6

Tracing

One of the most common ways to debug traditional ASP web applications is to use trusty calls to `Response.Write`. This enables you to create checkpoints in your code to view the contents of variables in the browser. This approach had several drawbacks, however. The output created by calls to `Response.Write` appears wherever the call is made in the code. It sometimes becomes a chore trying to interpret your debugging information because it is strewn all over the page in the browser. The formatting of your page is also affected.

When you are finished debugging your web application using calls to `Response.Write`, you are then faced with the daunting task of stripping out all this debug code. Because you are using the same type of code both to debug and to create valid output, you must carefully scan each of your ASP pages to make sure that you remove all the `Response.Write` calls that pertain to debugging. When your ASP pages get to be hundreds of lines long, this can be a real pain.

To address this issue, ASP.NET implements the `TraceContext` class. The `TraceContext` class solves all these issues and offers many more features. Let's take a look at some of the ways that the `TraceContext` class can help you debug your ASP.NET web applications more effectively.

Configuration

To use tracing in your ASP.NET web application, you need to enable it. This can be done at either the page level or the application level.

Page-Level Configuration

Enabling tracing at the page level entails adding the `Trace` attribute to the `@Page` directive, like this:

```
<%@ Page Language="C#" Trace="true" %>
```

If the `Trace` attribute has a value of `true`, tracing information will be displayed at the bottom of your ASP.NET page after the entire page has been rendered. Alternatively, you can include the `TraceMode` attribute. The value that you assign to this attribute determines the display order of the trace results. The possible values are `SortByTime` and `SortByCategory`. `SortByTime` is the default if you do not specify the `TraceMode` attribute.

Application-Level Configuration

Several tracing options are available at the application level. These settings are specified using the `<trace>` XML element in the `<system.web>` section of the web.config file. The attributes available to you are shown in Table 6.1.

Table 6.1 **Tracing Options**

`enabled`	Is `true` if tracing is enabled for the application; otherwise, is `false`. The default is `false`.
`pageOutput`	Is `true` if trace information should be displayed both on an application's pages and in the .axd trace utility; otherwise, is `false`. The default is `false`. Note that pages that have tracing enabled on them are not affected by this setting.
`requestLimit`	Specifies the number of trace requests to store on the server. The default is 10.
`traceMode`	Indicates whether trace information should be displayed in the order it was processed, `SortByTime`, or alphabetically by user-defined category, `SortByCategory`. `SortByTime` is the default.
`localOnly`	Is true if the trace viewer (trace.axd) is available only on the host Web server; otherwise, is false. The default is true.

An example of a trace entry in the web.config file might look like Listing 6.1.

Listing 6.1 **Application-Level Trace Configuration in the web.config File**

```
<configuration>
    <system.web>
        <trace enabled="true" pageOutput="false" requestLimit="20"
            traceMode="SortByTime" localOnly="true" />
    </system.web>
</configuration>
```

Even though this example uses all the available attributes, none of them is required. Also note that page-level configuration settings overrule application-level settings. For instance, if tracing is disabled at the application level but is enabled at the page level, trace information will still be displayed in the browser.

The `requestLimit` attribute sets a limit on how many page requests are kept in the trace log. This prevents the logs from getting too large.

Setting the `localOnly` attribute to `true` enables you to view trace information if you are logged into the server locally, but remote users will not see anything. That way, you can enable tracing to debug a problem, and the users of your website will never be the wiser.

Trace Output

Now that you've heard so much about configuring ASP.NET tracing, what exactly does it provide? Essentially, the trace output generated by the `TraceContext` object and displayed at the bottom of your rendered ASP.NET page contains several sections. Each of these sections is outlined here, along with explanations. Because the total output is too large to be viewed in one screenshot, a screenshot is included for each individual section of the trace output. Later in the chapter, when we discuss writing messages to the trace output, you'll get a chance to see what several sections put together look like.

Request Details

The Request Details section contains six pieces of information, outlined in Table 6.2.

Table 6.2 **Request Details**

Item	Description
Session Id	Unique identifier for your session on the server
Time of request	The time (accurate to the second) that the page request was made
Request encoding	The encoding of the request—for example, Unicode (UTF – 8)
Request type	`GET` or `POST`
Status code	The status code for the request—for example, 200
Response encoding	The encoding of the response—for example, Unicode (UTF – 8)

You can see it all put together in Figure 6.1.

