

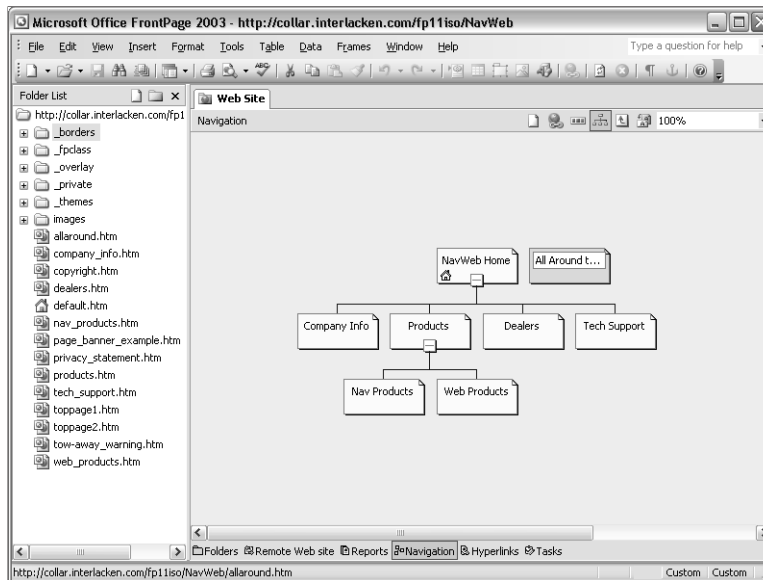
## Insider Extra 13

# Creating Your Own Statistics Pages

Given that Top 10 List components display much less data than the Usage reports available in Reports view, an inquiring mind might question where Microsoft Office FrontPage 2003 stores the summarized usage data and whether a Web page could display it. This chapter answers those questions.

## Locating Usage Data

The root folder of every FrontPage-based Web site contains a `_vti_pvt` subfolder where FrontPage routinely keeps system files. This folder won't appear in FrontPage even if you select the Show Hidden Files And Folders setting, but if you have access to the Web server's file system, you can see it in Windows Explorer. Figure IE13-1 shows the proof. (Be sure to select the folder option Show Hidden Files And Folders before trying to display the `_vti_pvt` folder on your own system.)



**Figure IE13-1.** The XML files in the `_vti_pvt` folder of this Web site contain FrontPage usage data. This folder is inaccessible to both browsers and FrontPage.

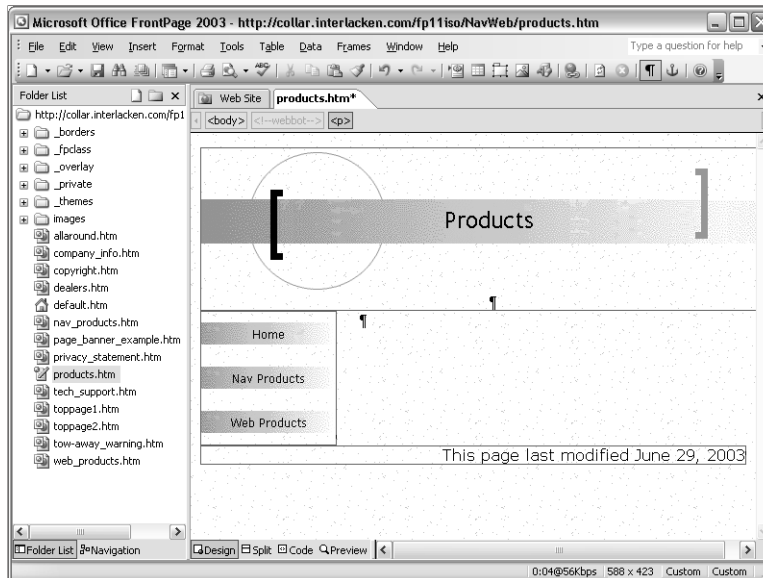
## Microsoft Office FrontPage 2003 Inside Out

When the FrontPage Server Extensions run Usage Analysis, they store the results in a series of XML files located in the `_vti_pvt` subfolder of each FrontPage Web site. Table IE13-1 lists the names of these files.

