NET over all versions of the .NET Framework that are installed. These two counters are named ASP.NET and ASP.NET Applications (see Figure 12.1).

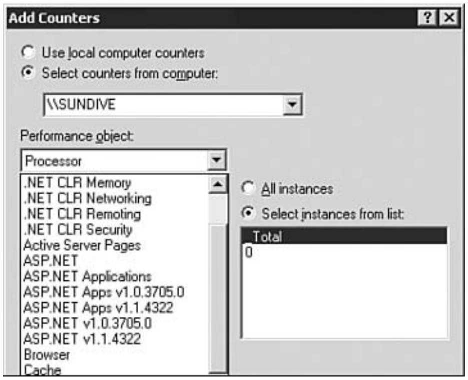


FIGURE 12.1 The ASP.NET performance counters, when multiple versions of ASP.NET are installed.

Running Version 1.0 Applications on Version 1.1 of the .NET Framework

In general, any application that is running on version 1.0 of the .NET Framework will run without modification on version 1.1. There are, however, five changes to the .NET Framework in version 1.1 that might affect your applications:

- Automatic input validation
- The `SelectedValue` property for ASP.NET list controls
- The ODBC provider for .NET
- Changes to forms authentication
- The Microsoft Mobile Internet Toolkit controls

Automatic Input Validation

A major addition to ASP.NET in version 1.1 of the .NET Framework is the implementation of a new feature that helps to reduce the risk of attacks that use cross-site scripting or SQL injection techniques being successful. By default, all input to a page within the `Request` collections (`QueryString`, `Form`, and `Cookies`) is checked against a hard-coded list of undocumented, but potentially dangerous, data strings.

If your existing version 1.0 pages depend on accepting this kind of data, they may fail to work correctly under version 1.1. However, you should always validate input to protect your pages against this type of attack, even in version 1.1, where there is some built-in protection.

The example in Listing 12.1 demonstrates the automatic validation feature. It provides a text box into which a value can be entered and a button to submit the form.

LISTING 12.1 An Example That Demonstrates Automatic Input Validation

```
<%@Page Language="VB" Debug="True" %>

<script runat="server">
  Sub ShowInput(Source As Object, E As EventArgs)
   lblResult.Text = txtTest.Text
  End Sub
</script>

<html>
<body>
<form runat="server">
   <asp:TextBox id="txtTest" runat="server" />
   <br/>
   <asp:button Text="Submit" onClick="ShowInput" runat="server" />
   <br/>
   <asp:Label id="lblResult" runat="server" />
</form>
</body>
</html>
```

When a potentially dangerous value, such as `<script>`, is submitted, an exception is raised and the standard ASP.NET error page is displayed (see Figure 12.2).

Of course, in an application, you'll probably want to trap this error and display a more suitable message or just ignore the input. You can experiment with this feature by turning off input validation. Automatic input validation is controlled by an addition to the `Page` directive in ASP.NET, a new addition to the `web.config` and `machine.config` files, and a new property of the `HttpRequest` class (which implements the `Request` object in ASP.NET), named `ValidateInput`.

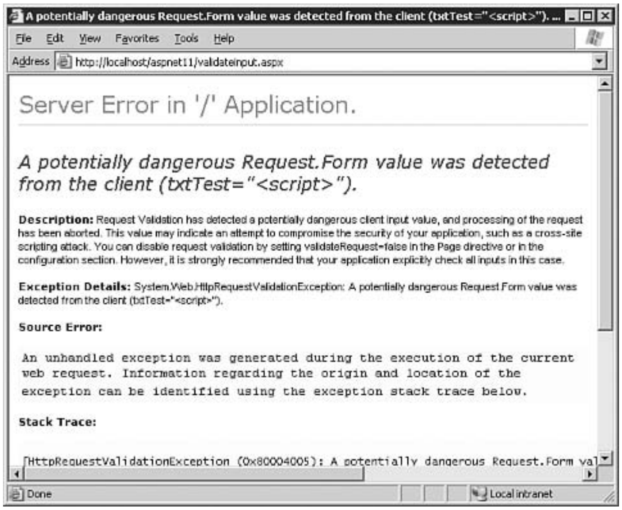


FIGURE 12.2
The resulting error page when potentially dangerous input is detected.

The `ValidateRequest` Page and `web.config` Directive

In version 1.1 of ASP.NET, adding an attribute to the `Page` directive allows you to turn off automatic input validation (the default, if this value is omitted, is `"true"`, and input validation is carried out):

```
<%@Page Language="VB" ValidateRequest="false" %>
```

You can also control input validation by adding an attribute to the `<pages>` element of the `web.config` file or changing the existing attribute in the `machine.config` file. This is the default `machine.config` file for version 1.1:

```
<pages buffer="true"
       enableSessionState="true"
       enableViewState="true"
       enableViewStateMac="true"
       autoEventWireup="true"
       validateRequest="true"
/>
```

If the input is invalid, an `HttpRequestValidationException` error is raised.

The `HttpRequest.ValidateInput` Method

If you disable automatic input validation in the `web.config` file or the `machine.config` file, you can still validate the input to a specific page by using the new `ValidateInput` method of the `HttpRequest` class:

```
Request.ValidateInput()
```

Again, if the input is invalid, an `HttpRequestValidationException` error is raised.

The `SelectedValue` Property for ASP.NET List Controls

The ASP.NET list controls expose several properties that you can use to extract the selected value(s) from them. For a `DropDownList`, `CheckBoxList`, `RadioButtonList`, or `ListBox` control, you can access the `SelectedIndex` property after a postback to get the index of the `ListItem` instance (within the `List` collection of the control) that was selected. If the list allows more than one item to be selected, this property returns the index of the first item selected in the list.

You can also access the `SelectedItem` property to get a reference to the first item selected in the control, and then you can query the `Text` or `Value` property of that `ListItem` object to get the currently selected text or value of the control.

In version 1.1 of ASP.NET, the `DropDownList`, `CheckBoxList`, `RadioButtonList`, and `ListBox` controls gain a new property, named `SelectedValue`. Following a postback, this property is automatically set to the value of the `Value` property for the first selected `ListItem` object in the list.

For example, the following code populates one of each of the four list controls that expose this property, and the button at the bottom of the page causes a postback during which the `SelectedValue` property of each control is extracted and displayed:

```
lblResult.Text = "DropDownList.SelectedValue = <b>'" _
               & MyDropDown.SelectedValue & "'</b><br />"
lblResult.Text &= "ListBox.SelectedValue = <b>'" _
               & MyListBox.SelectedValue & "'</b><br />"
lblResult.Text &= "CheckBoxList.SelectedValue = <b>'" _
               & MyCheckBoxList.SelectedValue & "'</b><br />"
lblResult.Text &= "RadioButtonList.SelectedValue = <b>'" _
               & MyRadioButtonList.SelectedValue & "'</b><br />"
```

Figure 12.3 shows the result of running this code. You can see that only the first selection in the `CheckBoxList` control is returned by the `SelectedValue` property.

When you have a list control that allows multiple selection—in other words, a `CheckBoxList` control or a `ListBox` control with the `SelectionMode="Multiple"` attribute—you still have to iterate through the `Items` collection, checking the `Selected` property of each `ListItem` object.

You can also use the `SelectedValue` property to select an item in these four list controls, by assigning the required `String` value to the property. In the sample page, the following code is executed when the Set to 'Sun' button is clicked:

```
MyDropDown.SelectedValue = "Sun"
MyListBox.SelectedValue = "Sun"
MyCheckBoxList.SelectedValue = "Sun"
MyRadioButtonList.SelectedValue = "Sun"
```

This sets the current selection to the entry for `"Sun"` in all the controls. If the value you specify is not in the list, an exception is raised.

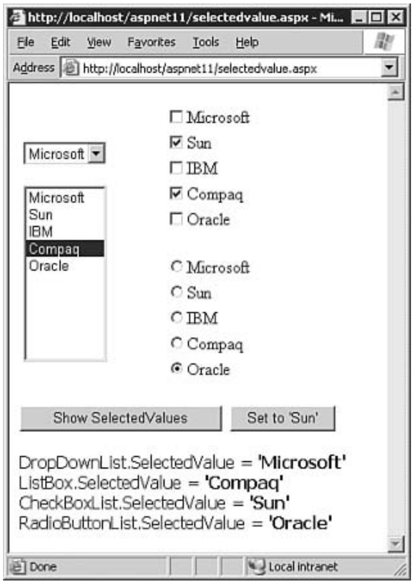


FIGURE 12.3 The `SelectedValue` property for the ASP.NET version 1.1 list controls.

`System.Data` Namespace Changes

In version 1.1 of the .NET Framework, there have been several changes in the classes from the `System.Data` namespace and its subsidiary namespaces. These are the classes that implement ADO.NET. The changes fall into several categories:

- Two new data-related namespaces that have been added to the .NET Framework. `System.Data.Odbc` implements the ODBC data provider (which was originally available in beta form for use with version 1.0). `System.Data.OracleClient` implements the .NET Framework data provider for Oracle. An important point to note is that the namespace name for the ODBC data provider has changed between version 1.0 and version 1.1 of the .NET Framework. The namespace for the (beta) version 1.0 is `Microsoft.Data.Odbc`, whereas for version 1.1 it is `System.Data.Odbc`. This means that you must change any `Import` directives that specify the namespace when you move your pages or components to version 1.1.
- The new property `HasRows`, which is added to the `DataReader` classes, returns `True` if there are one or more rows in the result set to which the `DataReader` instance is attached, following a call to the `Execute` method of the `Command` object that provides the result set. If the SQL statement or stored procedure executed by the `Command` object does not return any rows, the `HasRows` property returns `False`.
- A new method named `EnlistDistributedTransaction` for the `Connection` classes, which allows `Connection` instances to manually enlist into the current transaction if auto-enlist is disabled.
- Fixes for bugs or security issues in the existing classes. Some of these may affect your existing code. For example, see `www.daveandal.net/alshed/datasetkludges/default.asp` for details

on some of the changes to the workings of the DataSet class. For details of other changes between versions 1.0 and 1.1, see the GotDotNet pages, at www.gotdotnet.com/team/changeinfo/default.aspx.

Changes to Forms Authentication

When forms authentication is used, an encrypted cookie is stored on the client machine and sent with each request for a secured page. This encryption uses the value of the <machineKey> element within the <system.web> section of the machine.config file or the web.config file. The <machineKey> element also specifies the value used for encrypting and validating the viewstate in a page that contains a server-side <form> element.

In version 1.1 of the .NET Framework, by default, the validationKey and encryptionKey attribute values within the <machineKey> element contain a new modifier, named IsolateApps:

```
<machineKey validationKey="AutoGenerate,IsolateApps"
            decryptionKey="AutoGenerate,IsolateApps"
            validation="SHA1"/>
```

When this is present, the auto-generated keys include details of the ASP.NET application, so different applications running on the same machine will each generate different keys for securing their cookies or viewstate. This improves security and application isolation, especially where a server is hosting multiple sites or applications. In version 1.0 of the .NET Framework, where this modifier is not supported, the same key is used for all applications on the server.

This new behavior will cause a problem if you rely on shared authentication cookies, perhaps where you have a nested application (that is, an application within a subfolder of another application, with the path of the cookie set to / in the local web.config file) or if you are passing the viewstate from a page to a different application through a customized form post.

To retain the version 1.0 behavior when running under version 1.1 of the .NET Framework, you can do the following:

- Remove the IsolateApps modifiers from machine.config or (better) use a local web.config file that does not contain the IsolateApps modifiers.
- Change the validationKey and decryptionKey attribute values to specify an explicit key rather than auto-generating it. If you are using a Web farm or another shared server setup, you will be using a specific key that is the same on all the servers anyway.

There are also two new properties for the FormsAuthentication class—RequireSSL and SlidingExpiration. When the RequireSSL property is True, all requests must be made under the secure HTTPS protocol rather than the more usual HTTP.

The SlidingExpiration property specifies whether the timeout for forms authentication (as specified in the machine.config or web.config file) is absolute, or starts again on each request. In other words, when the SlidingExpiration property is True, the timer effectively restarts on each request. When it is False, authentication expires after the prescribed period, regardless of how many requests the user has made.

The MMIT Mobile Controls

In version 1.1 of the .NET Framework, the ASP.NET mobile controls from the MMIT are integrated into the class library and can be used directly, without requiring a separate installation. The two namespaces `System.Web.Mobile` (the core classes and authentication and error-handling features) and `System.Web.UI.MobileControls` (the controls themselves) are now an integral part of the .NET Framework. There is also a namespace `System.Web.UI.MobileControls.Adapters`, which contains the core control adapter classes that you can use to build you own mobile controls.

By default, ASP.NET does not create pages that are suitable for use with the mobile controls, and you still have to add the same "extra information" to the page to use these controls. This involves specifying that the page itself should be an instance of the `MobilePage` type, which allows multiple forms to exist on a page and provides integration with the core mobile capabilities:

```
<%@Page Inherits="System.Web.UI.MobileControls.MobilePage" Language="VB"%>
```

You must also continue to specify the tag prefix and the assembly that contains the mobile controls by using a `Register` directive, so that the controls can be identified. The usual prefix is `"mobile"`, as in this example:

```
<%@Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls"
            Assembly="System.Web.Mobile"%>
```

This means that existing version 1.0 pages that use the MMIT will function just the same on version 1.1, with no changes required to the code except where it uses other classes (for example, classes from the `System.Data` namespaces) that have changed in version 1.1.

Running Version 1.1 Applications on Version 1.0

If you write an application to run on version 1.1 of the .NET Framework and avoid using any features that are new or changed in version 1.1, you will be able to run that application on version 1.0. However, unless you are strictly limited to using only version 1.0 on the server that will host the application, you should consider always running on the latest version of the .NET Framework to benefit from the latest security fixes and performance enhancements.

How ASP.NET Selects the Runtime Version

IIS uses the concept of *mappings* (sometimes called *script mappings* or *application mappings*) to decide how to process a file or resource when it is requested through the WWW Service. You can view and change the mappings for a Web site or a virtual Web application in the Mappings tab of the Application Configuration dialog for a Web site. To open the Application Configuration dialog, you open the Properties dialog for the Web site, select the Home Directory tab, and click the Configuration button (see Figure 12.4).

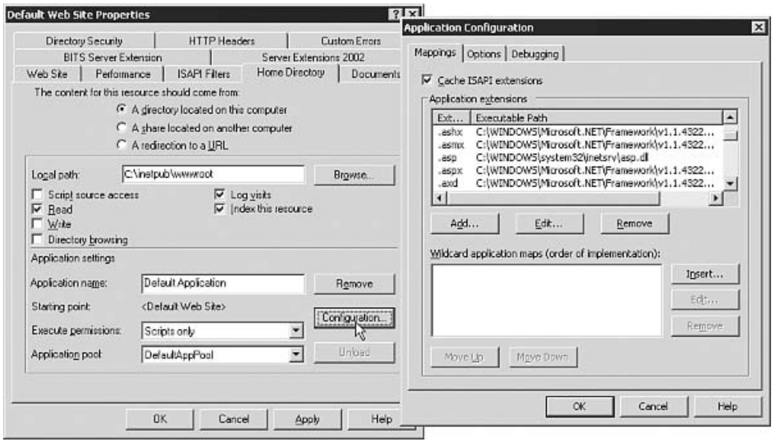


FIGURE 12.4
Viewing the script mappings in Internet Information Services Manager.

The mappings for ASP.NET pages and resources point to the file `aspnet_isapi.dll`, which is responsible for processing these pages and resources. If you have more than one version of the .NET Framework installed, the mapping will point to the version of `aspnet_isapi.dll` that will be used, and this determines which version of the .NET Framework classes and ASP.NET runtime will process the resources. In Figure 12.4, you can see that version 1.1 will be used (the full version number is 1.1.4322).

How to Specify the ASP.NET Version for Individual Applications

As you have seen in the preceding section, all you have to do to force ASP.NET resources to be executed under a different version of the .NET Framework is change the mapping to point to `aspnet_isapi.dll` in the appropriate `[`*`version`*`]` folder of the .NET Framework. One way to do this is to manually edit the entries; however, you have to repeat this process for several file types (all the extensions for ASP.NET, such as `.aspx`, `.asmx`, `.asax`, and `.ascx`).

A far easier way to force ASP.NET resources to be executed under a different version of the .NET Framework is to use the `aspnet_regiis.exe` application registration utility that is provided with every version of the .NET Framework. This utility can be used for several tasks related to script mappings in IIS, including updating the mappings for some or all of the Web sites and Web applications configured within IIS.

Installing ASP.NET Without Updating Script Mappings

The `Dotnetfx.exe` setup program executes the `aspnet_regiis.exe` utility automatically when you install the .NET Framework and when you uninstall it. However, you can prevent `aspnet_regiis.exe` from being executed, and hence maintain the existing script mappings, by running the Dotnetfx.exe setup program from a command window and specifying the special parameter sequence, as shown here:

```
Dotnetfx.exe /c:"install /noaspupgrade"
```

This means that you can install the latest version of ASP.NET without disturbing any existing applications and then update individual applications as and when required by using the aspnet_regiis.exe utility. When you create a new Web application, the version currently set up for the default Web site within which the new application is created is used for the new application until you specifically change it. Again, you can use the aspnet_regiis.exe utility for this.