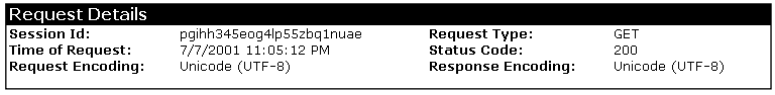


Figure 6.1 Request Details section of trace output.

Trace Information

The Trace Information section contains the various trace messages and warnings that both you and the ASP.NET engine add to the trace output. By default, the ASP.NET engine adds messages for when any events begin or end, as with `PreRender` and `SaveViewState`. The fields displayed for each item are Category, Message, time interval from the beginning of page processing, and time interval from the last trace output item. The order in which the contents of the Trace Information section appear is determined by either the `TraceMode` attribute of the `@Page` directive or the `TraceMode` property of the `TraceContext` class. Figure 6.2 shows an example of the Trace Information section.

Trace Information

Category	Message	From First(s)	From Last(s)
	Custom message		
	Custom warning	0.000083	0.000083
aspx.page	Begin PreRender	0.005187	0.005104
aspx.page	End PreRender	0.005275	0.000088
aspx.page	Begin SaveViewState	0.005498	0.000223
aspx.page	End SaveViewState	0.005588	0.000090
aspx.page	Begin Render	0.005668	0.000080
aspx.page	End Render	0.063119	0.057451

Figure 6.2 Trace Information section of trace output.

Control Tree

The Control Tree section lists all the elements on your ASP.NET page in a hierarchical fashion. This enables you to get a feeling for which controls contain other controls, helping you to decipher control scope and ownership issues. The fields displayed for each item are Control Id, Type, Render Size Bytes, and Viewstate Size Bytes. Figure 6.3 shows an example of the Control Tree section.

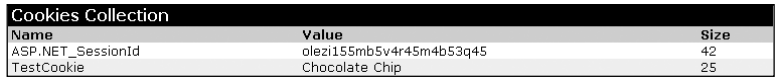
Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.test_vb_aspx	371	20
ctrl1	System.Web.UI.LiteralControl	47	0
ctrl0	System.Web.UI.HtmlControls.HtmlForm	312	0
ctrl2	System.Web.UI.LiteralControl	5	0
textbox1	System.Web.UI.WebControls.TextBox	77	0
ctrl3	System.Web.UI.LiteralControl	6	0
button1	System.Web.UI.WebControls.Button	68	0
ctrl4	System.Web.UI.LiteralControl	12	0

Figure 6.3 Control Tree section of trace output.

Cookies Collection

The Cookies Collection section lists all the cookies that are associated with your ASP.NET web application. The fields displayed for each item are Name, Value, and Size. Figure 6.4 shows an example of the Cookies Collection section.

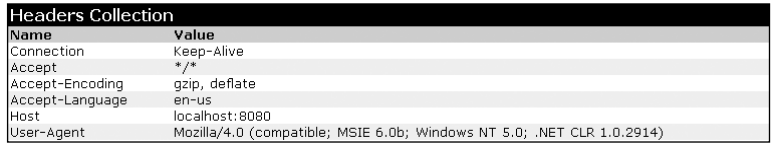


Cookies Collection		
Name	Value	Size
ASP.NET_SessionId	olezi155mb5v4r45m4b53q45	42
TestCookie	Chocolate Chip	25

Figure 6.4 Cookies Collection section of trace output.

Headers Collection

The Headers Collection section lists all the HTTP headers that are passed to your ASP.NET page. The fields displayed for each item are Name and Value. Figure 6.5 shows an example of the Headers Collection section.

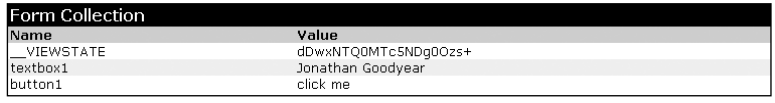


Headers Collection	
Name	Value
Connection	Keep-Alive
Accept	*/*
Accept-Encoding	gzip, deflate
Accept-Language	en-us
Host	localhost:8080
User-Agent	Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)

Figure 6.5 Headers Collection section of trace output.

Form Collection

The Form Collection section is displayed only if your ASP.NET page includes a web form and you have already submitted it back to the server. It contains two important pieces of information. First, it displays the page's `VIEWSTATE`. This is the condensed representation of the state of each of the controls on the web form. Below the `VIEWSTATE` item is a listing of each control in the Form Collection section, along with its value. The fields displayed for each item are Name and Value. Figure 6.6 shows an example of the Form Collection section.