**Table IE13-1. Usage Report Data Files**

File name	Description
<code>_x_browsers.xml</code>	Browsers
<code>_x_domains.xml</code>	Visiting hosts
<code>_x_pagehits.xml</code>	Visited pages
<code>_x_referrers.xml</code>	Referring URLs
<code>_x_refdomains.xml</code>	Referring domains
<code>_x_systems.xml</code>	Operating system
<code>_x_users.xml</code>	Visiting users

You've probably noticed the `.xml` file extensions in Table IE13-1. XML, short for Extensible Markup Language, is an emerging standard for storing and exchanging information in a universal way. XML and HTML are related, but not very closely. Whereas HTML concerns itself mostly with format and presentation, XML concerns itself entirely with data. Opening the `_x_browsers.xml` file in Microsoft Internet Explorer, for example, produces the results shown in Figure IE13-2.



**Figure IE13-2.** Because XML files contain no formatting information, Internet Explorer displays them as is.

## Creating Your Own Statistics Pages

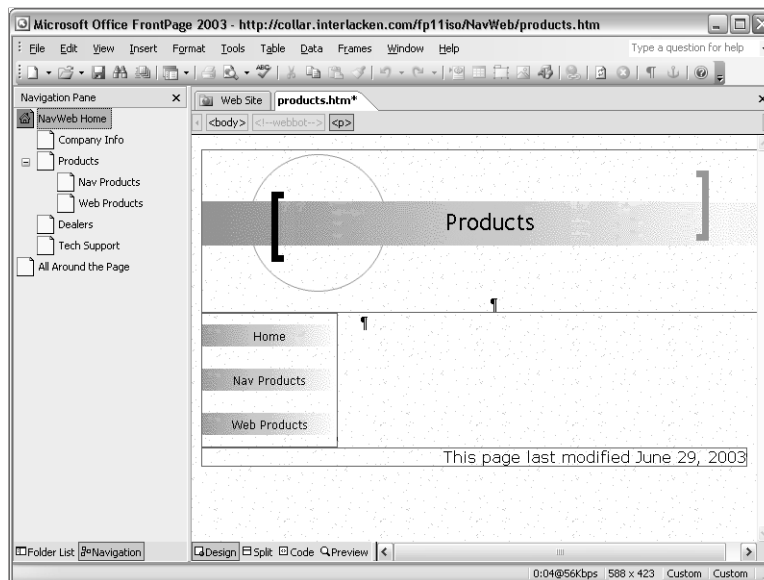
As this figure illustrates, XML, like HTML, makes extensive use of tags enclosed by angle brackets (<>). In the case of XML, however, the tags contain data identifiers (field names) rather than formatting codes.

- A pair of <usagedata>...</usagedata> tags, for example, marks the beginning and end of the data portion of the file.
- Pairs of <month>...</month> tags enclose the data for each month.
- <item> and </item> tags mark the bounds of each detail record.
- The actual data values are coded much like the attributes in an HTML tag.

Writing a script that reads this file as text and parses out its contents would cause quite a headache. Fortunately, there are plenty of free software programs around that do this job very easily. They're called *parsers*, and Microsoft provides them for both ASP and ASP.NET pages. The next two sections will describe ASP and ASP.NET pages that parse and report the XML-based FrontPage usage data.

## Displaying Usage Data with ASP

This Insider Extra displays all available FrontPage browser statistics in an ASP Web page. Figure IE13-3 shows how the finished page looks in a browser.



**Figure IE13-3.** This Web page reports usage data that the Microsoft FrontPage Server Extensions extracted from Web server log files and stored in XML format.

