7

Visual Studio .NET Debugging Environment

THE ASP SCRIPT DEBUGGER THAT IS INTEGRATED with the new Visual Studio .NET IDE is, without a doubt, one of the greatest enhancements to debugging from the previous versions of ASP. Unlike trying to debug traditional ASP pages in Visual Interdev, this actually works right out of the box!

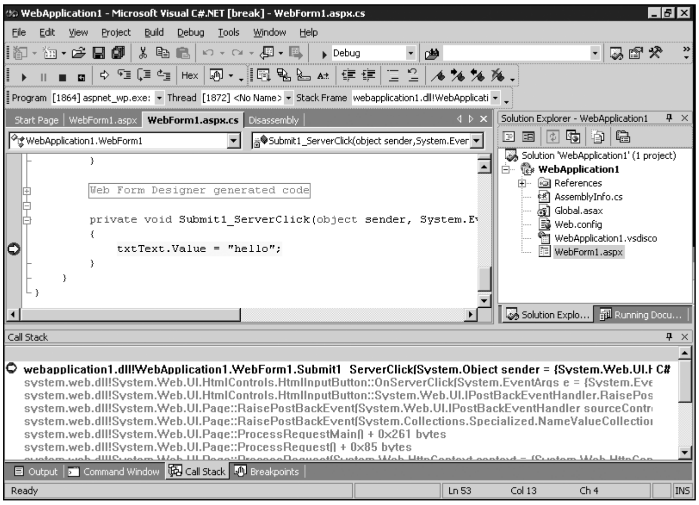
In this chapter, you will be looking at all the features available for debugging in the Visual Studio .NET IDE, how to use them, and where each one is applicable for the problem you might be trying to conquer. This will all be accomplished by building a project from scratch in the IDE, so we recommend creating the project on your own as it is done in this chapter.

Introduction to Features

Let's start out by taking a look at the most important features of the Visual Studio .NET IDE debugger. You will take a detailed look at all of these as the chapter progresses. Many of these features have existed in the Visual Studio 6.0 IDEs; however, not all were previously available when debugging traditional ASP pages in Visual InterDev.

Call Stack

The call stack enables you to display the list of functions currently called. As the name implies, this can be imagined simply as a stack. (As functions are called from within function, which, in turn, are called from within functions, a stack is created.) With the call stack viewer, you can look at this stack and jump forward and backward into the stack to debug at any point in the chain. Figure 7.1 shows the call stack in action.


WebApplication1 - Microsoft Visual C#.NET [break] - WebForm1.aspx.cs
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Program [1864] aspnet_wp.exe: Thread [1872] <No Name> Stack Frame webapplication1.dll!WebApplicati
Start Page WebForm1.aspx WebForm1.aspx.cs Disassembly
WebApplication1.WebForm1
Submit1_ServerClick(object sender,System.Ever
Solution Explorer - WebApplication1
Solution 'WebApplication1' (1 project)
WebApplication1
References
AssemblyInfo.cs
Global.asax
Web.config
WebApplication1.vsdisco
WebForm1.aspx
Solution Explo...
Running Docu...
Call Stack
Output
Command Window
Call Stack
Breakpoints
Ready
Ln 53
Col 13
Ch 4
INS

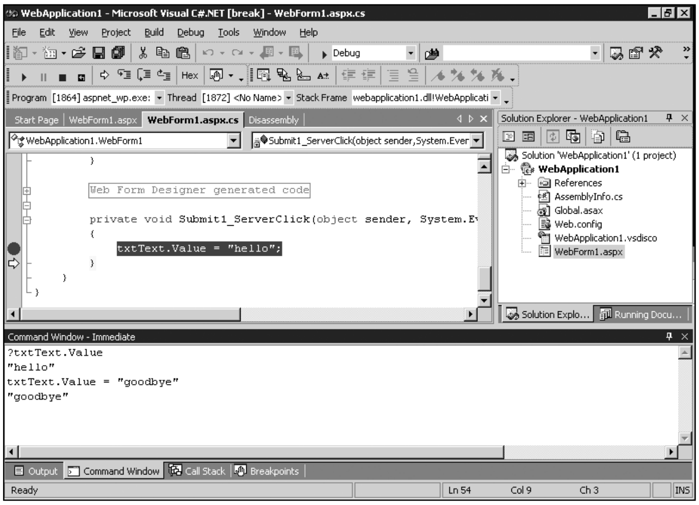
```
        }

        Web Form Designer generated code

        private void Submit1_ServerClick(object sender, System.E
        {
            txtText.Value = "hello";
        }
    }
}
```

```
webapplication1.dll!WebApplication1.WebForm1.Submit1_ServerClick(System.Object sender = {System.Web.UI.H C#
system.web.dll!System.Web.UI.HtmlControls.HtmlInputButton::OnServerClick(System.EventArgs e = {System.Eve
system.web.dll!System.Web.UI.HtmlControls.HtmlInputButton::System.Web.UI.IPostBackEventHandler.RaisePos
system.web.dll!System.Web.UI.Page::RaisePostBackEvent(System.Web.UI.IPostBackEventHandler sourceContr
system.web.dll!System.Web.UI.Page::RaisePostBackEvent(System.Collections.Specialized.NameValueCollectio
system.web.dll!System.Web.UI.Page::ProcessRequestMain() + 0x261 bytes
system.web.dll!System.Web.UI.Page::ProcessRequest() + 0x85 bytes
```

Figure 7.1 The call stack window.

Command Window

If you have used the Visual Basic IDE previously, the command window will be quite familiar to you. The command window enables you to execute program statements separately from the running program. For example, if you want to set a variable equal to a new value, or if you want to print the value of a variable, or even if you want to create an object and call some methods on it, you can do it from this window. Figure 7.2 shows how the command window can be used.