Form Collection	
Name	Value
__VIEWSTATE	dDwxNTQ0MTc5NDg0Ozs+
textbox1	Jonathan Goodyear
button1	click me

Figure 6.6 Form Collection section of trace output.

Querystring Collection

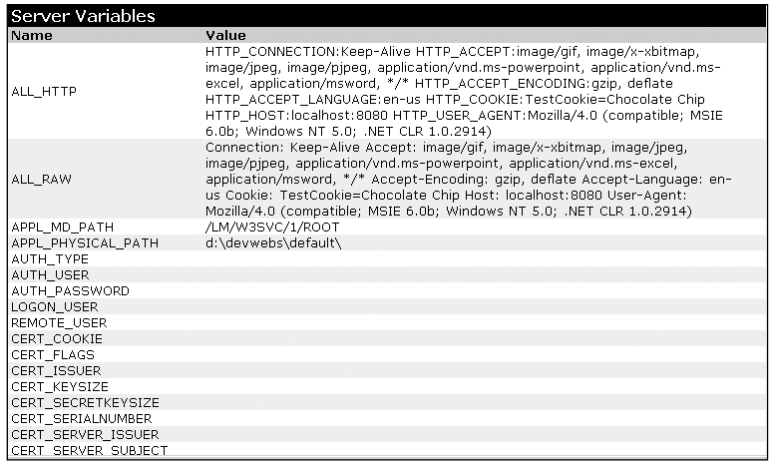
The Querystring Collection section is displayed only if your ASP.NET page has `Querystring` parameters passed to it. The fields displayed for each item are Name and Value. Figure 6.7 shows an example of the Querystring Collection section.

Querystring Collection	
Name	**Value**
trace	1

Figure 6.7 Querystring Collection section of trace output.

Server Variables

The Server Variables section contains a listing of all the server variables associated with your ASP.NET page. A few examples are `PATH_INFO`, `REMOTE_HOST`, and `SCRIPT_NAME`. The fields displayed for each item are Name and Value. Figure 6.8 shows an example of the Server Variables section (truncated because of the large number of elements in the collection).



Server Variables	
Name	**Value**
ALL_HTTP	HTTP_CONNECTION:Keep-Alive HTTP_ACCEPT:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */* HTTP_ACCEPT_ENCODING:gzip, deflate HTTP_ACCEPT_LANGUAGE:en-us HTTP_COOKIE:TestCookie=Chocolate Chip HTTP_HOST:localhost:8080 HTTP_USER_AGENT:Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
ALL_RAW	Connection: Keep-Alive Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */* Accept-Encoding: gzip, deflate Accept-Language: en-us Cookie: TestCookie=Chocolate Chip Host: localhost:8080 User-Agent: Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
APPL_MD_PATH	/LM/W3SVC/1/ROOT
APPL_PHYSICAL_PATH	d:\devwebs\default\
AUTH_TYPE	
AUTH_USER	
AUTH_PASSWORD	
LOGON_USER	
REMOTE_USER	
CERT_COOKIE	
CERT_FLAGS	
CERT_ISSUER	
CERT_KEYSIZE	
CERT_SECRETKEYSIZE	
CERT_SERIALNUMBER	
CERT_SERVER_ISSUER	
CERT_SERVER_SUBJECT	

Figure 6.8 Server Variables section of trace output.

Setting Trace Messages

The `TraceContext` class has a fairly simple interface, with only one constructor, two properties, and two methods. Of course, these are in addition to the standard properties and methods inherited from the `Object` class. An instance of the `TraceContext` class is available to your ASP.NET pages through the `Trace` property of the `Page` object, so you will need the constructor only if you want to enable tracing in your .NET components (discussed later in this chapter).

TraceContext Properties

The `IsEnabled` property works the same way as the `Trace` attribute of the `@Page` directive. The nice part about having this property available to you is that, unlike the `@Page` directive, it can be dynamically assigned. For instance, you can specify tracing through a `Querystring` parameter, as shown in Listings 6.2 and 6.3.