## Microsoft Office FrontPage 2003 Inside Out

Because this is an ASP page, and because it makes use of FrontPage usage information, the page requires the following environment:

- The page must reside in a server-based Web site.
- The server-based Web site must reside in a Microsoft Windows 2000, Windows XP Professional, or Windows Server 2003 system with IIS installed.
- The folder where the ASP page resides must be marked as executable.
- The Web server must be running the FrontPage Server Extensions 2002. (Recall that ASP pages won't run on Web servers running Microsoft Windows SharePoint Services.)
- The Web site server must be configured to capture usage information.

For programs like ASP pages that use Microsoft ActiveX controls, Microsoft provides an XML parser named *Microsoft.XMLDOM*. You can use a *Microsoft.XMLDOM* object either at the browser or in an ASP script. This example uses an ASP script because FrontPage locks browsers out of the `_vti_pvt` folder where the necessary data resides.

Here's the VBScript code that creates a *Microsoft.XMLDOM* object and loads it with the contents of the `_x_browsers.xml` file:

```
Set xmlDoc = Server.CreateObject("Microsoft.XMLDOM")
xmlDoc.async = False
xmlDoc.Load (Server.MapPath(GetAppPath() & "_vti_pvt/_x_browsers.xml"))
```

In this code:

- The first statement creates a *Microsoft.XMLDOM* object named *xmlDoc*.
- The second statement configures the *xmlDoc* object to suspend script execution while it loads an XML file.
- The third statement loads the *xmlDoc* object with the file located at the application root plus `_vti_pvt/_x_browsers.xml`. The *Server.MapPath* method converts this URL to a physical file path.

Because the second statement sets the *async* property to *False*, any statements after the *xmlDoc.load* statement won't execute until the *xmlDoc* object succeeds in loading the file or gives up.

So much for getting data into an XMLDOM object. Now, how do you get it out? All *XMLDOM* objects contain a *documentElement* object that represents the highest level of the XML data structure. In the XML document shown in Figure IE13-2, *documentElement* points to the `<usedata>` node. So, with that document loaded into the *XMLDOM* object named *xmlDoc*, the following expression provides a path to whatever you want to know about the `<usedata>` node:

```
xmlDoc.documentElement
```

## Creating Your Own Statistics Pages

All the `<month>` nodes are children of the `<usedata>` node (that is, they appear between the `<usedata>` tag at the beginning of the file and the `</usedata>` tag at the end). The collection of all such nodes resides in an object named `ChildNodes`. The paths to the three `<month>` nodes in Figure IE13-2 are shown here:

```
xmlDoc.documentElement.childNodes.Item(0)
xmlDoc.documentElement.childNodes.Item(1)
xmlDoc.documentElement.childNodes.Item(2)
```

Here are a few more facts about this data structure:

- The following expression returns the number of `Item` nodes in a `ChildNodes` collection:

```
xmlDoc.documentElement.childNodes.Length
```

- The `GetAttribute` method returns the value of any attribute coded within a node. The following expression, for example, would return “*apr 2003*”:

```
xmlDoc.documentElement.childNodes.Item(0).getAttribute("val")
```

- Because expressions such as this are long and tiresome to code, experienced programmers often set up aliases to parts of the structure that they plan to access frequently. The first statement that follows, for example, sets up an alias named `xmlHdrData` that the programmer can use instead of the longer expression `xmlDoc.documentElement.childNodes`. The second statement is therefore equivalent to the longer version just shown:

```
Set xmlHdrData = xmlDoc.documentElement.childNodes
xmlHdrData.Item(0).getAttribute("val")
```

You now know everything necessary to code a loop that traverses the `xmlDoc` document and displays all three pairs of `val` and `total` values. After testing and debugging, your results should look very much like the following:

```
1 Set xmlHdrData = xmlDoc.documentElement.childNodes
2 For hPos = 0 To xmlHdrData.Length - 1
3   cnt = xmlHdrData.Item(hPos).getAttribute("total")
4   mon = xmlHdrData.Item(hPos).getAttribute("val")
5   Response.Write "<tr>" & vbCrLf & _
6     "<td>" & mon & "</td>" & vbCrLf & _
7     "<td>Total</td>" & vbCrLf & _
8     "<td>" & cnt & "</td>" & vbCrLf & _
9     "</tr>" & vbCrLf
10 Next
```