Remember that if the version of ASP.NET you are installing is older than the most recent version already installed, the setup program does not automatically execute aspnet_regiis.exe—and so the existing script mappings are not updated.

Using the aspnet_regiis.exe Tool to Configure Runtime Versions

The aspnet_regiis.exe tool is supplied with each version of the .NET Framework and is located in the %windir%/Microsoft.NET/Framework/[*version*]/ folder. The version of the tool is different for each version of the .NET Framework, so you must use the correct one, depending on what configuration changes you want to make. For example, to configure an application to use version 1.0 of the .NET Framework, you must run the version of aspnet_regiis.exe from the folder %windir%/Microsoft.NET/Framework/v1.0.3705/.

You run the aspnet_regiis.exe utility from a command window. As shown in Table 12.1, aspnet_regiis.exe accepts a range of parameters that determine the configuration changes it makes. Note that you can use this tool to create the aspnet_client folder for your Web sites and populate it with the required client-side script files, and you can also use it to set the script mappings or display information about the versions of ASP.NET that are installed.

In Windows Server 2003, with IIS 6.0, you must also manage the Web service extensions to allow ASP.NET to serve pages. You'll learn more on this topic later, but you can see in Table 12.1 that the aspnet_regiis.exe utility can set these for you as well.

TABLE 12.1

The Command-Line Parameters for the aspnet_regiis.exe Utility

Parameter	Description
-i	Registers this version of ASP.NET, adds the matching Web service extension to IIS 6.0, and updates the mappings for all Web sites and Web applications to point to this version of aspnet_isapi.dll.
-ir	Registers this version of ASP.NET but does not update Web site and Web application mappings.
-enable	Is used with the -i or -ir parameters to set the status to Allowed for the Web service extension it installs for ASP.NET (version 1.1 and above with IIS 6.0 and above only).
-s <*path*>	Updates the mappings for all Web sites and Web applications at the specified path and updates any applications nested within this path to point to this version of aspnet_isapi.dll (for example, aspnet_regiis.exe -s W3SVC/1/ROOT/ProAspNet).
-sn <*path*>	Updates the mappings for all Web sites and Web applications at the specified path, but not those nested within this path, to point to this version of aspnet_isapi.dll.

TABLE 12.1
Continued

Parameter	Description
`-r`	Updates the mappings for all Web sites and Web applications configured within IIS to point to this version of `aspnet_isapi.dll`. Does not register this version of ASP.NET or add a Web service extension.
`-u`	Unregisters this version of ASP.NET and removes the Web service extension. Any existing mappings for this version are remapped to the highest remaining version of ASP.NET that is installed on the machine.
`-ua`	Unregisters all versions of ASP.NET on the machine.
`-k` *<path>*	Removes all mappings to all versions of ASP.NET for all Web sites and Web applications at the specified path and any applications nested within this path (for example, `aspnet_regiis.exe -k W3SVC/1/ROOT/ProAspNet`).
`-kn` *<path>*	Removes all mappings to all versions of ASP.NET from the specified path but does not remove those nested within this path.
`-lv`	Lists all versions of ASP.NET that are installed on the machine, along with the current status (`Valid` or `Invalid`) and path to `aspnet_isapi.dll` for that version (when the status is `Valid`).
`-lk`	Lists the paths of all the IIS metabase keys that contain ASP.NET mappings, together with the version each one is mapped to. Does not include any keys that inherit ASP.NET mappings from a parent key.
`-c`	Installs the client-side scripts for this version into the `aspnet_client` subfolder of every IIS Web site directory.
`-e`	Removes the client-side scripts for this version from the `aspnet_client` subfolder of every IIS Web site directory.
`-ea`	Removes the client-side scripts for all versions of ASP.NET from the `aspnet_client` subfolder of every IIS Web site directory.
-?	Prints the help text in the command window.

One issue to be aware of is that installing the .NET Framework adds to your `PATH` environment variable the path to the utilities folder. Therefore, depending on the order in which you installed the .NET Framework versions, you might find that typing just `aspnet_regiis` will not run the version you expect or require. To get around this, you need to enter the full path to the version of `aspnet_regiis.exe` that you want or edit your `PATH` environment variable to change the order of the paths or add the one you need.

To edit your `PATH` environment variable, you open the System applet by selecting Start, Settings, Control Panel; then you click the Environment Variables button in the Advanced tab of the System Properties dialog.

Listing Versions, Web Sites, and Application Roots

As an example of using `aspnet_regiis`, the following command uses the `-lv` (list versions) parameter to list the versions of the .NET Framework that are installed on the machine by printing the path to the `aspnet_isapi.dll` file for each version and showing which is the default (root) entry in IIS:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>aspnet_regiis -lv
1.0.3705.0  Valid
```

```
➥ C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\aspnet_isapi.dll
1.1.4322.0  Valid (Root)
➥ C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\aspnet_isapi.dll
```

To get a list of the Web sites and virtual Web applications, together with the version that each one is currently mapped to, you can use the `-lk` (list keys) parameter:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>aspnet_regiis -lk
W3SVC/                   1.1.4322.0
W3SVC/1/ROOT/            1.1.4322.0
W3SVC/1/ROOT/MSMQ/       1.1.4322.0
W3SVC/1/ROOT/Printers/   1.1.4322.0
W3SVC/1/ROOT/ASPNETInsiders/  1.1.4322.0
```

Updating the ASP.NET Runtime Configuration

To demonstrate how to change the mappings for Web sites and Web applications, the following command shows how you can use the `-s` (script-map) parameter (the path can be obtained using the `-lk` parameter as shown in the preceding section):

```
C:\WINDOWS\...\v1.0.3705>aspnet_regiis -s W3SVC/1/ROOT/ASPNETInsiders
Start installing ASP.NET DLL (1.0.3705.0)
➥ recursively at W3SVC/1/ROOT/ASPNETInsiders
Finished installing ASP.NET DLL (1.0.3705.0)
➥ recursively at W3SVC/1/ROOT/ASPNETInsiders
```

Now the mappings for the virtual application root named `ASPNETInsiders` and all nested virtual applications are configured so that they will execute under version 1.0 of the .NET Framework. One point to watch here is that because IIS 6.0 was not available when version 1.0 of the .NET Framework was created, the `aspnet_regiis` tool does not install ASP.NET 1.0 in the Web service extensions section of IIS 6.0. You have to create this entry manually (as shown in the following section) and set the status to `Allowed`.

Installing the ASP.NET Client-Side Script Folder

When you create a new Web site, the `aspnet_client` subfolder that contains the client-side scripts required by some ASP.NET server controls is not automatically added to that Web site. You can ensure that it is present and correctly populated with the required scripts for all Web sites by using the `-c` option of `aspnet_regiis.exe`:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>aspnet_regiis -c
```

ASP.NET and IIS 6.0 on Windows Server 2003

IIS 6.0 on Windows Server 2003 contains a new extra layer of security for the Web service, in the form of Web service extensions. Basically, *Web service extensions* are subsets of the script mappings that are installed on the machine, with the option to block requests for files that have the file extension specified in that mapping set.

You have to ensure that the status for the Web service extension that specifies the version of ASP.NET you are using for your applications is set to `Allowed`. If it isn't, the client will simply receive a "Page not found" response—even though the page exists and the user has requested the correct URL.

IIS 6.0 Web Service Extensions

To configure Web service extension settings in IIS 6.0, you open Internet Information Services Manager and select the Web Service Extensions folder. You can see in Figure 12.5 that the Web service extension for version 1.1 of ASP.NET is configured within the list and has its status set to `Allowed` so that it can handle requests. This is because this machine was specified as an application server when the Windows Server 2003 operating system was installed.

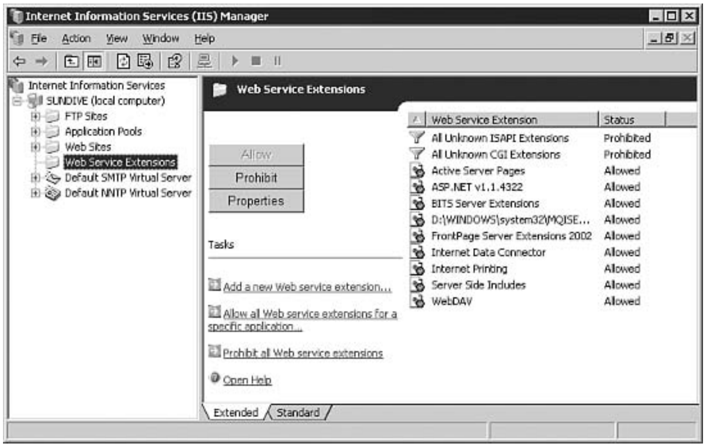


FIGURE 12.5
Managing the Web service extensions in IIS 6.0 on Windows Server 2003.

To add a new extension for a different version of the .NET Framework, you click the Add a New Web Service Extension link. Then you type the name of the extension in the New Web Service Extension dialog, check the option Set Extension Status to Allowed, and click the Add button. In the Add File dialog that appears, you navigate to the appropriate .NET Framework version folder and select the `aspnet_isapi.dll` file (see Figure 12.6).

After you click OK twice, the new Web service extension appears in the list. Now any ASP.NET pages or resources that are configured to use this version of the .NET Framework—in other words, applications that specify this version of `aspnet_isapi.dll` in their script mappings—will run (see Figure 12.7).

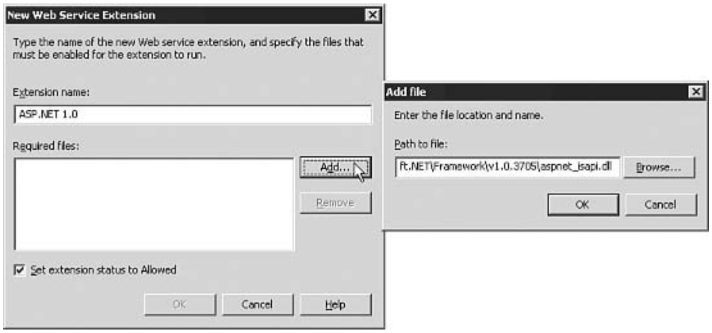


FIGURE 12.6
Adding a new Web service extension.

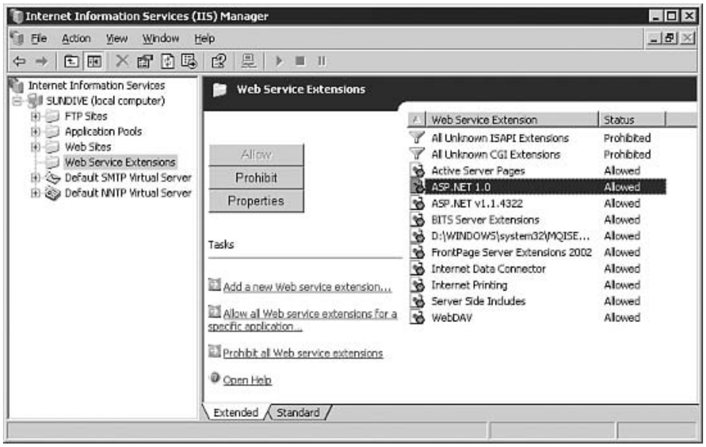


FIGURE 12.7
A new Web service extension in Internet Information Services Manager.

IIS 6.0 Application Pools

If you try to run ASP.NET applications that are configured to use different versions of the .NET Framework on the same machine under Windows 2003 and IIS 6.0, you must either segregate them by version in different application pools or disable application pooling altogether and run in IIS 5.0 isolation mode (described later in this chapter, in the section "Using IIS 5.0 Isolation Mode in IIS 6.0"). By default, IIS 6.0 uses a common process for all the applications running in the same application pool. If applications in the same application pool try to use different versions of ASP.NET, you'll see the Server Application Unavailable page and the error message shown in Figure 12.8 appears in the Application section of the event log.

To get around this, you can create a new application pool and then assign the applications that require different versions of the .NET Framework to different pools. You can run all the applications that use the same version of the .NET Framework in the same application pool, or you can create multiple application pools and allocate your applications between them.

FIGURE 12.8
The error messages when multiple versions of ASP.NET are not configured in separate application pools.

Creating a New Application Pool

To create a new application pool, you right-click the Application Pools folder in Internet Information Services Manager and select New; then you select Application Pool. Next, you enter the name for the new application pool in the Add New Application Pool dialog that appears, and you select the first option button to use the default settings. Alternatively, if you have created a template for application pools, you can base the new one on that by selecting the second option button (see Figure 12.9).

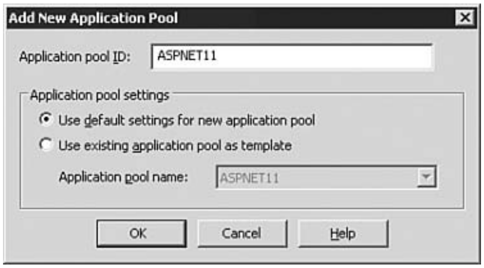


FIGURE 12.9
Creating a new application pool in IIS 6.0.

Allocating ASP.NET Applications to an Application Pool

To assign a Web site or virtual Web application to an existing application pool, you just have to select it in the Properties dialog for the site or application. In the Home Directory tab or the Virtual Directory tab of the Properties dialog, you use the drop-down Application Pool list at the bottom of the dialog to specify which application pool you require (see Figure 12.10).

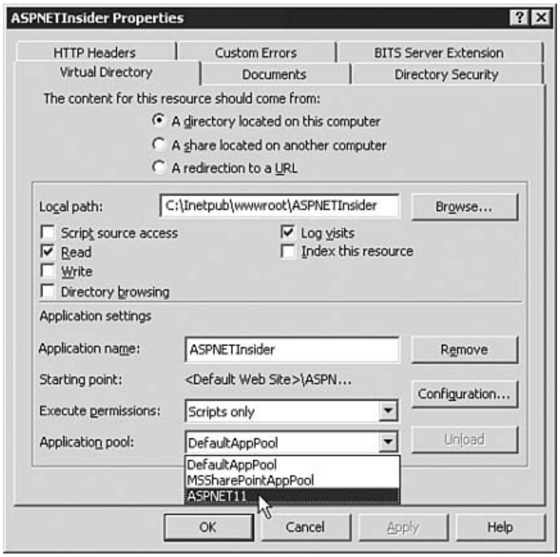


FIGURE 12.10
Selecting the application pool for an ASP.NET Web application.

Using IIS 5.0 Isolation Mode in IIS 6.0

You can configure IIS 6.0 to run in IIS 5.0 isolation mode. In this mode, the application-pooling feature that is turned on by default in IIS 6.0 is disabled, and applications run under the same process isolation model as in IIS 5.0. If you enable IIS 5.0 isolation mode, you can run ASP.NET applications that execute under different versions of the .NET Framework without having to create separate application pools.

To enable IIS 5.0 isolation mode, you open the Properties dialog for the Web Sites folder and check the Run WWW Service in IIS 5.0 Isolation Mode option (see Figure 12.11). When you close the Properties dialog, IIS prompts you to restart the service to put the new setting into effect.

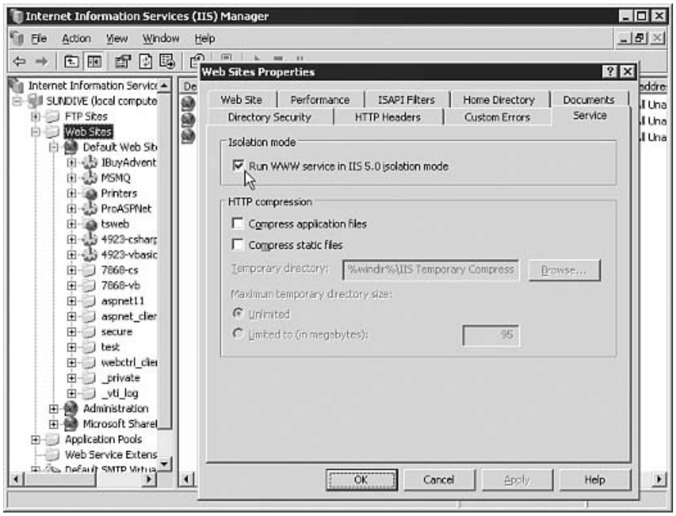


FIGURE 12.11
Specifying IIS 5.0 isolation mode in IIS 6.0.

However, in IIS 5.0 isolation mode you do not benefit from many of the improvements in IIS 6.0, including better process management and deadlock detection. You should avoid using IIS 5.0 isolation mode unless it is absolutely necessary.

Summary

This chapter looks at how the .NET Framework allows you to run multiple versions side-by-side and select which version each application should run under. This is a huge advance over previous versions of ASP, where you had to perform a full server upgrade and shift all your applications to the newly installed version.

Along with the fundamental changes that the .NET Framework provides, such as freedom from reliance on COM components and "DLL hell," ASP.NET side-by-side execution also solves many issues you had to cope with in the past. In particular, running and testing different versions of your Web sites and Web applications are now much easier and much more controllable. You can move an application from one version of the .NET Framework to another quickly and easily.

As well as side-by-side execution, this chapter also looks at the changes to the namespaces in the .NET Framework that are relevant to ASP.NET and Web applications. There are many minor changes between versions 1.0 and 1.1, and there are quite a lot of bug fixes, but only a few of these affect applications when you migrate from one version to another. This chapter summarizes the changes that are most likely to affect your applications and how you can get around the issues these changes raise.

Finally, this chapter looks at the latest version of the Windows operating system, Windows Server 2003, and the way it affects ASP.NET applications. The better performance and robustness of IIS version 6.0 certainly make it worth considering an upgrade to Windows Server 2003.

13

Taking Advantage of Forms Authentication

Using forms authentication is a great way to create ASP.NET applications that require users to sign in to perform certain operations. The features provided by forms authentication make it quick and easy to create a secure authentication system and to make checks against that system in code.

Sometimes, though, you want an authentication system that you have built on forms authentication to do things that the basic forms authentication implementation does not. Fortunately, the ASP.NET developers at Microsoft anticipated this and built the entire system in a way that makes it easy to customize to your particular needs.

This chapter looks at lots of situations in which you need to use forms authentication in ways that are different from the standard approach.

This chapter assumes that you are already familiar with the basics of forms authentication, setting up the `web.config` file, and creating a sign-in form. If you have not used forms authentication before at all, it would

IN THIS CHAPTER

be worth working through a basic example (there are loads available online and in other books) before reading this chapter.

One basic piece of advice that you should keep in mind while reading this chapter is to use SSL for your sign-in page. Forms authentication protects the authentication ticket that is used to identify signed-in users by encrypting it and testing for tampering, but that is useless if you allow your users' passwords to be stolen by having them submitted to your sign-in page in plain text.

The Internet Information Services online help (available by browsing to `http://localhost/iishelp` on a default installation of IIS) includes details of how to get a server certificate and then use it to set up SSL.

Note that provided that your users are not entering or viewing any confidential information through your application, you only need to protect the sign-in page with SSL. Once the user has signed in, the encryption provided by ASP.NET by default will protect the user's subsequent requests. Of course, if you include a sign-in control on several pages, you will need to protect all those pages.

Building a Reusable Sign-in Control

The standard way to do sign-in in ASP.NET applications that use forms authentication is to provide a sign-in Web form to which users are redirected when they attempt to access a page that they are not authorized to view (based on the settings in the `<Authorization>` section of the configuration file). However, many Web applications do not divide features for authenticated and anonymous (non-authenticated) users into separate Web forms; instead, they display additional features for authenticated users on the same Web forms that all users see. For example, a forum application might allow all users to view posts but allow only authenticated users to reply to posts or start new threads.

In situations like this, it makes a lot of sense to include sign-in controls as part of the overall page structure of the application. This section shows an example of a user control you can build to show sign-in controls for anonymous users and other controls for authenticated users. This example simply shows a welcome message and a sign-out link, but you could use the ideas presented in this example for all sorts of application-specific options.

When the user is not signed in, the control looks as shown in Figure 13.1. When the user is signed in, the control looks as shown in Figure 13.2.

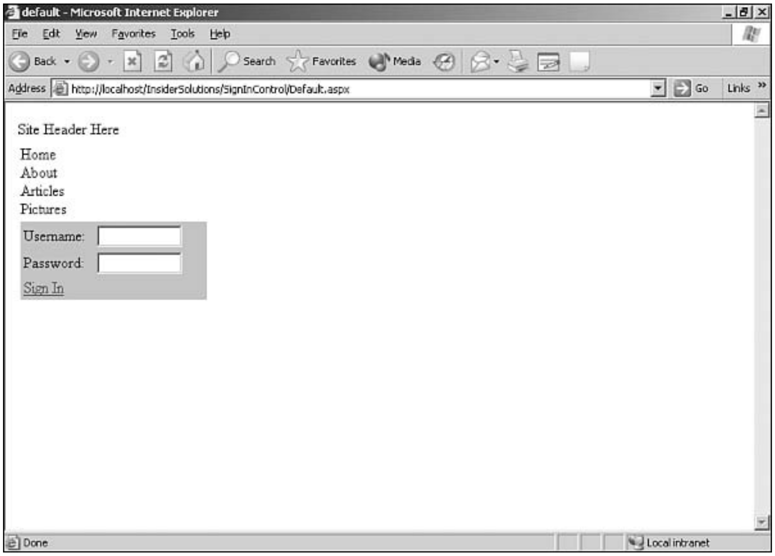


FIGURE 13.1
A sample sign-in control, when the user is not signed in.

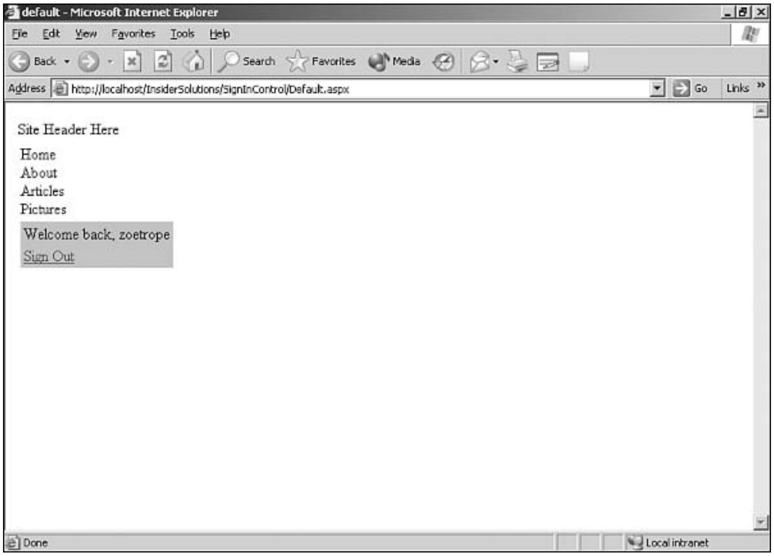


FIGURE 13.2
A sample sign-in user control, when the user is signed in.

Listing 13.1 shows the code for the .ascx file of a simple sign-in control.

LISTING 13.1 .ascx Code for the Sample Sign-in Control