Listing 6.2 **Setting the *IsEnabled* Property Dynamically (C#)**

```
<%@ Page Language="C#" %>

<script language="C#" runat="server">
       protected void Page_Load(object Sender, EventArgs e)
       {
              bool traceFlag = Request.QueryString["trace"] != null
                     ? true : false;
              Trace.IsEnabled = traceFlag;
      }
</script>
```

Listing 6.3 **Setting the *IsEnabled* Property Dynamically (Visual Basic .NET)**

```
<%@ Page Language="Visual Basic" %>

<script language="Visual Basic" runat="server">
       Protected Sub Page_Load(Sender As Object, e As EventArgs)
              Dim traceFlag As Boolean = IIF(Request.QueryString("trace") _
                     <> Nothing, True, False)
              Trace.IsEnabled = traceFlag
       End Sub
</script>
```

These listings set the `IsEnabled` property of the `Trace` object dynamically, based on the presence of the `trace Querystring` variable. Notice that no `Trace` attribute is assigned to the `@Page` directive. It is interesting to note that even if you specify a `Trace` attribute and set it to `false`, the `IsEnabled` property value still dictates whether trace information was displayed to the page.

The real power of using the `IsEnabled` property is that when you set it to `false`, the trace information not only isn't displayed, but it also isn't even compiled. This means that you can leave your tracing code in your ASP.NET application when you move it to production. As long as the `IsEnabled` property is set to `false`, you will not suffer any performance penalty.

The `TraceMode` property works exactly like the `TraceMode` attribute of the `@Page` directive. The same behaviors and advantages that apply to the `IsEnabled` property also exist for the `TraceMode` property.

TraceContext Methods

Only one thing (besides their names) differentiates the two methods of the `TraceContext` class, `Write` and `Warn`: The output generated by the `Write` method is black, while the output generated by the `Warn` method is red. For this reason, we will be discussing only the `Write` method. Just realize that everything said about the `Write` method can also be applied to the `Warn` method. There are three overloaded versions of the `Write` and `Warn` methods. The first version accepts a trace message. The second version accepts a trace message and a category. The third version accepts a trace message, a category, and an instance of an `Exception` class. Each of these is covered in more detail next.

TraceContext.Write (string)

The first of the overloaded `Write` methods of the `TraceContext` class accepts a single-string parameter. This string contains the message that is displayed in the Message field of the Trace Information section of the trace output (as seen in Figure 6.2). Listings 6.4 and 6.5 demonstrate its use.

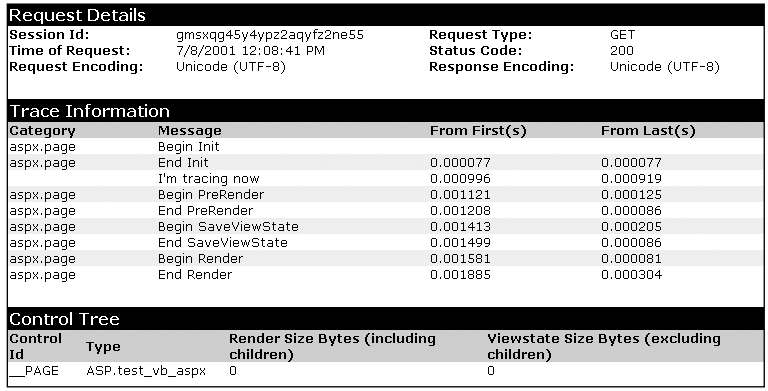
Listing 6.4 **Implementing *TraceContext.Write (string)* (C#)**

```
<%@ Page Language="C#" Trace="true" %>

<script language="C#" runat="server">
      protected void Page_Load(object Sender, EventArgs e)
      {
            Trace.Write("I'm tracing now");
      }
</script>
```

Listing 6.5 **Implementing *TraceContext.Write (string)* (Visual Basic .NET)**

```
<%@ Page Language="Visual Basic" Trace="true" %>

<script language="Visual Basic" runat="server">
      Protected Sub Page_Load(Sender As Object, e As EventArgs)
            Trace.Write("I'm tracing now")
      End Sub
</script>
```

Figure 6.9 shows what the trace output for the previous code looks like.



Request Details

Session Id:	gmsxqg45y4ypz2aqyfz2ne55	Request Type:	GET
Time of Request:	7/8/2001 12:08:41 PM	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000077	0.000077
	I'm tracing now	0.000996	0.000919
aspx.page	Begin PreRender	0.001121	0.000125
aspx.page	End PreRender	0.001208	0.000086
aspx.page	Begin SaveViewState	0.001413	0.000205
aspx.page	End SaveViewState	0.001499	0.000086
aspx.page	Begin Render	0.001581	0.000081
aspx.page	End Render	0.001885	0.000304

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.test_vb_aspx	0	0

Figure 6.9 Viewing a trace message in the trace output.