In this code:

- The statement on line 1 sets up *xmlHdrData* as an alias to *xmlDoc.documentElement.childNodes*.
- The statement on line 2 initiates a loop that varies *hPos* from zero to 1 less than the number of *Item* nodes in *xmlHdrData*. (If there are three such nodes, the loop varies *hPos* from 0 to 2).
- The statements on lines 3 and 4 retrieve the values of the *total* and *val* attributes from the current node, storing them in two variables named *cnt* and *mon*.
- The statement on lines 5 through 9 writes the *cnt* and *mon* values into one row of an HTML table.
- The last statement marks the end of the loop.

Compared to the easy job of displaying the `<month>` node data, displaying the `<item>` node data is downright trivial. The following code goes just before the *Next* statement in the previous loop:

```

1 Set xmlDetData = xmlHdrData.Item(hPos).childNodes
2 For dPos = 0 To xmlDetData.Length - 1
3   cnt = xmlDetData.Item(dPos).getAttribute("cnt")
4   bsr = xmlDetData.Item(dPos).getAttribute("str")
5   Response.Write "<tr>" & vbCrLf & _
6     "<td>&nbsp;</td>" & vbCrLf & _
7     "<td>" & bsr & "</td>" & vbCrLf & _
8     "<td align=right>" & cnt & "</td>" & vbCrLf & _
9     "</tr>" & vbCrLf
10 Next

```

In this code:

- The statement on line 1 sets up *xmlDetData* as an alias to *xmlHdrData.Item(hPos).childNodes*.

Just as *xmlHdrData* pointed to all children of the `<usedata>` node, *xmlDetData* now points to all children of the current `<month>` node. The alias *xmlDetData* is also equivalent to the following expression:

```
xmlDoc.documentElement.childNodes.Item(hPos).childNodes
```

- The statement on line 2 initiates a loop that varies *dPos* from zero to 1 less than the number of *Item* nodes in *xmlDetData*. (If there are two such nodes, the loop varies *hPos* from 0 to 1).
- The statements on lines 3 and 4 retrieve the values of the *total* and *val* attributes from the current node, storing them in two variables named *cnt* and *bsr*.
- The statement on lines 5 through 9 writes the *cnt* and *mon* values into one row of an HTML table.
- The last statement marks the end of the loop.

## Creating Your Own Statistics Pages

To see how all this fits together in a Web page, carry out the following procedure:

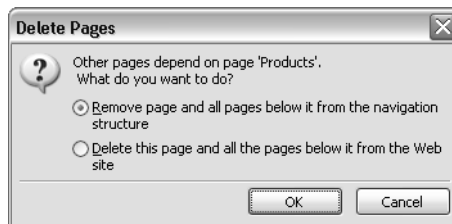
- 1 Open the Insider Extra Web site you installed from the companion CD. The Sample Files setup program installs this site at [My Documents]\Microsoft Press\FrontPage 2003 Inside Out\fp11extras.
- 2 Open the Web page usage/custbrow.asp.
- 3 Click the Code tab at the bottom of the Design view window.

By the way, all seven files listed in Table IE13-1 use the same format. To display a different report, simply change the file name in the following statement to another file name from Table IE13-1:

```
xmlDoc.Load (Server.MapPath("../vti_pvt/_x_browsers.xml"))
```

## Displaying Usage Data with ASP.NET

This Insider Extra repeats the challenge of the previous section, this time with an ASP.NET Web page. Figure IE13-4 shows this version of the Web page displayed in Internet Explorer.