```
<%@ Control Language="vb" AutoEventWireup="false"
    Codebehind="SignIn.ascx.vb" Inherits="SignInControl.SignIn"
    TargetSchema="http://schemas.microsoft.com/intellisense/ie5" %>
<table id="AnonymousControls" width="100%" runat="server">
  <tr>
    <td style="WIDTH: 73px">Username:
    </td>
    <td>
      <asp:textbox id="UsernameTextBox" runat="server" Width="88px" />
      <asp:regularexpressionvalidator id="UsernameValidator"
                                      runat="server"
                                      Display="None"
                                      ControlToValidate="UsernameTextBox"
                                      ValidationExpression="[a-z¦A-Z¦0-9¦]{5,20}"
                                      />
    </td>
  </tr>
  <tr>
    <td style="WIDTH: 73px">Password:
    </td>
    <td>
    <asp:textbox id="PasswordTextBox" runat="server" Width="88px" TextMode="Password" />
    <asp:regularexpressionvalidator id="PasswordValidator"
                                    runat="server"
                                    Display="None"
                                    ControlToValidate="PasswordTextBox"
                                    ValidationExpression="[a-z¦A-Z¦0-9¦]{5,20}"
                                    />
    </td>
  </tr>
  <tr>
    <td colSpan="2">
      <asp:linkbutton id="SignInButton"
                      runat="server"
                      CausesValidation="False">
         Sign In
      </asp:linkbutton>
   </td>
  </tr>
```

LISTING 13.1 Continued

```
</table>
<table id="AuthenticatedControls" runat="server">
  <tr>
    <td>
      Welcome back,
      <asp:label id="UsernameLabel" runat="server">[username]</asp:label>
    </td>
  </tr>
  <tr>
    <td>
      <asp:LinkButton id="SignOutButton" runat="server"
                                   CausesValidation="False">
        Sign Out
      </asp:LinkButton>
    </td>
  </tr>
</table>
```

The control is composed of two <table> elements, which are set to runat="server" so that you can make them visible or invisible, depending on whether the user is signed in.

Standard <table> elements are used rather than <asp:Table> controls because server-side access is only required in order to set the visibility. Using the Web control table would mean creating server-side table row and table cell controls and would require extra overhead.

Note that you include RegularExpressionValidator controls for both the username and the password input controls. In both cases, you set up the regular expression to accept only alphanumeric characters and require the input to consist of between 5 and 20 characters.

The regular expression used here, [a-z¦A-Z¦0-9¦]{5,20}, has a group (marked by []) which will match to a character that falls into any of the three ranges defined within it, followed by the minimum and maximum number of characters (marked by {}). If you wanted to allow any number of characters, you would replace the {5,20} with *.

The CausesValidation attribute of each LinkButton control is set to False. This might seem strange, considering that you have included validators, but it will become clear shortly.

RegularExpressionValidator as a Validation Tool

If you are not familiar with regular expression syntax, you really should learn it. RegularExpressionValidator is an excellent validation tool, and it is just the tip of the iceberg for using regular expressions—they are great for all kinds of text matching and processing tasks.

There is lots of information in the .NET Framework documentation. For some reason, the JScript .NET section of the documentation has a particularly good guide to the syntax and usage of this powerful pseudo-language. A search for "regular expressions" will provide links to all the relevant sections.

BEST PRACTICE

Validating User Input

You should always validate users' input to your application to ensure that it contains what you expect it to contain. Getting into the habit of validating every input is a great way to prevent problems due to unexpected inputs.

A couple common attacks are made against Web applications that are best prevented through validation of all input. Script injection (the addition of malicious JavaScript code in an attempt to get it displayed by the application and thus run by your visitors' browsers) is stopped dead by the prevention of the characters it needs from being entered. Similarly, SQL injection, where malicious SQL code is entered in an attempt to have your database execute it, is prevented by good validation.

Both script injection and SQL injection can be prevented in other ways (indeed, ASP.NET now has a default defense against the inputting of harmful code), but it is always wise to *defend in depth*—that is, to protect your application at every stage rather that rely on a single defense.

Good validation across the board has other advantages, too. Providing users with feedback on what they are doing wrong is a great way to help them with any difficulties they may have.

The code-behind file shown in Listing 13.1 includes declarations for the two server-side `<table>` elements that are used:

```
Public Class SignIn
  Inherits System.Web.UI.UserControl
  Protected WithEvents AnonymousControls As System.Web.UI.HtmlControls.HtmlTable
  Protected WithEvents AuthenticatedControls As System.Web.UI.HtmlControls.HtmlTable
```

The control is initialized with a simple `Page_Load` event handler:

```
Private Sub Page_Load(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
        Handles MyBase.Load
  'check whether the user is authenticated
  If Request.IsAuthenticated Then
    'the user is authenticated, so display the authenticated controls
    AnonymousControls.Visible = False
    AuthenticatedControls.Visible = True

    'populate the username display
    UsernameLabel.Text = Context.User.Identity.Name
  Else
    'the user is not authenticated, so display the anonymous controls
    AnonymousControls.Visible = True
    AuthenticatedControls.Visible = False
  End If
End Sub
```

The interesting stuff happens in the event handler for the `Click` event of the `SignInLinkButton` control:

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

    Dim valid As Boolean = True
    Dim c As Control
    Dim v As BaseValidator

    'loop through all validators on the page
    For Each v In Page.Validators
      'check whether the validator is attached to this user control
      If Not Me.FindControl(v.ControlToValidate) Is Nothing Then
        'validate the control
        v.Validate()
        'check whether the control validated successfully
        If Not v.IsValid Then
          Response.Write(v.ID)
          'if it did not validate, set valid to false
          valid = False
        End If
      End If
    Next

    'only proceed with sign in if the controls on this user control are valid
    If valid Then
      'authenticate the user against the credentials stored in the web.config
      'if you use a different credentials store, check against that here
      If FormsAuthentication.Authenticate(UsernameTextBox.Text, _
                                          PasswordTextBox.Text) Then
        'set the authentication cookie
        FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, False)

        'refresh the page
        Response.Redirect(Request.Url.PathAndQuery)
      End If
    End If

  End Sub
```

Note that this code assumes that the `System.Web.Security` namespace has been specified using an `Imports` statement (in C#) at the top of the code file.

The first part of this code performs validation for all the validators that are attached to controls that are in this user control. This is why the `CausesValidation` property of the `SignInButton` control was set to `False`: You are calling the `Validate` methods of the validators rather than having ASP.NET do it automatically when the `LinkButton` controls are clicked.

You call the `Validate` methods of the validators because you do not want the sign-in control to be affected by the validation states of controls that are outside the user control. If you used the standard approach, a failed validation anywhere on the page would prevent the sign-in control from signing the user in, even if the username and password `TextBox` controls were valid. This is a problem for any user control that you want to operate independently of other parts of the page because ASP.NET groups all validators into a single collection under the `Page` object.

You could explicitly call the `Validate` methods on the two validators, but we thought it would be worth showing some general code that can be added to any user control to perform limited validation for the controls it contains. This approach will have a very slight performance implication, but it also means that any changes to the validation controls will be automatically reflected in the validation code.

After performing validation, you check the `valid` variable to ensure that no validators failed validation and, if everything is fine, you check the user's credentials. For simplicity, the standard `web.config` file credentials store is used in this example, but you can insert your own credentials check code to check against whatever store you like.

If the credentials are okay, you set the authentication cookie with the following code:

```
FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, False)
```

At this point, this code differs from the standard forms authentication login page code. Rather than use the `FormsAuthentication.RedirectFromLoginPage` method, it uses the `SetAuthCookie` method, which sets the authentication cookie but does not do a redirection.

You want to refresh the page after setting the cookie, so you redirect the user back to the same page and query string:

```
Response.Redirect(Request.Url.PathAndQuery)
```

Hashing Passwords

These days, most decent applications do not store their users' passwords as plain text. You have to assume that because nothing is 100% secure, there is a chance that an application will be compromised and the credentials, however they are stored, may be stolen.

In a small application, this might not be a huge problem in comparison to other issues that arise when security is breached; the users' passwords can be reset in order to render the stolen passwords useless. But imagine trying to do this for an application with more than a handful of users—it would be a nightmare!

There is way to mitigate the risk of passwords being stolen. By using a technique called *hashing*, you can store encrypted passwords, rather than plain-text passwords, in your credentials store. Hashing is also known as *one-way encryption* because after you have created a hash from a password, it is not practical to work back the other way and recover the password. If someone steals the hashed passwords, they will be of no use in further compromising the system.

Another advantage of using hashed passwords is that, with the passwords hashed, it is a lot harder for an administrator to pretend to be another user; he or she cannot simply read the password from the database and use it to sign in. This helps to ensure that actions apparently carried out by a particular user really were done by that user.

Forms authentication has support for password hashing built in, through the `FormsAuthentication.HashPasswordForStoringInPasswordFile` method and the `passwordFormat` attribute of the `<credentials>` section of the `web.config` file.

In order to use hashed credentials in the `web.config` file, you need a way to generate the hashes. The following is the button click event from the code-behind file for a simple Web form that has a text box, a button, and two labels on it to accept a password and generate hashes in the two formats that ASP.NET can use:

```
Private Sub GenerateHashes_Click(ByVal sender As System.Object, _
                                 ByVal e As System.EventArgs) _
            Handles GenerateHashes.Click
  MD5Label.Text = "MD5: " + _
    FormsAuthentication.HashPasswordForStoringInConfigFile _
      (PasswordTextBox.Text, "MD5")
  SHA1Label.Text = "SHA1: " + _
    FormsAuthentication.HashPasswordForStoringInConfigFile _
      (PasswordTextBox.Text, "SHA1")
End Sub
```

Note that if you do not use the Visual Studio .NET designer to create the form, you need to add declarations to the code-behind file for the `GenerateHashes` control (a `Button` control) and the `MD5Label` and `SHA1Label` controls (both `Label` controls).

When you have a hashed password, you simply need to include it in the `web.config` file's `<credentials>` section and set the `passwordFormat` attribute. The following example uses an SHA1 hash:

```
<credentials passwordFormat="SHA1">
  <user name="zoetrope" password="C983A1F054842D9220847ED5628E7038887138A7" />
</credentials>
```

With this hash in place, the `FormsAuthentication.Authenticate` method will now automatically hash the password the user has entered before comparing it to the value stored in the configuration file.

Remember, this hashing will only protect the password while it is stored on the server; it will not in any way protect the password as it is being transferred from the user's browser to the server that the application runs on. In order to be secure, you really need to use SSL to protect sign-in.

`web.config` is not very often used as the credentials store in serious applications. It is just not designed to hold application-updatable data. It is much more common to use a separate XML file or, more commonly, a database to hold the credentials.