Notice the message "I'm tracing now" that appears as the third line item in the Trace Information section. No category was specified, so it is blank. The next overloaded version of the `Write/Warn` method includes the category parameter.

TraceContext.Write (string, string)

The second overloaded `Write` method of the `TraceContext` class takes two string parameters. The first parameter is the category of the trace item. It appears in the Category field of the Trace Information section of the trace output. The second parameter is the message that will be displayed in the Message field, and it is the same as the single-string parameter in the first overloaded `Write` method.

This is probably the most likely version of the `Write` method that you will use when debugging your ASP.NET pages. You can assign categories to your trace items, leveraging the `TraceMode` attribute of the `@Page` directive or the `TraceMode` property of the `TraceContext` class to sort the Trace Information section results. As previously described, this is done using the `SortByCategory` member of the `TraceMode` enumeration.

Listings 6.6 and 6.7 demonstrate the use of this version of the `Write` method.

Listing 6.6 **Implementing *TraceContext.Write (string, string)* (C#)**

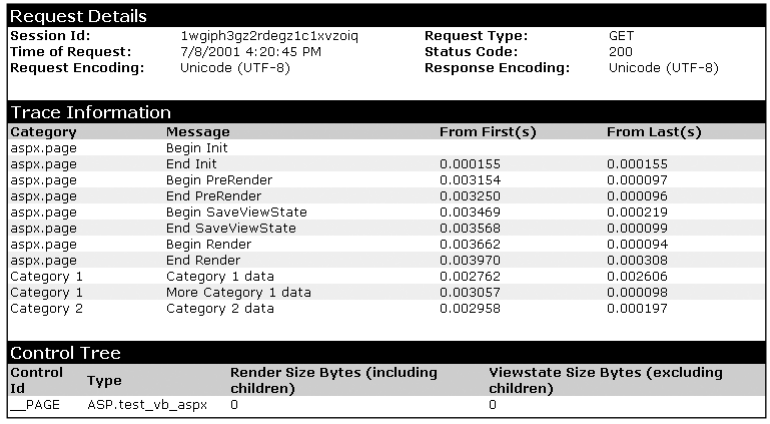
```
<%@ Page Language="C#" Trace="true" %>

<script language="C#" runat="server">
      protected void Page_Load(object Sender, EventArgs e)
      {
            Trace.TraceMode = TraceMode.SortByCategory;
```

```
            Trace.Write("Category 1", "Category 1 data");
            Trace.Write("Category 2", "Category 2 data");
            Trace.Write("Category 1", "More Category 1 data");
      }
</script>
```

Listing 6.7 **Implementing *TraceContext.Write (string, string)* (Visual Basic .NET)**

```
<%@ Page Language="Visual Basic" Trace="true" %>

<script language="Visual Basic" runat="server">
      Protected Sub Page_Load(Sender As Object, e As EventArgs)
            Trace.TraceMode = TraceMode.SortByCategory
            Trace.Write("Category 1", "Category 1 data")
            Trace.Write("Category 2", "Category 2 data")
            Trace.Write("Category 1", "More Category 1 data")
      End Sub
</script>
```

Figure 6.10 shows the trace output for the previous code.



Request Details

Session Id:	1wgiph3gz2rdegz1c1xvzoiq	**Request Type:**	GET
Time of Request:	7/8/2001 4:20:45 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000155	0.000155
aspx.page	Begin PreRender	0.003154	0.000097
aspx.page	End PreRender	0.003250	0.000096
aspx.page	Begin SaveViewState	0.003469	0.000219
aspx.page	End SaveViewState	0.003568	0.000099
aspx.page	Begin Render	0.003662	0.000094
aspx.page	End Render	0.003970	0.000308
Category 1	Category 1 data	0.002762	0.002606
Category 1	More Category 1 data	0.003057	0.000098
Category 2	Category 2 data	0.002958	0.000197

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.test_vb_aspx	0	0

Figure 6.10 Viewing a trace message with a category in the trace output.