**Figure IE13-4.** ASP.NET pages can also report FrontPage usage data. This version includes a drop-down list for selecting the category of usage data you want.

Compared to the ASP page in the previous section, this version has one additional feature: a drop-down list that you can use to select the category of statistics you want. To view the code that makes all this possible, open the Insider Extra Web site you installed from the companion CD, open the usage/custbrow.aspx page, and switch to Code view.

The custbrow.aspx page begins with the following directives. The first tells ASP.NET to compile the subsequent code with Microsoft Visual Basic .NET. The second and third are purely cosmetic; they provide access to ASP.NET custom control versions of a page banner and a page footer.

```
<%@ Page Language="vb"%>

<%@ Register TagPrefix="fp11iso" TagName="banner" Src="../banner.ascx"%>

<%@ Register TagPrefix="fp11iso" TagName="footer" Src="../footer.ascx"%>

<%@ Import Namespace="System.IO"%>

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

The two *@Import* declarations provide access to objects and methods for manipulating files and data.

The body of the Web page contains the following ASP.NET server controls:

- An ASP.NET form server control:
- Two tags that display the page banner and page footer:
- An ASP.NET DropDownList server control named *ddlFiles*:

```
<form id="Form1" method="post" runat="server">
</form>
```

```
<fp11iso:banner runat="server" />
<fp11iso:footer runat="server" />
```

```
<asp:dropdownlist id="ddlFiles" runat="server"
  AutoPostBack="True"
  onSelectedIndexChanged="ddlFiles_SelectedIndexChanged">
</asp:dropdownlist>
```

This tag includes two special attributes:

- The *AutoPostBack* attribute tells ASP.NET that whenever the Web visitor changes the drop-down list selection, the form should immediately submit its input to the Web server.
  - The *onSelectedIndexChanged* attribute tells ASP.NET what subroutine to run after the Web visitor changes the selection (that is, after the form submits its input).
- An HTML table named *tblCounts*. The Web page will display the usage data by adding rows to this table.

```
<table id="tblCounts" runat="server">
  <tr>
    <th>Period</th>
    <th>Item</th>
    <th>Count</th>
  </tr>
</table>
```

## Creating Your Own Statistics Pages

- A single code declaration block located immediately after the directives:

```
<script runat="server">
</script>
```

As in most ASP.NET pages, the *Page\_Load* subroutine is where most of the action begins. To code this subroutine, proceed as follows:

- 1 Within the code declaration block, define a subroutine named *Page\_Load* that expects the arguments that ASP.NET requires:

```
Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
End Sub
```

- 2 Within this subroutine, declare a *DirectoryInfo* object named *dinVtiPvt* and a *FileInfo* object named *finVtiPvt*:

```
Dim dinVtiPvt As DirectoryInfo
Dim finVtiPvt As FileInfo
```

- 3 Next make sure that the page isn't running because of a postback:

```
If Not Page.IsPostBack Then
End If
```

A *postback* occurs when an ASP.NET form submits input to the page that creates that form. The form submission that occurs after the visitor changes the list box is a postback, but initial display of the form is not.

- 4 Within this condition, translate the root folder of the current ASP.NET application to a physical path, and combine that result with the folder name *\_vti\_pvt\*.

Then create a new *DirectoryInfo* object that describes the folder at that location, and store its address in the *dinVtiPvt* variable. Here's the code:

```
dinVtiPvt = New DirectoryInfo(Path.Combine( _
    Server.MapPath(Request.ApplicationPath), _
    "_vti_pvt\"))
```

- 5 Next iterate through all the files in the *DirectoryInfo* object *dinVtiPvt*, selecting those with names beginning with *\_x\_* and ending with *.xml*.

```
For Each finVtiPvt In dinVtiPvt.GetFiles
    If (LCase(Left(finVtiPvt.Name, 3)) = "_x_") _
        And (LCase(finVtiPvt.Extension) = ".xml") Then
    End If
Next
```

## Microsoft Office FrontPage 2003 Inside Out

- 6 For each file name that satisfies these conditions, add an item to the *ddlFiles* drop-down list. The text should be the current file name, discarding the first three characters (*\_x\_*) and the file extension.