If you want to use a different credentials store, you need to use the `HashPasswordForStroringInConfigFile` method to hash passwords when they are set by users and to hash the password that a user enters when he or she signs in before comparing it to the stored hash in the credentials store.

Helping Users Who Forget Their Passwords

There is one big problem with password hashing. As mentioned in the previous section, hashing is a one-way operation; after you have created a hash, it is not practical to return to the plaintext password. This causes a problem if a user forgets his or her password: How can you tell the user what his or her password is? The answer is that you cannot, but there are other ways in which you can help them.

We could provide a "forgot my password" page in the application that provides an option to reset the password to a random value and email it to the user's registered email address. The problem with this is that malicious users could continually reset other users' passwords, causing them a lot of inconvenience.

Another possibility is to store the answer to a secret question that must be answered in order to reset the password. The problem with this is that users who forget their password are also liable to forget the answers to their secret questions (unless they make the answers really obvious, in which case they will be insecure).

A good solution is to provide a "forgot my password" page that emails the user a special email message, containing another link that, when clicked, takes the user back to the "forgot my password" page, with a code that allows the user to reset his or her password. They key to making this work in a secure way is through another use of hashing.

With hashing, when a user requests a password change, he or she receives an email message that contains a special link back to the "forgot my password" page. The link contains the following things in its URL parameters:

- The username of the user who is requesting the password change
- The current date and time (in *ticks* [100-nanosecond intervals since January 1, 0001])
- A hash generated from the username, ticks, and a configured hash password

When the user clicks the link, the application creates a new hash from the username, the date and time in the link, and the hash password. This ensures that only links generated by the application are allowed (no one else will have access to the hash password, so no one else will be able to generate a hash that will match).

The date and time in the link are also compared to the current date and time to ensure that the link is not too old. This is important because you do not want change-password emails to be valid forever.

If both checks are passed, the user sees controls that he or she can use to set a new password.

The HTML code for such a Web form is shown in Listing 13.2.

LISTING 13.2 `.aspx` Code for a "Forgot My Password" Web Form

```
<body>
  <form id="Form1" method="post" runat="server">
    <div id="RequestControls" runat="server">
    Enter your username to receive an email
      with instructions for changing your password
      <div>
      <asp:textbox id="UsernameTextBox" runat="server" />
      <asp:button id="RequestButton"
                  runat="server"
                  Text="Request a password change" />
      </div>
    </div>
    <div id="RequestMadeControls" runat="server">
    You will now receive an email with
      instructions for changing your password.
    </div>
    <div id="ChangePasswordControls" runat="server">
      <div>Enter a new password
        <asp:textbox id="Password1TextBox" runat="server"/>
      </div>
      <div>Enter the password again
      <asp:textbox id="Password2TextBox" runat="server" />
      </div>
      <div>
      <asp:button id="ChangePasswordButton"
                  runat="server"
                  Text="Change My Password" />
      </div>
    </div>
  </form>
</body>
```

There are three parts to the page, each contained in a server-side `<div>` element so that you can display them one at a time:

- **`RequestControls`**—Controls that allow the user to request a password change
- **`RequestMadeControls`**—Controls that are displayed after a request is made
- **`ChangePasswordControls`**—Controls that allow the user to change his or her password

The `Page_Load` event in the code-behind file (see Listing 13.3) determines which to display.

LISTING 13.3 Code-Behind Code for a "Forgot My Password" Web Form

```
Private Sub Page_Load(ByVal sender As System.Object, _
                      ByVal e As System.EventArgs) _
            Handles MyBase.Load
  If Not Page.IsPostBack Then
    If Request.QueryString("Username") Is Nothing Then
      RequestControls.Visible = True
      RequestMadeControls.Visible = False
      ChangePasswordControls.Visible = False
    Else

      Dim username As String = Request.QueryString("Username")
      Dim ticks As String = Request.QueryString("Date")
      Dim UrlHash As String = Request.QueryString("Check")
      Dim stringtohash As String = username & ticks & _
         ConfigurationSettings.AppSettings("PasswordRequestHashPassword")

      Dim dt As DateTime = New DateTime(Long.Parse(ticks))

      If dt.AddHours(ConfigurationSettings.AppSettings("PasswordRequestTimeout")) _
                            > DateTime.Now Then
        Dim computedHash = _
         FormsAuthentication.HashPasswordForStoringInConfigFile(stringtohash, "sha1")

        If UrlHash = computedHash Then
          RequestControls.Visible = False
          RequestMadeControls.Visible = False
          ChangePasswordControls.Visible = True
        Else
          RequestControls.Visible = True
          RequestMadeControls.Visible = True
          ChangePasswordControls.Visible = False
          RequestMadeControls.InnerText = _
             "There was a problem with your request, please request another email"
        End If
```

LISTING 13.3 Continued

```
      Else
        RequestControls.Visible = True
        RequestMadeControls.Visible = True
        ChangePasswordControls.Visible = False
        RequestMadeControls.InnerText = _
             "Your request email has timed out, please request another email"
      End If
    End If
  End If
End Sub
```

Note that error-handling code has been omitted from this example for simplicity. Normally, it would be wise to include code to deal with an error from the call to `ConfigurationSettings.AppSettings`, in case the setting is not available.

If the `UserID` parameter does not appear in the URL, you simply display the `RequestControls` controls.

If the `UserID` parameter is present, you need to process the URL parameters to determine whether the page has been linked to from a valid change-password email.

First, you extract the username, tick value, and hash from the URL parameters:

```
Dim username As String = Request.QueryString("Username")
Dim ticks As String = Request.QueryString("Date")
Dim UrlHash As String = Request.QueryString("Check")
```

You can then generate the hash value, using the username and tick value from the URL and the configured hash password:

```
Dim stringtohash As String = username & ticks & _
    ConfigurationSettings.AppSettings("PasswordRequestHashPassword")
```

Before proceeding any further, you check that the tick value does not correspond to a date and time that is too old:

```
Dim dt As DateTime = New DateTime(Long.Parse(ticks))

If dt.AddHours(ConfigurationSettings.AppSettings("PasswordRequestTimeout")) _
      > DateTime.Now Then
```

If the date and time are not too old, you compute the hash value:

```
         Dim computedHash = _
FormsAuthentication.HashPasswordForStoringInConfigFile(stringtohash, "sha1")
```

You can then compare the computed hash to the hash included in the URL, to ensure that they match:

```
If UrlHash = computedHash Then
  RequestControls.Visible = False
  RequestMadeControls.Visible = False
  ChangePasswordControls.Visible = True
Else
  RequestControls.Visible = True
  RequestMadeControls.Visible = True
  ChangePasswordControls.Visible = False
  RequestMadeControls.InnerText = "There was a problem
➥with your request, please request another email"
End If
```

If the computed hash and the hash included in the URL match, you display the change-password controls. If they do not match, you display an error message.

The code in Listing 13.4 shows how a change-password email is created and sent.

LISTING 13.4 The Click Event Handler for the Request Button

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

  Dim username as string = UsernameTextBox.Text

  If BusinessLogic.UsernameExists(username) Then

    Dim dateTimeTicks As Long = DateTime.Now.Ticks
    Dim stringToHash As String = username & dateTimeTicks &
ConfigurationSettings.AppSettings("PasswordRequestHashPassword")

    Dim hash As String = _
FormsAuthentication.HashPasswordForStoringInConfigFile(stringToHash, "sha1")

    Dim email As New MailMessage
    email.To = BusinessLogic.GetEmailAddress(username)
    email.From = ConfigurationSettings.AppSettings("AdminEmail")
    email.Subject = "Your password change request for " & _
      ConfigurationSettings.AppSettings("CommunityName")
      Dim body As New StringBuilder
      Body.Append("Navigate to the following link to change your password: ")
      body.Append("http://")
      body.Append(Request.Url.Authority)
      body.Append(Request.Url.AbsolutePath)
```

LISTING 13.4 Continued

```
      body.Append("?MemberID=")
      body.Append(member.PrimaryKey1)
      body.Append("&Date=")
      body.Append(dateTimeTicks)
      body.Append("&Check=")
      body.Append(hash)

      email.Body = body.ToString
    RequestMadeControls.InnerText = _
   "You will now receive an email with instructions for changing your password."
    RequestMadeControls.Visible = True
    RequestControls.Visible = False

    Try
      SmtpMail.SmtpServer = ConfigurationSettings.AppSettings("SMTPServer")
      SmtpMail.Send(email)
    Catch ex As Exception
      RequestMadeControls.InnerText = _
         "The email could not be sent - please contact the site admin"
      RequestMadeControls.Visible = True
    End Try

  Else
    RequestMadeControls.InnerText = "Username not recognised - did you mistype it?"
    RequestMadeControls.Visible = True
  End If
End Sub
```

The important part of this code is the following section, which computes the hash that should be included in the link in the email:

```
Dim dateTimeTicks As Long = DateTime.Now.Ticks
Dim stringToHash As String = username & dateTimeTicks &
ConfigurationSettings.AppSettings("PasswordRequestHashPassword")

Dim hash As String = _
 FormsAuthentication.HashPasswordForStoringInConfigFile(stringToHash, "sha1")
```

Also of interest is the section that adds the link to the email:

```
      Dim body As New StringBuilder
      Body.Append("Navigate to the following link to change your password: ")
      body.Append("http://")
      body.Append(Request.Url.Authority)
      body.Append(Request.Url.AbsolutePath)
```

```
body.Append("?MemberID=")
body.Append(member.PrimaryKey1)
body.Append("&Date=")
body.Append(dateTimeTicks)
body.Append("&Check=")
body.Append(hash)

email.Body = body.ToString
```

Persistent Authentication Cookies

By default, each user will be able to continue to use an ASP.NET application that is configured with the default forms authentication settings after logging in until one of two things occurs: The user does not make a request for a period of time set by the `timeout` attribute of the `<forms>` configuration, or the user closes his or her browser. The mechanism for this is provided by the cookie that is used to persist the authentication token between requests. When the forms authentication module sends the cookie to the client, the cookie is set to expire at a particular time and is set to be nonpersistent; browsers should store it in memory so that it is removed when the browser is closed. By default, the expiration time is updated with each new request (although a new cookie is not set with every request); you can have the cookie fixed to expire a configuration time after sign-in by setting the `slidingExpiration` attribute of the `<forms>` element to `False`.

But what if you want to remember the user between visits? It is common to give users the option to be signed in automatically if they visit the application again from the same browser on the same machine.

It is very easy to have forms authentication create a persistent cookie to persist the authentication ticket. You simply set the second parameter of `FormsAuthentication.RedirectFromLoginPage`, `FormsAuthentication.SetAuthCookie`, or `FormsAuthentication.GetAuthCookie` to `True`. For example, to have the sign-in control discussed earlier in this chapter create a persistent cookie, you would use the following line of code:

```
FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, True)
```

We do not recommend that you create a persistent cookie by default: It is insecure for any users who are connecting to your application from a shared computer. Instead, we suggest that you default to a nonpersistent cookie and provide a `CheckBox` control that allows the user to specify that the application should remember him or her when he or she connects from that machine.

Using persistent cookies raises a couple issues that are rather complex to address: How do you have a persistent cookie timeout, and how can you enforce a timeout, even if users are willing to manipulate their cookies? The following sections describe the possibilities.

Setting a Timeout

The problem with setting a timeout for a persistent cookie is that, when forms authentication is set to use a persistent cookie, the forms authentication module will ignore the timeout setting in the configuration file.

In order to use timeouts with persistent cookies, you need to get a little more hands-on with the authentication cookie.

Rather than use the call to `FormsAuthentication.SetAuthCookie`, you need to use the following code:

```
Dim cookie As HttpCookie = _
  FormsAuthentication.GetAuthCookie(UsernameTextBox.Text, True)

cookie.Expires = DateTime.Now.AddDays(7)

Response.Cookies.Add(cookie)
```

You create a new cookie object and use `FormsAuthentication.GetAuthCookie` to store a valid authentication ticket for the username in it. Then you set the expiration date and time of the cookie to the current date and time plus a week. Finally, you add the cookie to the response.

Mandatory Expiration

Setting a cookie expiration date and time is all very well if you are not too worried about malicious users stealing and modifying cookies, but it does not help you if you want to enforce a limit on how long the authentication cookie is persisted. Cookies are stored on the user's machine as text files and, with a little information about the cookie format that the browser uses, any user could make changes to the expiration date and time.

In order to enforce a timeout that the user cannot tamper with, you need to store the timeout in the cookie, in encrypted form. Fortunately, the authentication ticket, which is by default encrypted, stores its own timeout value, independently of the cookie expiration.

The following changes to the sign-in code enforce expiration in the encrypted authentication ticket itself:

```
If FormsAuthentication.Authenticate(UsernameTextBox.Text, _
                                    PasswordTextBox.Text) Then
   'set the authentication cookie

   Dim ticket As New FormsAuthenticationTicket(1, _
                                       UsernameTextBox.Text, _
                                       DateTime.Now, _
                                       DateTime.Now.AddDays(7), _
                                       True, "")
```

```
    Dim encrypted As String = FormsAuthentication.Encrypt(ticket)

    Dim cookie As New HttpCookie(FormsAuthentication.FormsCookieName, _
                                 encrypted)


    cookie.Expires = DateTime.Now.AddDays(7)

    Response.Cookies.Add(cookie)

    'refresh the page
    Response.Redirect(Request.Url.PathAndQuery)
End If
```

Confusion in the .NET Framework Documentation

The .NET Framework is somewhat confusing when it comes to the `FormsAuthenticationTicket` object. It implies that the `IssueDate` and `ExpirationDate` properties are tied to the settings of the cookie that stores the authentication ticket. In fact, they are not: The properties of the ticket are used by the forms authentication module when authenticating users and are separate from the cookie settings.

Worse, the documentation suggests that the `ExpirationDate` property of the `FormsAuthenticationTicket` object should be set to the `DateTime` value when the ticket is issued. This is a very bad idea! Such tickets expire as soon as they are issued and are no use to anybody.

Now, you can create a `FormsAuthenticationTicket` object and use the constructor to specify the version number (in case future versions of forms authentication support different options), the username, the issue date and time, the expiration date and time, whether you want a persistent cookie, and some custom data (an empty string, in this case).

Using Forms Authentication in Web Farms

Because forms authentication uses the ticket stored in an authentication cookie (or, as you will see later in this chapter, the URL) to persist the user's authentication details between requests, the system very easily scales to Web farms. Provided that each server in the farm is set up to accept the same authentication ticket, users can connect to any server in the farm without any authentication problems.

There are two things you need to do to ensure that each server will accept the tickets issued by the others. First, you need to ensure that the `<forms>` elements of the servers' configurations match. Second, you need to ensure that all the servers use the same keys for encrypting and validating authentication tickets.

By default, ASP.NET auto-generates these keys at random, which obviously does not lead to the servers in a Web farm having matching keys. You therefore need to explicitly set the values. You do this through the `<machineKey>` configuration element, which looks like this when it is filled with some suitable keys:

```
<machineKey
    decryptionKey=" BA753FA48201BF29D4691149E33191F72A5D449F5847891F63101B4FF011475084"
    validationKey="51A7C24620AFC1BD27E37867EB5D57C83A92CC886C9612318B1348C868F91E8670
➥DF332B63222CD9345A73BD9295D113BDC5824E18FFD76B0A536C0461DE9C93B4"
    validation="SHA1" />
```

This element should go inside the `<system.web>` element of the `web.config` file or (if you want to use the same keys for the whole machine) in the `machine.config` file. We don't recommend that you put this element in the `machine.config` file. It is usually best to keep the Web applications on a machine separate unless you explicitly want them to use the same key.

The decryption key can be either 16 hexadecimal characters (for DES encryption) or 48 characters (for Triple DES [3DES] encryption). We strongly recommend that you use the more secure 3DES option.

The validation key can contain between 40 and 128 characters. The longer the key, the more secure it is. Again, we suggest that you use the strongest option possible.

How do you get the keys to enter into the configuration file? Well, you can enter them by hand, but that is both time-consuming and somewhat insecure; no matter how random you think your typing is, it is almost certainly not as random as a random number generator. Therefore, you need a tool to generate the keys for you.

A small Windows Forms application is included with the code for this chapter (see `www.daveandal.net/books/6744/`). Its interface looks as shown in Figure 13.3.

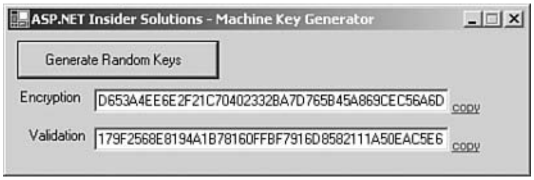


FIGURE 13.3 A key-generator application.

Clicking the Generate Random Keys button causes the application to generate a new random decryption key and validation key. The copy links then copy the relevant key to the Clipboard, so that they are ready to be pasted into the `web.config` file.

The important part of the code for this application is the following `GenerateKey` method:

```
Private Function GenerateKey(ByVal length As Integer) As String
  Dim randomBytes(length / 2) As Byte