Notice that the trace items are sorted by category so that both of the category 1 items appear together, instead of being separated by the category 2 item (which was the order in which the code made the calls to the `Write` method). Also, the previous code uses the `TraceMode` property of the `Trace` object to set the sort order. You could alternatively have used the `TraceMode` attribute of the `@Page` directive.

TraceContext.Write (string, string, Exception)

The third overloaded version of the `Write` method takes three parameters. The first two parameters match up with the two parameters of the previous overloaded method call. For the third parameter, you should pass in an object instance of the `Exception` class or an object instance of a class that inherits from the `Exception` class. You would most likely use this method call when writing trace output in conjunction with structured exception handling. Listings 6.8 and 6.9 demonstrate this concept by intentionally causing an exception in a `Try` block that adds to the trace information in the `Catch` block.

Listing 6.8 **Implementing *TraceContext.Write (string, string, Exception)* (C#)**

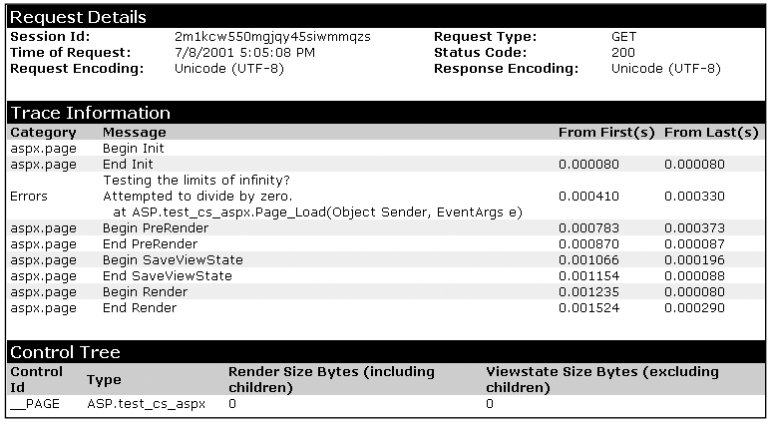
```
<%@ Page Language="C#" Trace="true" %>

<script language="C#" runat="server">
      protected void Page_Load(object Sender, EventArgs e)
      {
            int x = 1;
            int y = 0;

            try
            {
                  int z = x / y;
            }
            catch(DivideByZeroException ex)
            {
                 Trace.Write("Errors","Testing the limits of infinity?",ex);
            }
      }
</script>
```

Listing 6.9 **Implementing *TraceContext.Write (string, string, Exception)* (Visual Basic .NET)**

```
<%@ Page Language="Visual Basic" Trace="true" %>

<script language="Visual Basic" runat="server">
      Protected Sub Page_Load(Sender As Object, e As EventArgs)
            Dim x As Integer = 1
            Dim y As Integer = 0

            Try
```

```
            Dim z As Integer = x / y
        Catch ex As OverflowException
            Trace.Write("Errors","Testing the limits of
            ➥infinity?",ex)
        End Try
    End Sub
</script>
```

Figure 6.11 shows the trace output for the C# version of this code.



Request Details

Session Id:	2m1kcw550mgjqy45siwmmqzs	**Request Type:**	GET
Time of Request:	7/8/2001 5:05:08 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000080	0.000080
Errors	Testing the limits of infinity? Attempted to divide by zero. at ASP.test_cs_aspx.Page_Load(Object Sender, EventArgs e)	0.000410	0.000330
aspx.page	Begin PreRender	0.000783	0.000373
aspx.page	End PreRender	0.000870	0.000087
aspx.page	Begin SaveViewState	0.001066	0.000196
aspx.page	End SaveViewState	0.001154	0.000088
aspx.page	Begin Render	0.001235	0.000080
aspx.page	End Render	0.001524	0.000290

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.test_cs_aspx	0	0

Figure 6.11 Viewing a trace message with a category and exception information in the trace output.

In addition to the message that you specify in the call to the `Write` method, you get the message from the exception that was thrown, as well as the name of the procedure where the exception occurred. You can see valuable debugging information associated with the error that was thrown alongside your own custom comments, making it easier to combine the two into a solution to the bug.

Trace Viewer

In the "Application-Level Configuration" section at the beginning of the chapter, we discussed the various attributes of the `<trace>` XML element in the web.config file. You'll recall that the `requestLimit` attribute sets how many page requests to keep in the trace log. So, now that you have all that data stored in the trace log, what do you do with it? Another fine question! The answer is to use the Trace Viewer to analyze it.