```
ddlFiles.Items.Add(Mid( _
    Path.GetFileNameWithoutExtension( _
        finVtiPvt.Name), 4))
```

- 7 By default, the display value and the submitted value of a drop-down list item are the same. To make the submitted value contain the entire file name, add the following statement next in sequence:

```
ddlFiles.Items(ddlFiles.Items.Count - 1).Value = _
    finVtiPvt.Name
```

- 8 This completes the job of loading the drop-down list with one entry for each usage file in the current Web site. To query the first such file, if any, add the following code after the *Next* statement you created in step 5. The next procedure in this section will explain how to create the *QueryStats* subroutine.

```
If ddlFiles.Items.Count > 0 Then
    ddlFiles.SelectedIndex = 0
    QueryStats()
End If
```

This completes the code for the *Page\_Load* subroutine. A complete listing of this subroutine is shown here:

```
Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
    Dim dinVtiPvt As DirectoryInfo
    Dim finVtiPvt As FileInfo
    If Not Page.IsPostBack Then
        dinVtiPvt = New DirectoryInfo(Path.Combine( _
            Server.MapPath(Request.ApplicationPath), _
                "_vti_pvt\"))
        For Each finVtiPvt In dinVtiPvt.GetFiles
            If (LCase(Left(finVtiPvt.Name, 3)) = "_x_") _
                And (LCase(finVtiPvt.Extension) = ".xml") Then
                ddlFiles.Items.Add(Mid( _
                    Path.GetFileNameWithoutExtension( _
                        finVtiPvt.Name), 4))
                ddlFiles.Items(ddlFiles.Items.Count - 1).Value = _
                    finVtiPvt.Name
            End If
        Next
        If ddlFiles.Items.Count > 0 Then
            ddlFiles.SelectedIndex = 0
            QueryStats()
        End If
    End If
End Sub
```

## Creating Your Own Statistics Pages

The `ddlFiles_SelectedIndexChanged` subroutine that runs whenever the Web visitor changes the drop-down list selection is refreshingly simple. Here it is:

```
Sub ddlFiles_SelectedIndexChanged( _
    ByVal sender As Object, ByVal e As EventArgs)
    QueryStats()
End Sub
```

This subroutine runs the same `QueryStats` subroutine that the `Page_Load` subroutine runs. The `QueryStats` subroutine gets the file name that corresponds to the current list selection, opens the file, and displays any data it contains. To code this beauty, proceed as follows:

- 1 At the end of the code declaration block (just before the `</script>` tag), define a subroutine named `QueryStats` that expects no arguments. Here's the code:

```
Sub QueryStats()
End Sub
```

- 2 Within this subroutine, declare the following variables:

```
Dim dstUsage As DataSet
Dim rdrUsage As StreamReader
Dim tblUsage As DataTable
Dim rowUsage As DataRow
Dim intHpos As Integer
Dim intDpos As Integer
Dim rowDets As DataRow()
Dim relDetails As DataRelation
Dim rowCounts As HtmlTableRow
Dim cellCounts As HtmlTableCell
Dim strRelName As String
```

- 3 Create a new `DataSet` object, assigning an internal name of `Usage` and saving its address in the `dstUsage` variable:

```
dstUsage = New DataSet("Usage")
```

- 4 Combine the following items:

- The physical path to the ASP.NET application's root folder
- The folder name `_vti_pvt`
- The file name from the current selection in the `ddlFiles` drop-down list

Use the result to open a `StreamReader` object. Save this object's address in the `rdrUsage` variable. Here's the code:

```
rdrUsage = New StreamReader( _
    Path.Combine(Server.MapPath(Request.ApplicationPath), _
        "_vti_pvt\" & ddlFiles.SelectedItem.Value))
```

## Microsoft Office FrontPage 2003 Inside Out

- 5** Load the *DataSet* object *dstUsage* from the XML in the *rdrUsage* stream, and then close the stream:

```
dstUsage.ReadXml(rdrUsage)
rdrUsage.Close()
```

- 6** Use the *tblUsage* variable to store the address of the first table in the *DataSet* object *dstUsage*. This is strictly for convenience.