  Dim randomNumberGenerator As New RNGCryptoServiceProvider

  randomNumberGenerator.GetBytes(randomBytes)

  Dim sb As New StringBuilder(length)
  Dim b As Byte
```

```
  For Each b In randomBytes
    sb.Append(String.Format("{0:X2}", b))
  Next

  Return sb.ToString()

End Function
```

Note that the code file imports the `System.Security.Cryptography` and `System.Text` namespaces.

The `GenerateKey` method creates a new array of `Byte` objects, which are half the length of the required key. (Each randomly generated byte generates two hexadecimal characters in the output.)

The array is then filled with random bytes from an `RNGCryptoServiceProvider` class—a random number generator that is designed to be random enough for cryptographic purposes (much more random that the `Random` class).

Then you loop through the array and add each byte to a `StringBuilder` instance as two hexadecimal characters. Finally, the completed string is returned. The Windows Forms application uses the `GenerateKey` method to generate both of the keys.

When the `<machineKey>` elements are set up with matching keys in all the servers of a Web farm, each machine will accept authentication tickets issued by the other servers, so users will have no problem if they move between servers during a session.

Using `<machineKey>` Elements to Implement Single Sign-in Systems

There is no reason you cannot use the technique described in the preceding section for matching `<machineKey>` elements to get different applications to accept each other's authentication tickets. You could have several applications on the same server accept the same tickets and thus recognize the same users. You could even have different applications hosted on different servers recognize the same set of signed-in users.

The big limitation to this technique has to do with the use of cookies to carry the authentication ticket. Browsers only send cookies to the domain for which they are defined, and they accept cookies from a Web application only if the application is part of the domain that the cookie is defined for. This means that if I want to share forms authentication tickets between `www.syzygy-visuals.co.uk` and `www.zoetrope.org.uk`, I have a problem.

There are some solutions, though. If you use cookieless forms authentication, which is described later in this chapter, in the section "Cookieless Forms Authentication," the problem disappears because the authentication ticket will be carried in the URL rather than in a cookie. There are other approaches, but they are all very much more complex.

Cookieless Forms Authentication

Forms authentication usually uses a cookie to carry the user's authentication ticket between requests. This is not the only way to do this, though, and using a cookie imposes some limitations that you might want to avoid. As previously mentioned, using a cookie causes problems if you want to share authentication tickets between applications hosted in different domains. Another problem is a more general one: Some clients might not be set up to support cookies. Users whose client does not accept cookies will not be able to sign in to an application that relies on cookies to persist the authentication ticket.

The ASP.NET development team realized that this could be a problem and built the forms authentication system so that it is not limited to using cookies. If a cookie with the correct name is not found, the forms authentication module looks for a URL parameter with the same name and attempts to decrypt the parameter as an authentication cookie. This makes implementing cookieless forms authentication both very easy and very hard, depending on how you look at it. On one hand, you don't have to make any configuration changes; you just need to ensure that the authentication ticket is present in the URL parameters for any links that you make back to the application. On the other hand, ensuring that the authentication ticket is maintained from page to page could be problematic if you have lots of internal links in the application.

You need to make some changes to the login code in the sample application in order to support cookieless forms authentication: You need to add the authentication ticket to the URL after a user successfully signs in.

Listing 13.5 shows the sign-in code from the login control discussed earlier in this chapter, adapted to add the ticket to the URL.

LISTING 13.5 Sign-in Code Adapted for Cookieless Forms Authentication

```
If FormsAuthentication.Authenticate(UsernameTextBox.Text, _
                                    PasswordTextBox.Text) Then

  'set the authentication cookie
  FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, True)

  'get an authentication ticket
  Dim ticket As New FormsAuthenticationTicket(UsernameTextBox.Text, _
                                              False, 30)
  'encrypt the ticket
  Dim encryptedTicket As String = FormsAuthentication.Encrypt(ticket)

  'refresh the page with the authentication ticket added
  Dim currentUrl As New StringBuilder(Request.Url.PathAndQuery)

  'check whether there is already a query string
  If currentUrl.ToString.IndexOf("?") = -1 Then
```

LISTING 13.5 Continued

```
    'there is not a query string, so add one
    currentUrl.Append("?")
  Else
    'there is a query string, so add the parameter to it
    currentUrl.Append("&")
  End If
  currentUrl.Append(FormsAuthentication.FormsCookieName)
  currentUrl.Append("=")
  currentUrl.Append(encryptedTicket)

  Response.Redirect(currentUrl.ToString)
End If
```

You create a new `FormsAuthenticationTicket` object with the properties that you require. You then use the `FormsAuthentication.Encrypt` method to create an encrypted string that contains the ticket.

Note that you do not simply add the encrypted ticket string to the URL string; you have to check whether there are already URL parameters so that you can decide whether to use the `?` or the `&` before the parameter name.

The sign-in code still includes the line that sets the authentication cookie. This code is set up to do both cookie and cookieless authentication.

You need to make some changes to the sign-out code in order to remove the ticket from the URL:

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

  'remove the authentication cookie
  FormsAuthentication.SignOut()

  'now we need to ensure the ticket is removed from the URL...

  'get current URL without query string parameters
  Dim currentUrl As New StringBuilder(Request.Url.LocalPath)

  'we now need to add each query string parameter except the auth ticket
  'back onto the URL
  Dim parameterName As String
  Dim firstParameter As Boolean = True
  For Each parameterName In Request.QueryString.AllKeys
    'check that this is not the ticket
```

```
    If Not parameterName = FormsAuthentication.FormsCookieName Then
      'if this is the first parameter start a query string
      If firstParameter Then
        currentUrl.Append("?")
      Else
        'this is not the first parameter, so continue the query string
        currentUrl.Append("&")
      End If
      'add the parameter
      currentUrl.Append(parameterName)
      currentUrl.Append("=")
      currentUrl.Append(Request.QueryString.Item(parameterName))
    End If
  Next

  'redirect to the URL with the ticket removed
  Response.Redirect(currentUrl.ToString)
End Sub
```

You run the standard `FormsAuthentication.SignOut` method to remove the cookie. Then you get the current URL, without the query string, and loop through the query string parameters in `Request.QueryString`, adding them to the URL. If the parameter is the forms authentication ticket parameter, you do not add it.

The user can then sign in, regardless of whether his or her browser sends and receives the authentication ticket cookie. The user will also be able to sign out. Both signing in and signing out will respect and preserve any existing URL parameters.

You hit a problem as soon as you want the user to be able to link to another page in the application, though. The authentication ticket URL parameter will not be carried along with the link unless you explicitly include it.

You could add code to every page in the application to append the ticket to every link it creates, but that would be a huge duplication of effort and code. There are a couple ways to avoid this. You can create a reusable hyperlink control that you can add to pages wherever you need a hyperlink, or you can add to the application some code that will automatically add the ticket to any local URLs on each page that it sends to the users. The following sections describe these two options.

Creating a Hyperlink Control to Add the Authentication Ticket

Thanks to ASP.NET's ability to inherit from existing controls, it is actually very easy to create a hyperlink control that will maintain the authentication ticket.

Listing 13.6 shows a control that inherits from `System.Web.UI.WebControls.HyperLink`.

LISTING 13.6 A Hyperlink Control That Automatically Includes the Authentication Ticket

```
Imports System.Web.UI
Imports System.Web.Security
Imports System.Text

Public Class LinkWithTicket
  Inherits WebControls.HyperLink

  Protected Overrides Sub Render(ByVal writer As HtmlTextWriter)

    Dim cookieName As String = FormsAuthentication.FormsCookieName

    'check that the request is authenticated
    If Not Page.Request.QueryString.Item(cookieName) Is Nothing Then


      Dim UrlBuilder As New StringBuilder(NavigateUrl)
      'check whether there is already a query string in the link
      If NavigateUrl.IndexOf("?") = -1 Then
        'there is no query string, so start one
        UrlBuilder.Append("?")
      Else
        'there is a query string, so add to it
        UrlBuilder.Append("&")
      End If

      'add the parameter
      UrlBuilder.Append(cookieName)
      UrlBuilder.Append("=")
      UrlBuilder.Append(Page.Request.QueryString.Item(cookieName))

      'set the Url to the new one, including the ticket
      NavigateUrl = UrlBuilder.ToString()
    End If

    'pass the rest of the rendering work on to the base class
    MyBase.Render(writer)
  End Sub
End Class
```

You override the `Render` method in order to add the authentication ticket to the URL just before the control is rendered. The code that adds the ticket is much the same as the code that you used in the sign-in control earlier in this chapter, in the section “Building a Reusable Sign-in Control.”

If you now use this control wherever you want an internal hyperlink in the application, the authentication ticket parameter will be added to the query string of the link.

Protecting Non-ASP.NET Content

You usually use forms authentication to control access to a Web application itself—mainly the `.aspx` files that users must request in order to view the Web forms of the application. However, you can use forms authentication with any kind of files, provided that those files are served by ASP.NET.

In order to bring a file type under the control of ASP.NET—and thus forms authentication—you need to do two things:

- Map the file type to ASP.NET in Internet Information Services (IIS)
- Define which `HttpHandler` implementation you would like ASP.NET to use to handle requests for that file type

To map the file type to ASP.NET, you select Control Panel, Administrative Tools, Internet Information Services. Then you right-click the Web application to configure and select Properties. Finally, you click the Configuration button in the Application Settings section of the Directory tab. You should see something like what is shown in Figure 13.4.

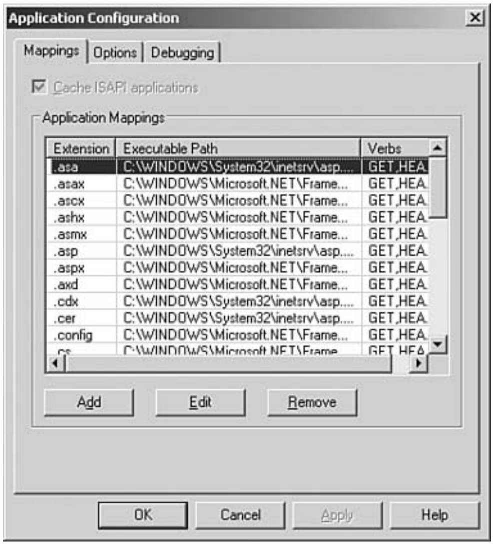


FIGURE 13.4 The Application Configuration window in IIS.

The list in Figure 13.4 shows each file type, along with the executable it is mapped to (note that not all these are actually executables—many, including ASP.NET, are DLLs). Double-clicking any of them brings up a window like the one shown in Figure 13.5.

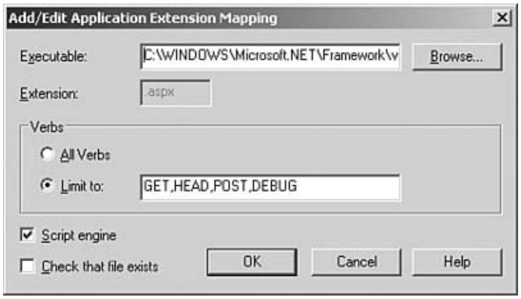


FIGURE 13.5 The file type settings window.

In order to map a new file type to ASP.NET, you simply need to copy the executable path from one of the file types already mapped to ASP.NET and create a new mapping. (You need to right-click and then select Copy because the keyboard shortcut does not work in the Mappings tab of the Application Configuration dialog). In this way, you can specify whatever file types you'd like to be sent to ASP.NET for processing when users request them from IIS.

The second part of the process is to tell ASP.NET what it should do with the file types you map to it. You can get away with not doing this step if you simply want ASP.NET to perform authorization and then pass the file to the user; this is the default option. However, it is best, especially when it comes to security, to explicitly define what you want to happen.

You tell ASP.NET what to do with each file type by associating the types with `HttpHandler` implementations in the configuration file. The machinewide defaults are stored in `Windows/Microsoft.NET/Framework/`*`[version]`*`/Config/Machine.Config`:

```
<httpHandlers>
<add verb="*" path="*.vjsproj" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.java" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.jsl" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="trace.axd" type="System.Web.Handlers.TraceHandler"/>
<add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
<add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory"/>
<add verb="*" path="*.asmx" type="System.Web.Services.Protocols.WebServiceHandlerFactory,
➥System.Web.Services, Version=1.0.5000.0, Culture=neutral,
➥PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
<add verb="*" path="*.rem" type="System.Runtime.
➥Remoting.Channels.Http.HttpRemotingHandlerFactory,
➥System.Runtime.Remoting, Version=1.0.5000.0,
➥Culture=neutral, PublicKeyToken=b77a5c561934e089"
➥validate="false"/>
<add verb="*" path="*.soap" type="System.Runtime.
➥Remoting.Channels.Http.HttpRemotingHandlerFactory,
➥System.Runtime.Remoting, Version=1.0.5000.0,
➥Culture=neutral, PublicKeyToken=b77a5c561934e089"
➥validate="false"/>
<add verb="*" path="*.asax" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.ascx" type="System.Web.HttpForbiddenHandler"/>
```

```
<add verb="GET,HEAD" path="*.dll.config" type="System.Web.StaticFileHandler"/>
<add verb="GET,HEAD" path="*.exe.config" type="System.Web.StaticFileHandler"/>
<add verb="*" path="*.config" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.cs" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.csproj" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.vb" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.vbproj" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.webinfo" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.asp" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.licx" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.resx" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.resources" type="System.Web.HttpForbiddenHandler"/>
<add verb="GET,HEAD" path="*" type="System.Web.StaticFileHandler"/>
<add verb="*" path="*" type="System.Web.HttpMethodNotAllowedHandler"/>
</httpHandlers>
```

Lots of file types are mapped to `HttpForbiddenHandler`, which is used to prevent those file types from being downloaded and to display an error message if an attempt is made to download one.

The system works by using the first handler in the list that matches with the file type and verb of the request. Therefore, the defaults are found at the bottom of the file. The default when a `GET` or `HEAD` verb is used is `StaticFileHandler`:

```
<add verb="GET,HEAD" path="*" type="System.Web.StaticFileHandler"/>
```

This handler simply reads the file and streams it to IIS for delivery to the client.

If you want to explicitly set the handler for a file type, you add an `<add>` element to the `<httpHandlers>` element of the configuration file. (You can do this either in the `machine.config` file or in a `web.config` file if you want to add the handler for a specific application.) For example, if you have mapped PDF files to ASP.NET in IIS, you can then specify that you want ASP.NET to treat them as static files by adding the following `<add>` element:

```
<add verb="GET,HEAD" path="*.pdf" type="System.Web.StaticFileHandler"/>
```

After you have mapped file types to ASP.NET, they fall under the protection of the ASP.NET security framework. Requests for these file types will be subject to authorization against the rules in the `<authorization>` section of the configuration file. In addition, ASP.NET will only be able to access files that that `ASPNET` user is configured with access for, so it will only be able to serve those files.

Mapping additional file types to ASP.NET causes a very slight performance hit. It is not too great if the `StaticFileHandler` handler is used, but still, you should map only the file types that need authorization by ASP.NET.

There seems to a problem with using the redirection technique that `Response.Redirect` (and `FormsAuthentication.RedirectFromLoginPage`) employs when redirecting to certain file types. For some reason, redirecting in this way causes the browser to report a corrupt file or simply not display the file. An example of this is Adobe Acrobat files (PDF files). These files will load fine when they're linked to directly, but when they're redirected to (for example, after a successful sign-in), they will not display properly.

There is a way around this problem. The refresh HTTP header works fine with these file types, so you can use it to do the redirection. On a forms authentication sign-in page, you can use the following code in place of the call to `RedirectFromLoginPage`:

```
FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, false)

Dim url as String = FormsAuthentication.GetRedirectUrl(UsernameTextBox.Text, false)

Response.AppendHeader("refresh", "0;url=" + url)
```

You set the authentication cookie, get the URL that you need to redirect to, and then add the refresh header, with a time of `0` so that the redirection happens immediately.

Supporting Role-Based Authorization with Forms Authentication

Role-based authorization is a common requirement for Web applications. In order for role-based authorization to be able to work, the authentication system has to provide it with the roles that the current user belongs to. By default, forms authentication does not do this, so it does not support role-based authorization. However, it is very easy to implement roles with forms authentication because most of the work has already been done. You just need to add a little more code to persist each user's roles in his or her authentication ticket and bind the roles to the context of the user's requests.

In the sign-in method, you need to create an authentication ticket from scratch in order to store the roles in the `UserData` property of the ticket (see Listing 13.7).