Accessing the Trace Viewer

The Trace Viewer is accessed via a special URL. In any directory of your web application, you can access it by navigating to trace.axd. You'll notice that there is no trace.axd file anywhere. Instead, any request for this file is intercepted by an `HttpHandler` that is set up in either the machine.config file or your web application's web.config file. An entry within the `<httpHandlers>` XML element looks like Listing 6.10.

Listing 6.10 ***HttpHandlers* Section of the machine.config File**

```
<httpHandlers>
      ...other handler entries...
       <add verb="*" path="trace.axd"
             type="System.Web.Handlers.TraceHandler,
             System.Web, Version=1.0.2411.0, Culture=neutral,
             PublicKeyToken=b03f5f7f11d50a3a" />
       ...other handler entries...
</httpHandlers>
```

With this `HttpHandler` entry in place, all that is left to do to use the Trace Viewer is make sure that the `enabled` attribute of the `<trace>` XML element in your web.config file is set to `true`.

Using the Trace Viewer

The Trace Viewer uses a fairly simple interface, consisting of two different pages. When you first navigate to the trace.axd file, you are presented with the Application Trace screen. It contains a list of page requests for which trace information has been tracked.

Three items are present in the header of this page. The first is a link to clear the current trace log. Clicking this link resets tracing, clearing all page requests from the screen. The second item in the header is the physical directory of the ASP.NET web application. The third header item is a counter that tells you how many more requests can be tracked before the `requestLimit` is reached. After that point, trace information is not stored for anymore page requests until the trace information is cleared by clicking the Clear Current Trace link.

The fields displayed for each page request on the Application Trace screen are No., Time of Request, File, Status Code, and Verb. In addition, a link next to each item in the list shows the details for that specific page request. Figure 6.12 shows an example of the Application Trace screen of the Trace Viewer.

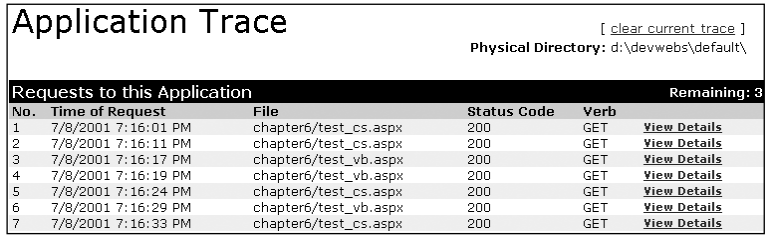


Figure 6.12 Application Trace page of the Trace Viewer.

When you click one of the View Details links on the Application Trace screen, you are taken to the Request Details screen. On this page you will see is an exact representation of the trace information that would be displayed at the end of the particular ASP.NET page if tracing had been enabled on it. The only difference is the large Request Details caption at the top of the page. Several examples of this screen have been shown in previous figures in this chapter, so there is no need to present it again.

Tracing via Components

The `Page` object in your ASP.NET pages contains an instance of the `TraceContext` class, making it easy to write trace information from your ASP.NET page. But what if you want to write trace information from within a component? Luckily, the .NET Framework makes this task equally easy. Let's take a look at how this would be done. First, you need to build your simple component. Listings 6.11 and 6.12 present the component.

Listing 6.11 **Component That Leverages ASP.NET Tracing (C#)**

```
using System;
using System.Web;

namespace Chapter6
{
      public class TestClass
      {
            public void DoStuff()
            {
                  HttpContext.Current.Trace.Write
                        ("Component", "I'm inside the component");
            }
      }
}
```

Listing 6.12 **Component That leverages ASP.NET Tracing (Visual Basic .NET)**

```
Imports System
Imports System.Web

Namespace Chapter6
       Public Class TestClass
              Public Sub DoStuff()
                     HttpContext.Current.Trace.Write _
                           ("Component", "I'm inside the component")
              End Sub
       End Class
End Namespace
```

Next, compile your component using one of the following compile scripts. The first is for C#, and the second is for Visual Basic .NET.

```
csc /t:library /out:Chapter6.dll /r:System.Web.dll Chapter6.cs
```

or

```
vbc /t:library /out:Chapter6.dll /r:System.Web.dll Chapter6.vb
```

Finally, you can see that this works by using it in an ASP.NET page.