```
tblUsage = dstUsage.Tables(0)
```

- 7** Code a loop that iterates through each row in the *tblUsage* table.

```
For intHpos = 0 To tblUsage.Rows.Count - 1
Next
```

- 8** Within this loop, test the *intHpos* counter to determine whether this is the second or later iteration. If it is, add a row to the *tblCounts* table. This row should contain one cell, with a *ColSpan* value of 3, and should contain a horizontal rule (an `<hr>` tag). Here's the required code:

```
If intHpos > 0 Then
    rowCounts = New HtmlTableRow()
    cellCounts = New HtmlTableCell()
    cellCounts.InnerHtml = "<hr>"
    cellCounts.ColSpan = 3
    rowCounts.Cells.Add(cellCounts)
    tblCounts.Rows.Add(rowCounts)
End If
```

This code first creates an *HtmlTableRow* object and an *HtmlTableCell* object in memory. It then sets the cell's *InnerHtml* property to "`<hr>`" and its *ColSpan* property to 3. Finally, it adds the cell to the row, and the row to the table.

This sort of manipulation might seem peculiar at first. Remember, however, that when an ASP.NET page is running, these objects reside only in the Web server's memory, and not on the Web visitor's screen.

- 9** For convenience, save the address of the current *DataRow* object in the *rowUsage* variable:

```
rowUsage = tblUsage.Rows(intHpos)
```

- 10** Displaying the first level of usage statistics requires three table cells. To create these:

- 1** Create a new table row.
- 2** Create, and configure the first table cell, and then add it to the new row.
- 3** Repeat this for the second and third table cells.
- 4** Add the new table row to the *tblCounts* table.

## Creating Your Own Statistics Pages

The first cell contains the `Val` column from the current `DataRow` object. The second cell contains the word *Total*, and the third cell contains the `Total` column from the current `DataRow` object. The following code accommodates all of these requirements and adds some formatting details:

```
rowCounts = New HtmlTableRow()
cellCounts = New HtmlTableCell()
cellCounts.InnerHtml = _
    "<b>" & rowUsage("Val") & "</b>"
rowCounts.Cells.Add(cellCounts)
cellCounts = New HtmlTableCell()
cellCounts.InnerHtml = "<b>Total</b>"
cellCounts.Align = "center"
rowCounts.Cells.Add(cellCounts)
cellCounts = New HtmlTableCell()
cellCounts.InnerHtml = _
    "<b>" & rowUsage("Total") & "</b>"
cellCounts.Align = "right"
rowCounts.Cells.Add(cellCounts)
tblCounts.Rows.Add(rowCounts)
```

- 11** The details statistics for each row exist as child structures within that row. Retrieving these children requires a *relationship name*, which is something you might not know. To find this name, write a loop that iterates through each relationship in the current row and saves its name in the `strRelName` variable. Here's the code:

```
For Each relDetails In tblUsage.ChildRelations
    strRelName = relDetails.RelationName.ToString
Next
```

- 12** Within this loop (that is, for each relationship), retrieve a collection of all matching rows and store a pointer to that collection in the `rowDets` variable. Then iterate through each member of the collection. Add the following code just before the `Next` statement from the previous step:

```
rowDets = rowUsage.GetChildRows(strRelName)
For intDpos = 0 To UBound(rowDets)
Next
```

- 13** For each row the loop in step 12 detects, add another row (containing three cells) to the *tblCounts* table. The first cell should be blank. The second cell should contain the *str* value from the current row, and the third cell should contain the *cnt* value from that row. Here's the required code:

```

rowCounts = New HtmlTableRow()
celCounts = New HtmlTableCell()
celCounts.InnerText = ""
rowCounts.Cells.Add(celCounts)
celCounts = New HtmlTableCell()
celCounts.InnerText = rowDets(intDpos)("str")
rowCounts.Cells.Add(celCounts)
celCounts = New HtmlTableCell()
celCounts.InnerText = rowDets(intDpos)("cnt")
celCounts.Align = "right"
rowCounts.Cells.Add(celCounts)
tblCounts.Rows.Add(rowCounts)

```

This completes the code for the *QueryStats* subroutine, and for the Web page. A complete listing of the *QueryStats* subroutine is shown here:

```

Sub QueryStats()
    Dim dstUsage As DataSet
    Dim rdrUsage As StreamReader
    Dim tblUsage As DataTable
    Dim rowUsage As DataRow
    Dim intHpos As Integer
    Dim intDpos As Integer
    Dim rowDets As DataRow()
    Dim relDetails As DataRelation
    Dim rowCounts As HtmlTableRow
    Dim celCounts As HtmlTableCell
    Dim strRelName As String

    dstUsage = New DataSet("Usage")
    rdrUsage = New StreamReader( _
        Path.Combine(Server.MapPath(Request.ApplicationPath), _
            "_vti_pvt\" & ddlFiles.SelectedItem.Value))
    dstUsage.ReadXml(rdrUsage)
    rdrUsage.Close()
    tblUsage = dstUsage.Tables(0)
    For intHpos = 0 To tblUsage.Rows.Count - 1
        If intHpos > 0 Then
            rowCounts = New HtmlTableRow()
            celCounts = New HtmlTableCell()
            celCounts.InnerHtml = "<hr>"
            celCounts.ColSpan = 3
            rowCounts.Cells.Add(celCounts)
            tblCounts.Rows.Add(rowCounts)
        End If
        rowUsage = tblUsage.Rows(intHpos)
        rowCounts = New HtmlTableRow()
        celCounts = New HtmlTableCell()
        celCounts.InnerHtml = _

```

## Creating Your Own Statistics Pages

```

        "<b>" & rowUsage("Val") & "</b>"
    rowCounts.Cells.Add(ce1Counts)
    ce1Counts = New HtmlTableCell()
    ce1Counts.InnerHtml = "<b>Total</b>"
    ce1Counts.Align = "center"
    rowCounts.Cells.Add(ce1Counts)
    ce1Counts = New HtmlTableCell()
    ce1Counts.InnerHtml = _
        "<b>" & rowUsage("Total") & "</b>"
    ce1Counts.Align = "right"
    rowCounts.Cells.Add(ce1Counts)
    tblCounts.Rows.Add(rowCounts)
    For Each relDetails In tblUsage.ChildRelations
        strRelName = relDetails.RelationName.ToString
        rowDets = rowUsage.GetChildRows(strRelName)
        For intDpos = 0 To UBound(rowDets)
            rowCounts = New HtmlTableRow()
            ce1Counts = New HtmlTableCell()
            ce1Counts.InnerText = ""
            rowCounts.Cells.Add(ce1Counts)
            ce1Counts = New HtmlTableCell()
            ce1Counts.InnerText = rowDets(intDpos)("str")
            rowCounts.Cells.Add(ce1Counts)
            ce1Counts = New HtmlTableCell()
            ce1Counts.InnerText = rowDets(intDpos)("cnt")
            ce1Counts.Align = "right"
            rowCounts.Cells.Add(ce1Counts)
            tblCounts.Rows.Add(rowCounts)
        Next
    Next
End Sub

```

These ASP and ASP.NET Web pages might have seemed surprisingly complex. This is due in large part to the structure of the XML file, which uses child nodes rather than separate tables to store data involved in a header-detail relationship.