LISTING 13.7 Sign-in Code with Support for Roles

```
If FormsAuthentication.Authenticate(UsernameTextBox.Text, PasswordTextBox.Text) Then

  'get the roles
  Dim roles() As String = BusinessLogic.GetRoles(UsernameTextBox.Text)

  'create a semicolon delimited string of roles
  Dim rolesBuilder As New StringBuilder
  For Each role As String In roles
    rolesBuilder.Append(role)
    rolesBuilder.Append(";")
  Next

  'create the auth ticket
```

LISTING 13.7 Continued

```
  Dim ticket As New FormsAuthenticationTicket(1, _
                                            UsernameTextBox.Text, _
                                            DateTime.Now, _
                                            DateTime.Now.AddDays(7), _
                                            True, _
                                            rolesBuilder.ToString)

  'encrypt the ticket
  Dim ticketString As String = FormsAuthentication.Encrypt(ticket)

  'put the ticket in a cookie
  Dim cookie As New HttpCookie(FormsAuthentication.FormsCookieName, _
                               ticketString)

  'add the cookie to the response
  Response.Cookies.Add(cookie)

  'refresh the page
  Response.Redirect(Request.Url.PathAndQuery)
End If
```

You extract the roles for the user from the business logic, create a semicolon-delimited string that contains them, and then store that in the ticket.

You then persist each user's roles in his or her authentication ticket. There is one more thing you need to do in order for role-based authorization to work: At the start of each page request, you need to store the roles from the ticket in the `Context.User` object, where the authorization module expects to find them. You can do this by adding the following method to the `Global.Asax` code-behind file:

```
Sub Application_AuthenticateRequest(ByVal sender As Object, ByVal e As EventArgs)

  If Request.IsAuthenticated Then
    Dim identity As FormsIdentity = CType(Context.User.Identity, _
                                          FormsIdentity)

    Dim roles() As String = identity.Ticket.UserData.Split(";")

    Dim principal As New GenericPrincipal(Context.User.Identity, roles)

    Context.User = principal
  End If
End Sub
```

This method handles the `AuthenticateRequest` event, which fires after authentication is carried out on each request. You therefore need to check that authentication was successful before you bind roles.

If the request is authenticated, you extract the roles from the authentication ticket (which is now found in the `FormsIdentity` object in `Context.User.Identity`). Next, you create a new `GenericPrincipal` object with the existing identity and the roles you have extracted. Finally, you replace the existing principal in `Context.User` with the new `GenericPrincipal` object.

Now the authorization module will be able to find the roles where it expects them, so you can use the standard role-based authorization configuration in the `web.config` file exactly as you would if you were using Windows authentication.

Using Multiple Sign-in Pages

Standard forms authentication provides one sign-in page that is used to deal with all users who require authentication. But what if you want different parts of the application to use a different sign-in page?

You could set up the different parts of the application as separate Web applications so that each could have its own configured sign-in page, but you might want to share session state, caching, or other features between the parts of the application, which would not be possible if you set up separate applications.

The solution is to set up the main sign-in page so that it will look for a sign-in page in the folder of the originally requested file. If one is found, the main sign-in page will redirect to that sign-in page. If one is not found, the sign-in page will display as it usually does.

You can use the following `Page_Load` event handler to set up the main sign-in page in this way:

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

  'get the original request URL
  Dim originalUrl As String = FormsAuthentication.GetRedirectUrl("", True)

  'regex to match everything after the final /
  Dim filenameRegEx As New Regex("/([^/]*)$")

  'get the path by removing the filename and querystring
  Dim path As String = filenameRegEx.Replace(originalUrl, "")

  'create an OS filesystem path to a signin page in the folder of the request
  Dim signinFile = Server.MapPath(path + "/signin.aspx")

  'check whether the signin page exists
  If File.Exists(signinFile) Then
```

```
    Dim redirectUrl As New StringBuilder
    redirectUrl.Append(path)
    redirectUrl.Append("/signin.aspx?ReturnUrl=")
    redirectUrl.Append(Server.UrlEncode(originalUrl))

    Response.Redirect(redirectUrl.ToString)
  End If

End Sub
```

Note that this code requires several `Imports` statements:

```
Imports System.IO
Imports System.Text
Imports System.Text.RegularExpressions
Imports System.Security
Imports System.Web.Security
```

You also need to ensure that each sign-in page in a subfolder is configured so that anonymous users can access it. (This is done automatically for the main sign-in page, but you have to do it yourself for any other sign-in pages.) You need to add a `<location>` element to the `web.config` file for each sign-in page:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <location path="trade/signin.aspx">
    <system.web>
      <authorization>
        <allow users="?" />
      </authorization>
    </system.web>
  </location>
  <location path="partners/signin.aspx">
    <system.web>
      <authorization>
        <allow users="?"
      </authorization>
    </system.web>
  </location>
    <location path="suppliers/signin.aspx">
    <system.web>
      <authorization>
        <allow users="?"
      </authorization>
    </system.web>
  </location>
```

You can now customize each sign-in page as you want to. You can even have the authentication code check against different sets of credentials if it makes sense to do so.

Dealing with Failed Authorization

A situation in which you might like a sign-in page to behave a little more intelligently is when a signed-in user attempts to access a file that he or she is not authorized to see.

In the standard forms authentication setup, such users are forwarded to the sign-in page in the same way as users who are not signed in. It would be much better if you could display a message to let these users know that they tried to access a resource they are not authorized to view rather than simply showing them the sign-in controls.

This is actually very easy to do. All you have to do is to check whether a user is already signed in before displaying the sign-in page:

```
'check whether there is a user signed in
If Request.IsAuthenticated Then
  'there is a user signed in, so they must have failed authorization
  Response.Redirect("NotAuthorized.aspx")
Else
  'no user signed in, so redirect to a sign in page if one exists for
  'the folder of the original request

  'get the original request URL
  Dim originalUrl As String = FormsAuthentication.GetRedirectUrl("", True)

  'regex to match everything after the final /
  Dim filenameRegEx As New Regex("/([^/]*)$")

  'get the path by removing the filename and querystring
  Dim path As String = filenameRegEx.Replace(originalUrl, "")

  'create an OS filesystem path to a signin page in the folder of the request
  Dim signinFile = Server.MapPath(path + "/signin.aspx")

  'check whether the signin page exists
  If File.Exists(signinFile) Then
    Dim redirectUrl As New StringBuilder
    redirectUrl.Append(path)
    redirectUrl.Append("/signin.aspx?ReturnUrl=")
    redirectUrl.Append(Server.UrlEncode(originalUrl))
```

```
      Response.Redirect(redirectUrl.ToString)
    End If

  End If

End Sub
```

If `Request.IsAuthenticated` returns `true`, there must be a user signed in, and therefore the user must have failed authorization.

Listing Signed-in Users

It is common for modern multiuser applications to show their users which other users are currently signed in. It is also useful for the administrators to know which users are actively using an application at a particular time.

Standard forms authentication is not set up to provide this functionality. Forms authentication uses a cookie or the URL to persist the authentication ticket between requests, so it does not remember the user between one request and the next. In order to provide a list of signed-in users, you have to do a little extra work to build the infrastructure that the feature requires.

You can create a data structure to hold the names of the signed-in users when the application starts, by adding code to the `Application_Start` event in the global `.asax` code-behind file:

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)

  'hashtable for sessionIDs and names of all signed in users
  Application.Item("SignedInUsers") = New Hashtable

  'counter for number of anonymous sessions
  Application.Item("AnonymousUsers") = CInt(0)

End Sub
```

You can add each user's session ID and username to the hash table when that user signs in. Before the user signs in (and after he or she signs out), you can track the user's presence on the application as an anonymous user by incrementing and decrementing the count of anonymous user sessions (that is, sessions that are not associated with a signed-in user).

You need to increment the counter of anonymous users when a new user session is started:

```
Sub Session_Start(ByVal sender As Object, ByVal e As EventArgs)
  Application.Item("AnonymousUsers") = _
                CInt(Application.Item("AnonymousUsers")) + 1
End Sub
```

When a session ends, you need to remove the user from the hash table (if there is a user signed in) or decrement the count of anonymous users (if there is not a user signed in):

```
Sub Session_End(ByVal sender As Object, ByVal e As EventArgs)
  Dim userList As Hashtable = _
    CType(Application.Item("SignedInUsers"), Hashtable)

  If userList.Contains(Session.SessionID) Then
    userList.Remove(Session.SessionID)
  Else
    Application.Item("AnonymousUsers") = _
                        CInt(Application.Item("AnonymousUsers")) - 1
  End If
End Sub
```

When a user signs in, you need to decrement the count of anonymous users and add the user to the hash table:

```
If FormsAuthentication.Authenticate(UsernameTextBox.Text, _
                                    PasswordTextBox.Text) Then

  'add the user to the list of sign-in users
  Dim userList As Hashtable = _
              CType(Application.Item("SignedInUsers"), Hashtable)
  If Not userList.ContainsValue(UsernameTextBox.Text) Then


    userList.Add(Session.SessionID, UsernameTextBox.Text)

    'decrement the number of anonymous sessions
    Application.Item("AnonymousUsers") = _
                        CInt(Application.Item("AnonymousUsers")) - 1

  End If


  'set the authentication cookie
  FormsAuthentication.SetAuthCookie(UsernameTextBox.Text, False)

  'refresh the page
  Response.Redirect(Request.Url.PathAndQuery)
End If
```

Note that you check to ensure that the user is not already in the hash table before adding the user. This guards against the possibility of a user signing in again when he or she is already signed in once.

When a user signs out, you do the reverse:

```
Private Sub SignOutButton_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
            Handles SignOutButton.Click
  Dim userList As Hashtable = _
    CType(Application.Item("SignedInUsers"), Hashtable)

  If userList.Contains(Session.SessionID) Then
    userList.Remove(Session.SessionID)
    Application.Item("AnonymousUsers") = _
                        CInt(Application.Item("AnonymousUsers")) + 1
  End If


  FormsAuthentication.SignOut()
  Response.Redirect(Request.Url.PathAndQuery)
End Sub
```

You check that the user is in the hash table before proceeding, for the same reasons that you carry out the check when the user signs in.

You now have a hash table that contains all the signed-in users stored in the `Application` object, along with an integer value that counts the anonymous user sessions. You can very easily use these to create a control to display the users to either all other users of the application or just the administrators.

Forcibly Signing Out a User

Standard forms authentication is not set up to kick a user out of an application. You can easily prevent a banned user from signing in; you simply update the credentials store that the sign-in code uses. The problem comes when you want to eject a user who is already signed in.

The forms authentication module will accept any valid authentication ticket in order to allow access to the application, so after you have issued a ticket to the user, you cannot simply invalidate it without changing the encryption and validation keys and invalidating the authentication tickets of all users. You don't want to check against the credentials store with every page request to see if the user has been banned; that would cause an additional database access for every page request that is made to the application, so you need to find another approach.

The solution is to maintain a list of recently ejected users (hopefully, this list won't be too big) and check the current user against this list at the start of each page request.

You can use an array list stored in the Application object to hold the username of each banned user:

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
  'create arraylist to hold banned usernames
  Application.Item("BannedUsers") = New ArrayList

End Sub
```

You need to check the user against this list every time a user is successfully authenticated:

```
Sub Application_AuthenticateRequest(ByVal sender As Object, ByVal e As EventArgs)
  Dim bannedUsers As ArrayList = _
            CType(Application.Item("BannedUsers"), ArrayList)

  'check whether the user is banned
  If bannedUsers.Contains(User.Identity.Name) Then
    'the user is banned so replace their principal with an anonymous one

    'create an anonymous identity
    Dim identity As New System.Security.Principal.GenericIdentity("")

    'create an anonymous principal
    Dim principal As New System.Security.Principal.GenericPrincipal( _
                              identity, New String() {})

    'bind the anonymous principal to the context
    Context.User = principal

  Else
    'the user is not banned - proceed as normal
  End If
End Sub
```

You can then add a user to the banned list in the admin system, using the following simple code:

```
Private Sub BanUser(ByVal username As String)
  Dim bannedUsers As ArrayList = _
    CType(Application.Item("BannedUsers"), ArrayList)

  If Not bannedUsers.Contains(username) Then
    'add user to the list of banned users in memory
    bannedUsers.Add(username)

    'add more code here to set the user as banned in the credentials store
  End If
End Sub
```

If the user is on the banned list, his or her principal object is replaced with a `GenericPrincipal` instance that contains an anonymous `GenericIdentity` object (that is, it contains an empty username). The user will then be treated as an anonymous user.

You might want to add additional code that logs attempts by banned users to access the application. You might also want to add to the `Application_Start` event handler code that loads the list of banned users from a persistent credentials store such as a database. If you don't do this, the list will be wiped each time you restart the application. This might not be a problem because auto-generated encryption and validation keys are regenerated with each application restart, but if you have defined explicit values for the `<machineKey>` element, authentication tickets will remain valid between restarts, and you will need to ensure that you persist the list of banned users.

Summary

This chapter covers one specific type of authentication, but it is the type of authentication that is likely to be encountered most often in ASP.NET applications. It is also, as you have seen, extremely flexible.

This chapter shows a wide variety of ways to customize forms authentication. The chapter starts by showing how to provide the sign-in feature in a reusable control rather than in a sign-in form.

This chapter discusses protection of passwords with one-way encryption (hashing), along with a way to help users who forget their passwords for systems that cannot recover them due to hashing protection.

Some methods for gaining additional control over the cookie that is used to persist the authentication ticket are presented, along with techniques for doing without cookies altogether.

This chapter shows the ease with which forms authentication can be applied across multiple servers. The difficulties of using this approach with multiple domains is explained, and cookieless authentication is suggested as one solution to the problem.

This chapter shows the influence of ASP.NET forms authentication by describing how to use it to protect content other than files directly related to ASP.NET.

Because role-based authorization is a very popular way to control access in Web applications, this chapter shows how to have forms authentication support roles in a way that allows it to interact transparently with the authorization system.

The chapter finishes by providing techniques for listing all the users who are signed into the application and for forcibly signing out particular users.

14

Customizing Security

IN THIS CHAPTER

Chapter 13, "Taking Advantage of Forms Authentication," shows how you can use the forms authentication module in a wide variety of ways. However, forms authentication is not the solution to all security needs. Sometimes you face security needs that require more customized solutions.

The event-based architecture of ASP.NET applications makes it easy for you to plug in to the same hooks that the authentication and authorization modules provided by Microsoft use. The first part of this chapter shows how you can build your own authentication and authorization modules to fulfill specific requirements. The second part of the chapter looks at the options presented by ASP.NET security configuration, particularly how you can configure ASP.NET applications to run at less than full trust.

Building a Custom Authentication Module

Authentication is the process of identifying users. Authentication modules use evidence in each request made to the application to identify which user is making the request. The authentication modules that ship with ASP.NET use as their evidence encrypted cookies in the request (forms authentication and Passport authentication) and evidence provided by IIS (Windows authentication).

What Is an Authentication Module?

In code terms, an *authentication module* is an HTTP module that handles the `AuthenticateRequest` event of the `HttpApplication` object. When the event fires, the authentication module checks the evidence associated with the request and populates the `Context.User` intrinsic object with an appropriate `IPrincipal` object (that is, an object of a class that implements the `IPrincipal` interface). The authorization module then uses `Context.User` as the basis for deciding whether the request should be authorized. The rest of the application code is then able to access whatever data is stored in the `IPrincipal` object.

It is actually pretty rare that you need to replace the standard authentication modules with a custom solution. Usually you can solve your problems by customizing one of the existing modules. Forms authentication, as you saw in Chapter 13, is particularly suitable for such manipulation.

One situation in which a custom authentication module is useful is when you want to identify access to an application according to which machine is trying to access the application rather than according to which specific user is making the request. For example, if you have an intranet application running on a closed network in a shopping mall, you might want to identify which client machines are used to make requests in order for the application to behave differently. (For example, the application might show special offers appropriate to stores near the client machine that is being used, or the map might be able to show a "you are here" label.)

There are all sorts of ways you can solve this problem. One clean way is to implement a custom authentication module that uses the IP address of the client machine as the evidence for authentication. The following sections show how to build a simple HTTP module that provides this functionality.

Building a Custom Identity Class

Before you build the HTTP module itself, you need to think about the `IPrincipal` object that it will use to populate `Context.User`. The *principal* represents the security context of the user. The interface has the following members:

Member	Return Type
`Identity`	`IIdentity`
`IsInRole` (String)	`Boolean`

Every `IPrincipal` object will store an `IIdentity` object that represents the identity of the authenticated user and will allow you to check whether the user is in a particular role.

You can build a custom class that implements `IPrincipal`, but there is rarely any need to do so; `System.Security.Principal.GenericPrincipal` does the job in the vast majority of cases. `GenericPrincipal` provides a constructor that takes an `IIdentity` object and an array of strings for the roles. It simply stores the roles internally and checks against them when `IsInRole` is called.

Most authentication approaches can use `GenericPrincipal`. The exception is Windows authentication, which needs to check roles against the user's Windows roles rather than against a set of roles stored by the principal object. It therefore defines a different `IPrincipal` implementation, `WindowsPrincipal`. In the IP authentication example, `GenericPrincipal` will do just fine. However, you should create a custom identity class that implements `IIdentity`. The reason becomes clear when you look at the members required by the `IIdentity` interface:

Member	Return Type
AuthenticationType	String
IsAuthenticated	Boolean
Name	String

The identity needs to return the type of authentication used to create it, so a new identity class is needed for each authentication module.