Listing 6.13 **Using Trace-Enabled Component in an ASP.NET Page (C#)**

```
<%@ Page Language="C#" Trace="true" %>
<%@ Import Namespace="Chapter6" %>

<script language="C#" runat="server">
      protected void Page_Load(object Sender, EventArgs e)
      {
              TestClass tc = new TestClass();
              tc.DoStuff();
       }
</script>
```

Listing 6.14 **Using Trace-Enabled Component in an ASP.NET Page (Visual Basic .NET)**

```
<%@ Page Language="Visual Basic" Trace="true" %>
<%@ Import Namespace="Chapter6" %>

<script language="Visual Basic" runat="server">
```

```
        Protected Sub Page_Load(Sender As Object, e As EventArgs)
              Dim tc As TestClass = New TestClass()
              tc.DoStuff()
        End Sub
</script>
```

When you run this code, you'll get results like those shown in Figure 6.13.

Request Details

Session Id:	mrkxgbbzh31d4fvak4spvt55	**Request Type:**	GET
Time of Request:	7/8/2001 9:51:53 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000180	0.000180
Component	I'm inside the component	0.000738	0.000557
aspx.page	Begin PreRender	0.000851	0.000113
aspx.page	End PreRender	0.000935	0.000085
aspx.page	Begin SaveViewState	0.001167	0.000232
aspx.page	End SaveViewState	0.001255	0.000088
aspx.page	Begin Render	0.001336	0.000080
aspx.page	End Render	0.001600	0.000264

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.test_vb_aspx	0	0

Figure 6.13 Viewing trace information written to the trace output from within a component.

Inside the Trace Information section, you'll see the trace message "I'm inside the component," with a category of Component that was added from within the component. This can be a powerful tool for finding bugs in your ASP.NET web applications.

Tips for Using Trace Information

Now that you have all this trace information sitting in front of you, how do you use it to your best advantage? Well, that really depends. Most of the trace information presented (such as cookies, headers, and server variables) was available to you in traditional ASP. It just wasn't neatly packaged like it is in the Trace Viewer. You can use that information just as you previously did.

The true power of ASP.NET tracing is in the Trace Information section. It enables you to see when each part of your ASP.NET page is processing and determine how long it takes to process. This can be crucial to the process of finding performance bottlenecks in your code. It can also help you solve mysteries about why certain code is not processing correctly. Often, the code isn't being executed in the same order that you thought it was. Or, maybe the code is being executed multiple times by accident. These nuances, which were tough to discover in traditional ASP, become fairly obvious when observing the contents of the Trace Information section of the trace output.

Application-level tracing, if used properly, can greatly reduce the amount of time and effort expended on debugging your ASP.NET web applications. For instance, you could turn on tracing but set the `pageOutput` attribute of the `<trace>` XML element in the web.config file to `false`. Then you could let some of the potential users of your web application try it out. You can record lots of information about what they are doing and what is going wrong with their experience, all behind the scenes. This can help you to determine which particular scenarios cause errors.

Summary

In this chapter, you took a detailed look at tracing in ASP.NET. You started by learning how to configure tracing in ASP.NET, at both the page level and the application level. This entailed adding attributes to the `@Page` directive and to the web.config file.

Next, you learned about the different sections that are included in the trace output at the page level. These include the Request Details, Trace Information, Control Tree, Cookies Collection, Headers Collection, Form Collection, QueryString Collection, and Server Variables Collection sections.

Following that, we discussed the primary player in the ASP.NET tracing process: the `TraceContext` class. You learned about its two properties, `IsEnabled` and `TraceMode`, and you learned how they can be used to control the trace output of your ASP.NET pages through the `TraceContext` object instance in the Page class's `Trace` property.

The `TraceContext` class's two methods, `Write` and `Warn`, were discussed next. Each of the three overloaded `Write` methods was explained and was correlated to the similar `Warn` method, which differs only in name and output appearance. We deferred the discussion of the constructor to the section on tracing via components, later in the chapter.

Then you learned how to both configure and use the Trace Viewer, as well as how it is accessed via an `HttpHandler` that intercepts requests for the trace.axd file.

Tracing in an ASP.NET web application is not just limited to ASP.NET pages. We also discussed how to leverage ASP.NET tracing from within components that your ASP.NET pages call. The chapter wrapped up with a few tips and techniques for utilizing ASP.NET tracing to its fullest potential.

In the next chapter, you'll get a thorough introduction to debugger in the Visual Studio .NET IDE.