The following is the code for an identity class for the IP authentication module:

```
Public Class IPIdentity
  Implements System.Security.Principal.IIdentity

  Private _IP As String = Nothing

  Public Sub New(ByVal ip As String)
    _IP = ip
  End Sub

  'do not allow an IPIdentity to be created without an IP address
  Private Sub New()
  End Sub

  Public ReadOnly Property AuthenticationType() As String _
        Implements System.Security.Principal.IIdentity.AuthenticationType
    Get
      Return "IP"
    End Get
  End Property

  Public ReadOnly Property IsAuthenticated() As Boolean _
         Implements System.Security.Principal.IIdentity.IsAuthenticated
```

```
    Get
      'An IP Identity will only be used when authentication is successful
      Return True
    End Get
  End Property

  Public ReadOnly Property Name() As String _
          Implements System.Security.Principal.IIdentity.Name
    Get
      Return _IP
    End Get
  End Property
End Class
```

You store the IP address of the machine that is making the request in a private field. A constructor is provided to allow a new `IPIdentity` object to be created from an IP address, but you mark the default constructor as private so it cannot be used.

The members required by `IIdentity` are each provided. `AuthenticationType` returns `"IP"`. `IsAuthenticated` returns true; you will be using `IPIdenity` only with authenticated requests. `Name` returns the IP address string.

This is a pretty minimal `IIdenity` implementation. You can add other data to the identity class. For example, `FormsIdentity` provides access to the forms authentication ticket.

Building the HTTP Module

Now that you know what the IP authentication module is going to populate `Context.User` with, you can press on and build the HTTP module that will implement it (see Listing 14.1).

LISTING 14.1 An HTTP Module Implementation for IP-Based Authentication

```
Imports System.Security.Principal

Public Class IPAuthenticationModule
  Implements IHttpModule

  Public Sub Dispose() Implements System.Web.IHttpModule.Dispose
    'we have no resources to dispose of
  End Sub

  Public Sub Init(ByVal context As System.Web.HttpApplication) _
                      Implements System.Web.IHttpModule.Init
    'handle the AuthenticateRequest of the application
    AddHandler context.AuthenticateRequest, AddressOf Me.Authenticate
  End Sub
```

LISTING 14.1 Continued

```
  Public Sub Authenticate(ByVal sender As Object, ByVal e As EventArgs)
    Dim application As HttpApplication = CType(sender, HttpApplication)

    'create identity and principal objects
    Dim identity As New IPIdentity(application.Request.UserHostAddress)
    Dim principal As New GenericPrincipal(identity, New String() {})

    'attach the principal to the application context
    application.Context.User = principal

  End Sub

End Class
```

You don't need to do anything in the `Dispose` method because you use no resources that require disposal. You have to include it, though, because it is required by the `IHttpModule` interface.

In the `Init` method, you bind the `Authenticate` method of this class to the `AuthenticateRequest` event of the application. In the `Authenticate` method, you simply use the IP address of the request (`Request.UserHostAddress`) to create an `IPIdentity` object, which is then used to construct a `GenericPrincipal` instance that is stored in `Context.User`.

Before the module can be used, you need to ensure that no other authentication modules are active by setting the `<Authentication>` element of the `web.config` file appropriately:

```
<authentication mode="None" />
```

You also need to add an `<httpModules>` section to the `web.config` file to add the module to the application:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>
    <httpModules>
      <add type="CustomAuthentication.IPAuthenticationModule,
➥CustomAuthentication" name="IPAuthentication" />
    </httpModules>
```

After you have done this, you can access the IP address of the client through `Context.User.Identity.Name`. (This is not very impressive, admittedly, because you can access it through `Request.UserHostAddress` anyway.) More interestingly, you can use URL authorization by specifying the IP addresses as the usernames:

```
<authorization>
    <allow users="127.0.0.1" />
```

```
    <allow users="56.43.43.1" />
    <allow users="56.43.43.2" />
    <allow users="56.43.43.3" />
    <deny users="*" />
</authorization>
```

You could also have the `IPAuthenticationModule` implementation create the `GenericPrincipal` object with roles that reflect what sort of information should be displayed to that client. You would then be able to use `Context.User.IsInRole` or any of the other programmatic authorization techniques to access this information.

Running Authentication Modules in Tandem

The built-in authentication modules in ASP.NET can only be used one at a time, but there is no reason you cannot have additional custom authentication modules running on top of the built-in modules.

For example, you might want to use forms authentication to allow administrators to sign in to the system but use the `IPAuthenticationModule` implementation that you built in the previous section to identify the client machine for users who are not authenticated by the forms authentication module.

The important thing to do if you want to use more than one module together is to ensure that they do not clash with each other. You can change the `IPAuthenticationModule` implementation so that it will populate the `Context.User` object only if it has not already been populated by another authentication module (see Listing 14.2).

LISTING 14.2 Adapting `IPAuthenticationModule` to Allow It to Work with Other Modules

```
Public Sub Authenticate(ByVal sender As Object, ByVal e As EventArgs)
  Dim application As HttpApplication = CType(sender, HttpApplication)

  'check that the user has not been created or is not authenticated
  If application.Context.User Is Nothing _
        OrElse Not application.Context.User.Identity.IsAuthenticated Then

    'create identity and principal objects
    Dim identity As New IPIdentity(application.Request.UserHostAddress)
    Dim principal As New GenericPrincipal(identity, New String() {})

    'attach the principal to the application context
    application.Context.User = principal

  End If
End Sub
```

The IPAuthenticationModule implementation will then happily coexist with the forms authentication module. When forms authentication does not authenticate the user, the IPAuthenticationModule implementation will populate the Context.User object. When forms authentication does pick up a ticket and authenticate the user, the IPAuthenticationModule implementation will do nothing.

This system will work best when forms authentication is set up to support role-based authorization, so that the administrators or other privileged users can be placed in a role.

The example shown here is very simple, but it shows all the features that are required from an authentication module. If you have some authentication requirements that are not served by the default modules, you should be able to build a custom module to do the job.

Building a Custom Authorization Module

Authorization is the process of deciding whether the current user has permission to access the resource that he or she requested. The authorization modules that ship with ASP.NET decide whether the resource can be accessed by either checking Windows access control lists (file authorization) or checking the <authorization> element of the configuration file (URL authorization).

There are lots of other possibilities for authorizing the requests of users. For example, you might allow access only at certain times, you might require users to have been registered with the application for a certain amount of time before accessing some content, or you might assign each user credits that he or she can use to access pay-per-view content. Or you might use a single-page application architecture and want to authorize based on the URL parameters of the request.

The best way to implement custom authorization behavior is by creating an authorization module. Like authentication modules, authorization modules are HTTP modules (although they hook into the AuthorizeRequest event rather than the AuthenticateRequest event).

Listing 14.3 shows a simple authorization module that checks the expiration date in a custom identity against the current date.

LISTING 14.3 A Custom Authorization Module

```
Public Class ExpirationAuthorizationModule
  Implements System.Web.IHttpModule

  Public Sub Dispose() Implements System.Web.IHttpModule.Dispose
    'nothing to dispose of
  End Sub

  Public Sub Init(ByVal context As System.Web.HttpApplication)
➥Implements System.Web.IHttpModule.Init
    AddHandler context.AuthorizeRequest, AddressOf Me.Authorize
```

LISTING 14.3 Continued

```
  End Sub

  Private Sub Authorize(ByVal sender As Object, ByVal e As EventArgs)
    Dim application As HttpApplication = CType(sender, HttpApplication)

    If application.Context.Request.IsAuthenticated Then
      If Not application.Context.User.IsInRole("NonExpiring") Then

        Dim identity As ExpiringIdentity = _
                CType(application.Context.User.Identity, ExpiringIdentity)

        If identity.Expires < DateTime.Now Then
          'the users registration has expired
          application.Context.Response.Redirect("RegistrationExpired.aspx")
        End If
      End If
    End If
  End Sub
End Class
```

The infrastructure of this authorization module is very similar to that of the custom authentication module discussed earlier in this chapter in the section, “What Is an Authentication Module?”

In the `Authorize` event handler, you check whether the user is authenticated. (This module is designed to only check for expired membership; it does not deny authorization to anonymous users. It is assumed that URL authorization would be used to do that.)

Next, you check that the user is not in the `NonExpiring` role. If he or she is, you do nothing more, and the user is allowed to view the resource he or she requested. If the user is not in the `NonExpiring` role, you check the current date and time against the user’s expiration date and time from the `ExpiringIdentity` object in `Context.User.Identitiy`. If the user has expired, you redirect the response to the “registration expired” page.

You may be wondering what the `ExpiringIdentity` class looks like. It is shown in Listing 14.4.

LISTING 14.4 The `ExpiringIdentity` Class

```
Public Class ExpiringIdentity
  Implements System.Security.Principal.IIdentity

  Private _Username As String
  Private _MembershipExpires As DateTime

  Public Sub New(ByVal username As String, ByVal expires As DateTime)
```

LISTING 14.4 Continued

```
    _Username = username
    _MembershipExpires = expires
  End Sub

  Private Sub New()
  End Sub

  Public ReadOnly Property AuthenticationType() As String _
        Implements System.Security.Principal.IIdentity.AuthenticationType
    Get
      Return "Custom"
    End Get
  End Property

  Public ReadOnly Property IsAuthenticated() As Boolean _
        Implements System.Security.Principal.IIdentity.IsAuthenticated
    Get
      Return True
    End Get
  End Property

  Public ReadOnly Property Name() As String _
         Implements System.Security.Principal.IIdentity.Name
    Get
      Return _Username
    End Get
  End Property

  Public ReadOnly Property Expires() As DateTime
    Get
      Return _MemberShipExpires
    End Get
  End Property
End Class
```

Running Authorization Modules in Tandem

As with authentication modules, you can run multiple authorization modules at the same time. You do not have to worry about clashing authorization modules as you do with authentication modules. If any of the authorization modules in the chain results in failed authorization, it will take action ahead of any other modules. This is as it should be; when it comes to security, you should always pay attention to the test that fails rather than to the test that passes.

The example shown in Listing 14.4 is designed to work in tandem with URL authorization. URL authorization would be used to keep anonymous users from using the application, whereas `ExpirationAuthorizationModule` implementation would be used to deal with users whose registrations have expired.

Trust Levels

By default, ASP.NET applications run at full trust. This means that the .NET Framework places no limitations on what they can do, aside from the limitations imposed by the operating system on the account that ASP.NET runs under. This is not a huge problem if you trust all the developers who write code for all the ASP.NET applications that run on your server, but what if you do not?

Also, running all applications at full trust breaks the principle of least privilege—the idea that for the best possible security, code should be allowed only the permissions that it absolutely needs.

The solution is to force applications to run at less than full trust. ASP.NET ships with four different levels of trust and allows you to configure your own trust levels if you need to specify a particular set of permissions.

Using One of the Preconfigured Trust Levels

In addition to the full-trust mode that places no restrictions, four trust levels are provided with ASP.NET (see Table 14.1).

TABLE 14.1
The Four Trust Levels Provided with ASP.NET

Trust Level	Main Permissions
Minimal	Only the bare minimum that are required for ASP.NET to function
Low	Very limited access to the file system
Medium	Limited read/write access to the file system; some other permissions
High	Full access to the file system; most other permissions

Remember that the permissions granted by these trust levels do not allow the ASP.NET user account to access anything that it does not have operating system permissions for. Trust levels can only impose additional restrictions; they can never remove existing operating system restrictions.

Let's look at each trust level in more detail to see what permissions the levels provide.

The Minimal Preconfigured Trust Level

The minimal trust level only grants the permissions that are absolutely required for an ASP.NET application to run.

An ASP.NET application running at this trust level will not be able to do a lot. It can't do any file system input/output work, can't do any data access, can't access isolated storage, can't access the registry, and can't execute any code that requires reflection. In fact, applications at this trust level can't really do anything apart from read the HTTP request and write to the HTTP response.

Another restriction when running at this trust level (and low trust) is that it does not allow debugging. Attempting to run an application that is configured for debugging at this trust level will result in an error message. For this reason, you have to appropriately set the `<compilation>` element in the `web.config` file for applications that will run at minimal or low trust:

```
<compilation defaultLanguage="vb" debug="false" />
```

The Low Preconfigured Trust Level

The low trust level does not allow very many more permissions than minimal trust. At this level, the application can read (but not write) files that are within its application directory, and it can read and write isolated storage with a quota of 1MB.

Note that the low trust level has the same limitation on debugging that the minimal trust level has: If you configure an application with low trust for debugging, you will get an error.

The Medium Preconfigured Trust Level

Compared to the minimal and low trust levels, quite a few additional permissions are added at the medium trust level. Debugging is now allowed. You can read and write files within the application directory. You also have access to isolated storage with an effectively unlimited quota. An application can access a SQL Server database through the `SqlClient` classes and can even print to the default printer.

At this level, an application also gets read access to the following environment variables:

- `TEMP`
- `TMP`
- `USERNAME`
- `OS`
- `COMPUTERNAME`

Finally, at medium trust, an application gets more control over threads, the principal object, and remoting.

The medium trust level contains the permissions that most ASP.NET applications need to run.

The High Preconfigured Trust Level

At high trust, an application gets unrestricted access to most resources, including the file system, isolated storage, environment variables, the registry, and sockets. An application also gains the ability to generate dynamic assemblies by using `Reflection.Emit`.

Remember, though, that ASP.NET is still limited by the permissions that the account under which it runs has been given.

Forcing an Application to Use a Trust Level

As mentioned earlier in this chapter, the default trust level for ASP.NET applications is full trust. You can see this by finding the appropriate section of the `machine.config` file:

```
<location allowOverride="true">
  <system.web>
    <securityPolicy>
      <trustLevel name="Full" policyFile="internal"/>
      <trustLevel name="High" policyFile="web_hightrust.config"/>
      <trustLevel name="Medium" policyFile="web_mediumtrust.config"/>
      <trustLevel name="Low" policyFile="web_lowtrust.config"/>
      <trustLevel name="Minimal" policyFile="web_minimaltrust.config"/>
    </securityPolicy>
    <!--  level="[Full¦High¦Medium¦Low¦Minimal]" -->
    <trust level="Full" originUrl=""/>
  </system.web>
</location>
```

A `<location>` element without a path is used to apply the settings to all Web applications. The five trust levels are defined, with the filenames of the files that hold their policies. The trust level full is then configured. This will be the default for all applications running on the server.

You can change the default trust level by changing this configuration. If you do, you should change the `allowOverride` attribute of the `<location>` element to `false`, or individual applications will simply be able to define their own trust levels in their `web.config` files.

Often, though, you want to configure different trust levels for different applications. This is especially true if you want to follow the principle of least privilege and want to have each application run with only the permissions it needs.

You can take a number of approaches to this. You could remove the `<trust>` element from the machinewide `<location>` element and instead include a `<location>` element for each application. If you do this, you have to be careful to include one for each application, or you will get errors. (Every application must have a `<trust>` element at some level of its configuration.)

You can't simply define a default in the machinewide `<location>` element and then include additional `<location>` elements in the `machine.config` file to specify the trust levels for individual applications. You are not allowed to include the `<trust>` element twice in a single configuration file.

If you want to configure a default and then define different trust levels for certain applications, you have to use multiple configuration files. If you can't define the settings in the `machine.config` file, along with the machinewide default, and you can't define it in the `web.config` file for the specific application, you have to use a `web.config` file at the Web site level.

The following `web.config` file will, when added to the root folder of the Web server (usually `wwwroot`), specify that the application called `UntrustedApp` that is in the `InsiderSolutions` folder should be run at low trust:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!--configure a lower trust level for one application -->
    <location allowOverride="false" path="InsiderSolutions/UntrustedApp">
        <system.web>
            <trust level="Low" originUrl="http://www.UntrustedApp.com">
        </system.web>
    </location>
</configuration>
```

The `originUrl` attribute is used to specify the URL that should be used for the `WebPermission` permission, which allows code to make web requests (although note that in this case, it will not do anything, as the low trust level does not include the `WebPermission` permission).

You could also specify a trust level for all applications in the `InsiderSolutions` folder as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!--configure a lower trust level for one application -->
    <location allowOverride="false" path="InsiderSolutions ">
        <system.web>
            <trust level="Low" originUrl=""/>
        </system.web>
    </location>
</configuration>
```

Creating Custom Trust Levels

The trust levels that ship with ASP.NET provide a pretty good spread of permission sets, but if you want to have complete control over what applications are doing on your server, you need to define your own trust levels.

The best way to go about this is to start from one of the existing trust levels and use it as the basis for your new trust level. By copying its policy file and making changes, you can construct a new trust level with just the permissions you want. With this in mind, let's take a look at the policy file for the medium trust level to see how the policy files are structured.

The policy file for the medium trust level can be found in `Windows/Microsoft .NET/Framework/[`*`version`*`]Config/Web_MediumTrust.config`. The file has an overall structure like this:

```
<configuration>
  <mscorlib>
      <security>
```

```
        <policy>
          <policyLevel>

            <securityClasses>
              a <securityClass> for each permission,
              code group and condition class used by the file
            </securityClasses>

            <namedPermissionSets>
              a <permissionSet> for each permission set used in the policy:
              <permissionSet>
                a set of 0 or more <IPermission>
                elements to define the permissions in the set
              </permissionSet>
            </namedPermissionSets>

            <codeGroup>
              nested <codeGroup> elements that link code
              to permission sets based on conditions:
              <codeGroup>
                0 or more nested <codeGroup> elements and one  <IMembershipCondition>
              </codeGroup>
            </codeGroup>

          </policyLevel>
        </policy>
      </security>
  </mscorlib>
</configuration>
```

This file contains three main things: definitions of the security classes that will be used, some permission sets, and some nested code groups that link code to the permission set that it should run with.

For the purposes of changing the permissions that ASP.NET applications run with, you do not need to change the `<codeGroup>` elements; you just need to add or remove permissions from the `<permissionSet>` element that holds the permissions that are granted to the ASP.NET application.

The `Web_MediumTrust.config` file contains three permission sets: a full access set that grants unrestricted privileges, a set that grants no privileges, and a set with the permissions that are actually granted to the ASP.NET application.

The unrestricted privileges are granted to assemblies signed with either the Microsoft or the ECMA strong name (that is, code that Microsoft or ECMA says should be trusted).

The permission set that grants no permissions is used as the default for code that is not matched to any other permission set.

Listing 14.5 shows the <permissionSet> element from Web_MediumTrust.config that is granted to the code of the ASP.NET application itself.

LISTING 14.5 Permissions from the Medium Trust Level Configuration File

```
<PermissionSet
        class="NamedPermissionSet"
        version="1"
        Name="ASP.Net">
    <IPermission
            class="AspNetHostingPermission"
            version="1"
            Level="Medium"
    />
    <IPermission
            class="DnsPermission"
            version="1"
            Unrestricted="true"
    />
    <IPermission
            class="EnvironmentPermission"
            version="1"
            Read="TEMP;TMP;USERNAME;OS;COMPUTERNAME"
    />
    <IPermission
            class="FileIOPermission"
            version="1"
            Read="$AppDir$"
            Write="$AppDir$"
            Append="$AppDir$"
            PathDiscovery="$AppDir$"
    />
    <IPermission
            class="IsolatedStorageFilePermission"
            version="1"
            Allowed="AssemblyIsolationByUser"
            UserQuota="9223372036854775807"
    />
    <IPermission
            class="PrintingPermission"
            version="1"
            Level="DefaultPrinting"
    />
    <IPermission
            class="SecurityPermission"
```

LISTING 14.5 Continued

```
            version="1"
            Flags="Assertion, Execution, ControlThread, ControlPrincipal,
➥RemotingConfiguration"
    />
    <IPermission
            class="SqlClientPermission"
            version="1"
            Unrestricted="true"
    />
    <IPermission
            class="WebPermission"
            version="1">
        <ConnectAccess>
            <URI uri="$OriginHost$"/>
        </ConnectAccess>
    </IPermission>
</PermissionSet>
```

The permission set named "ASP.NET" is assigned to the code of the application in all the trust level files. It is quite easy to read the various permissions from the file, to see what the application will be allowed to do.

Removing permissions from the trust level is easy: You simply need to delete the IPermission element for the permission you want to remove. For example, if you do not want applications running at this trust level to have access to isolated storage, you would remove the element that sets that permission:

```
<IPermission
        class="IsolatedStorageFilePermission"
        version="1"
        Allowed="AssemblyIsolationByUser"
        UserQuota="9223372036854775807"
/>
```

Adding a permission is slightly more complicated. As well as working out what IPermission element you need to add, you have to add an appropriate <securityClass> element for the permission class you want to use.

Looking at a <SecurityClass> element from the file, you can see that you have to provide the full namespace and classname for the class, the assembly it is in, the version, the culture, and a public key token:

```
<SecurityClass Name="AllMembershipCondition" Description=
➥"System.Security.Policy.AllMembershipCondition, mscorlib,
➥ Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

Fortunately, there is a way to generate the <SecurityClass> elements you need—by using the .NET Framework Configuration tool. The tool does not allow you to edit the ASP.Net trust level policy files directly, but you can add permissions to a permission set in one of the files that the tool can edit and then copy them across to the trust level file.

To use the .NET Framework Configuration tool, you select Start, Control Panel, Administrative Tools, .NET Configuration. Then you open Runtime Security Policy and then the User branch of the tree-view and click the Permission Sets branch. You can then click the New Permission Set link in the main window to create a permission set, as shown in Figure 14.1.

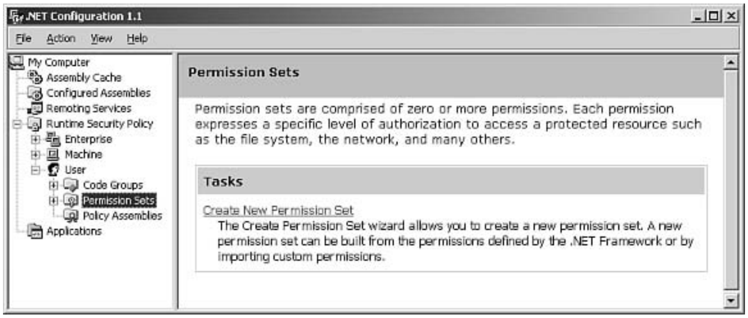


FIGURE 14.1
Creating a permission set in the .NET Framework Configuration tool.

Next, you enter a name (it doesn't really matter what name you use—you won't be using this permission set for anything). It would be best to give it a name and description that make it clear that this is a temporary set that you are using to create permissions and classes to copy across to ASP.NET configuration. Then you can click Next, and you will see the window where you can add permissions to the set (see Figure 14.2).

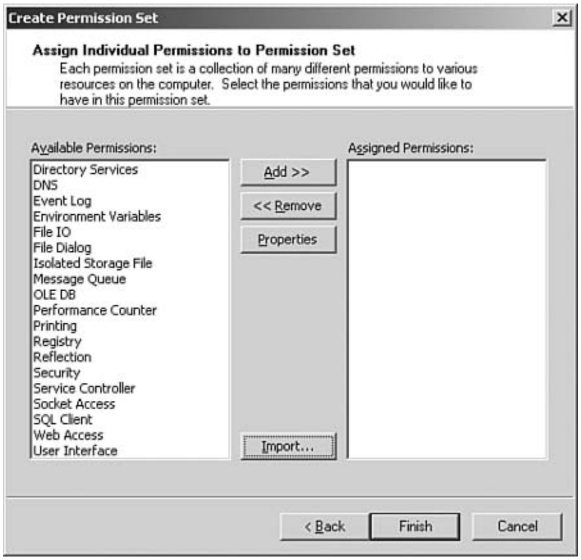


FIGURE 14.2
Adding permissions to a permission set.

On the Create Permission Set window, you simply move the permissions that you want to add over to the Assigned Permissions box on the right side. When a permission is added, a new window will open, with the relevant options for that permission. This is a great way to learn what permissions are available and what options are available for each one.

For example, if you add the OLE DB permission, which allows access to OLE DB modules, you see the options shown in Figure 14.3.

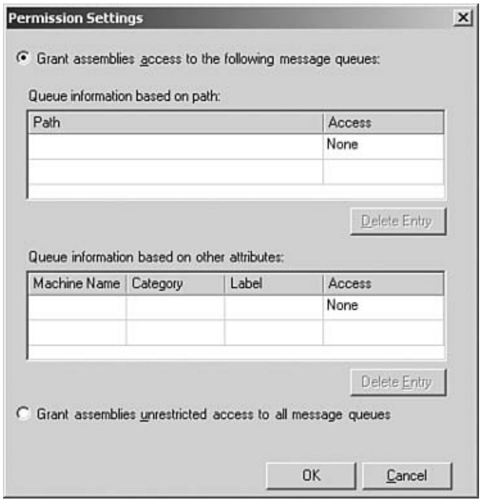


FIGURE 14.3 Options for the OLE DB permission.

After you set the settings you want, the file will be located in `Documents and settings/[`*`username`*`]/Application Data/Microsoft/CLR Security Config/[`*`version`*`]/security.config`. In this file, you will find the permission set that you have added:

```
<PermissionSet class="NamedPermissionSet"
               version="1"
               Name="tempForASPNET">
   <IPermission class="OleDbPermission"
                version="1"
                Unrestricted="true"/>
</PermissionSet>
```

You can simply copy the `<IPermission>` element from here into the Web trust level policy file that you want to add it to.

You also need to copy the matching `SecurityClass` element:

```
<SecurityClass Name="OleDbPermission"
               Description="System.Data.OleDb.OleDbPermission,
➥System.Data, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

After you have created a new trust level policy file, you need to add it to the set that is available for use, by editing the `<securityPolicy>` element of the `machine.config` file:

```
<location allowOverride="true">
  <system.web>
    <securityPolicy>
      <trustLevel name="Custom" policyFile="web_customTrust.config"/>
      <trustLevel name="Full" policyFile="internal"/>
      <trustLevel name="High" policyFile="web_hightrust.config"/>
      <trustLevel name="Medium" policyFile="web_mediumtrust.config"/>
      <trustLevel name="Low" policyFile="web_lowtrust.config"/>
      <trustLevel name="Minimal" policyFile="web_minimaltrust.config"/>
    </securityPolicy>
    <!--  level="[Full¦High¦Medium¦Low¦Minimal]" -->
    <trust level="Full" originUrl=""/>
  </system.web>
</location>
```

You can then specify this trust level as the default for the machine or for specific applications, as described earlier in this chapter.

You can use in the trust level policy files some special pieces of text that are not included in the .NET Configuration tool. These deal with situations in which each Web application needs to have a different setting for a particular permission. For example, you usually want to restrict the `FileIOPermission` permission to the application folder. The `FileIOPermission` permission from the `Web_MediumTrust.config` file shows how you do this:

```
<IPermission
        class="FileIOPermission"
        version="1"
        Read="$AppDir$"
        Write="$AppDir$"
        Append="$AppDir$"
        PathDiscovery="$AppDir$"
/>
```

The string `$AppDir$` is converted to the path to the application folder at runtime.

You can also use the following substitutions:

```
$OriginHost$ - the address of the client that made a request to the application
$AppDirUrl$ - the URL of the application
$CodeGen$ - The folder where ASP.NET stores dynamically generated assemblies
$Gac$ - the Global Assembly Cache folder
```

Note that the last three substitutions are usually used in code groups rather than in permissions.

Recommended Use of Permissions

Table 14.2 shows all the permissions that can be used, along with some recommendations for how to apply them.

TABLE 14.2
The Allowed Permissions

Permission	Recommendations
`DirectoryServicesPermission`	You should grant this permission if the application needs to access Active Directory or LDAP data.
`DnsPermission`	You should grant this permission if the application needs to resolve domain names to IP addresses. This is a pretty safe permission to grant.
`EventLogPermission`	You should grant this permission if the application needs to read or write to event logs (on the machine it is running on or another machine). Note that the ASP.NET account will need permission to access the event log as well.
`EnvironmentPermission`	You should grant this permission if the application needs to access environment variables. It is usually advisable to grant read access only; it is very rare that a Web application should need to set environment variables.
`FileIOPermission`	You should use this permission to allow access to files. It is usually used with the `$AppDir` substitution to allow access to the application's own folder, but it can also be used to allow access to other specific locations.
`FileDialogPermission`	This permission is not really relevant to Web applications and so should not be granted to them. It allows Windows Forms applications to create open and save dialogs.
`IsolatedStoragePermission`	You should grant this permission if the application needs to use isolated storage. The `AssemblyIsolatedByUser` option will allow access to the data from different applications (provided that they do it through the same assembly), whereas `DomainIsolatedByUser` will allow access only from the application that created the data.
`MessageQueuePermission`	You should grant this permission if the application needs to interact with message queues.
`OleDbPermission`	You should grant this permission if the application needs to use OLE DB data sources. You can grant access to all data modules or list specific modules. This is a common permission to grant.
`PerformanceCounterPermission`	You should grant this permission if the application needs to read (browse) or write (instrument) performance counters. You can specify permissions for counters on specific machines and also for specific categories of counters on a machine.
`PrintingPermission`	It is unlikely that a Web application will need to access printers, so it is best not to grant this permission.
`RegistryPermission`	This can be a highly dangerous permission to grant, so treat it with caution. (However, the risks are mitigated by the limitations that the operating system places on the ASP.NET account.)
`ReflectionPermission`	You should grant this permission if the application needs to be able to perform reflection-based operations on other assemblies. Quite a lot of .NET Framework applications now make use of reflection, so this permission is likely to be in demand. However, this permission is not required for reflection within an assembly.

TABLE 14.2
Continued

Permission	Recommendations
`SecurityPermission`	This permission has a myriad of options. Think carefully before granting any of these because many of them can allow code to circumvent code access security checks in one way or another.
`ServiceControllerPermission`	You should grant this permission if the application needs to connect to Windows services in order to work. The Browse option is sufficient to connect to services. If the application needs the ability to start and stop services, it will need the Control option.
`SocketAccessPermission`	You should grant this permission if the application needs to perform low-level networking tasks.
`SQLClientPermission`	You should grant this permission if the application needs to access SQL Server data. This is a common permission to grant.
`WebPermission`	You should grant this permission if the application needs to make Web requests of its own. This is actually quite rare (most Web applications respond to requests rather than make their own requests) but can be useful sometimes.
`UserInterfacePermission`	This permission is typically not granted to Web applications because it is really only relevant to Windows Forms applications.

A Permission Set for Normal Use

For most ASP.NET applications, the medium built-in trust level is ideal. If you want to run things with the least privileges (which you should where possible), you will probably want to remove some of the permissions that your application does not require.

If your application does not need access to environment variables, you can safely get rid of `EnvironmentPermission`.

You can get rid of `IsolatedStoragePermission` if your application does not use isolated storage.

Most Web applications can dispense with `PrintingPermission`.

If the application does not need to make Web requests of its own, you can remove `WebPermission`.

Implementing all these changes in a custom trust level would leave something like the permission set shown in Listing 14.6.

LISTING 14.6 A Permission Set Based on the Medium Trust Level, with Unnecessary Permissions Removed

```
<PermissionSet
        class="NamedPermissionSet"
        version="1"
        Name="ASP.Net">
    <IPermission
            class="AspNetHostingPermission"
            version="1"
            Level="Medium"
    />
```

LISTING 14.6 Continued

```
    <IPermission
            class="DnsPermission"
            version="1"
            Unrestricted="true"
    />
    <IPermission
            class="FileIOPermission"
            version="1"
            Read="$AppDir$"
            Write="$AppDir$"
            Append="$AppDir$"
            PathDiscovery="$AppDir$"
    />
    <IPermission
    <IPermission
            class="SecurityPermission"
            version="1"
            Flags="Assertion, Execution, ControlThread, ControlPrincipal,
➥RemotingConfiguration"
    />
    <IPermission
            class="SqlClientPermission"
            version="1"
            Unrestricted="true"
    />
</PermissionSet>
```

You might also want to make some changes to the settings of `FileIOPermission`. If you do not need to read or write any data to the file system, you can remove it completely. (The ASP.NET infrastructure code in the .NET Framework will still be able to work because it is signed by the Microsoft strong name and therefore gets full trust.)

You can probably remove the ability for the application to discover paths because browsing the file system is not usually a requirement for Web applications.

If your application only reads data (for example, an XML configuration file), you can remove the Write and Append attributes, too:

```
<IPermission
        class="FileIOPermission"
        version="1"
        Read="$AppDir$"
/>
```

If your application only needs to read and write data within a specific subfolder of your application, you can specify the folder:

```
<IPermission
        class="FileIOPermission"
        version="1"
        Read="$AppDir$/data"
        Write="$AppDir$/data"
        Append="$AppDir$/data"
/>
```

You can also configure a folder that is not under the application folder if that is where you store your data:

```
<IPermission
        class="FileIOPermission"
        version="1"
        Read="d:\webdata"
        Write="d:\webdata"
        Append="d:\webdata"
/>
```

If you need to specify more than one path, you can use semicolons to separate the paths in each attribute where you want them:

```
<IPermission
        class="FileIOPermission"
        version="1"
        Read="$AppDir$/data;d:\sharedData"
        Write="$AppDir$/data"
        Append="$AppDir$/data"
/>
```

This example would allow read and write access to the application folder but only read access to `d:\sharedData`.

Summary

ASP.NET provides a high degree of flexibility in the way that security is managed. This chapter looks at some things you can do, both with code and with configuration, when the security behavior of ASP.NET is not exactly what you require.

The chapter shows how to build custom authentication modules when the modules provided by Microsoft do not provide the features you need. A sample module is presented and explained. Custom authorization modules are also covered.

Finally, the chapter looks at how .NET Framework code access security can be applied to ASP.NET, through trust levels. The standard trust levels are explained, and the process of building custom trust levels is shown.

Index

A

B

C

E

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W

X-Y-Z