WebApplication1 - Microsoft Visual C#.NET [break] - WebForm1.aspx.cs
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Program [1864] aspnet_wp.exe: Thread [1872] <No Name> Stack Frame webapplication1.dll!WebApplicati
Start Page WebForm1.aspx WebForm1.aspx.cs Disassembly
WebApplication1.WebForm1
Submit1_ServerClick(object sender,System.Ever
Solution Explorer - WebApplication1
Solution 'WebApplication1' (1 project)
WebApplication1
References
AssemblyInfo.cs
Global.asax
Web.config
WebApplication1.vsdisco
WebForm1.aspx
Solution Explo... Running Docu...
Command Window - Immediate
Output Command Window Call Stack Breakpoints
Ready Ln 54 Col 9 Ch 3 INS

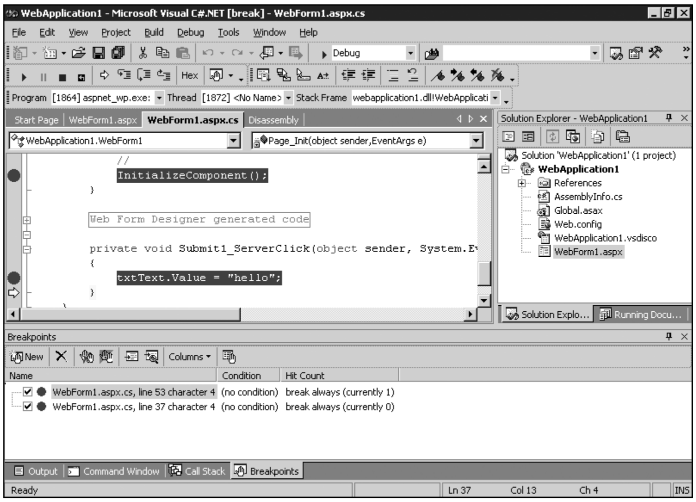
```
        }

        Web Form Designer generated code

        private void Submit1_ServerClick(object sender, System.E
        {
            txtText.Value = "hello";
        }
    }
}
```

```
?txtText.Value
"hello"
txtText.Value = "goodbye"
"goodbye"
```

Figure 7.2 The command window.

Breakpoints

Breakpoints enable you to stop the execution of a program at a defined point either in all instances or based on a certain set of criteria. The easiest way to set a breakpoint is to click in the gray left margin next to the line where you want to stop. The IDE drops a red "dot" in the margin; when you start the program in debug mode, it stops at that specified position. By popping up the breakpoint window from the debugging windows at the bottom of the screen, you can see all breakpoints currently set in the system. Figure 7.3 shows an example of the breakpoint window.


WebApplication1 - Microsoft Visual C#.NET [break] - WebForm1.aspx.cs
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Program [1864] aspnet_wp.exe: Thread [1872] <No Name> Stack Frame webapplication1.dll!WebApplicati
Start Page WebForm1.aspx WebForm1.aspx.cs Disassembly
WebApplication1.WebForm1
Page_Init(object sender,EventArgs e)
Solution Explorer - WebApplication1
Solution 'WebApplication1' (1 project)
WebApplication1
References
AssemblyInfo.cs
Global.asax
Web.config
WebApplication1.vsdisco
WebForm1.aspx
Solution Explo... Running Docu...
Breakpoints
New Columns
Output Command Window Call Stack Breakpoints
Ready Ln 37 Col 13 Ch 4 INS

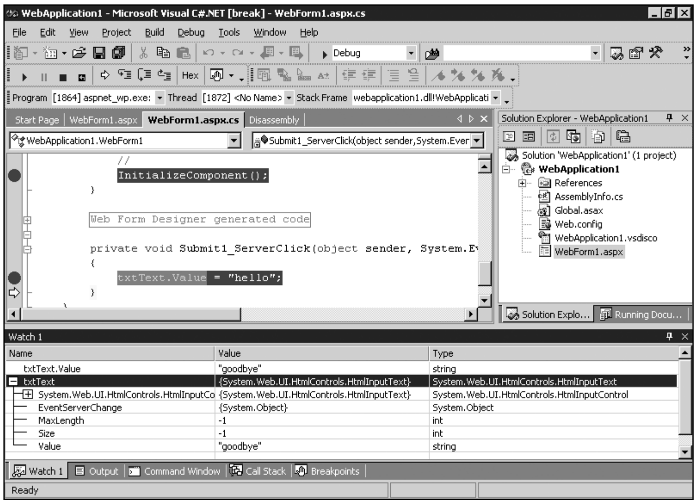
```
            //
            InitializeComponent();
        }

        Web Form Designer generated code

        private void Submit1_ServerClick(object sender, System.E
        {
            txtText.Value = "hello";
        }
```

Name	Condition	Hit Count
WebForm1.aspx.cs, line 53 character 4	(no condition)	break always (currently 1)
WebForm1.aspx.cs, line 37 character 4	(no condition)	break always (currently 0)

Figure 7.3 The breakpoint window.

Watch Window

The watch window can be used to watch the contents of a variable or series of variables. This window also enables you to change the value of a variable, which can be quite useful in debugging applications. You can see an example of the watch window in Figure 7.4.


WebApplication1 - Microsoft Visual C#.NET [break] - WebForm1.aspx.cs
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Program [1864] aspnet_wp.exe: Thread [1872] <No Name> Stack Frame webapplication1.dll!WebApplicati
Start Page WebForm1.aspx WebForm1.aspx.cs Disassembly
WebApplication1.WebForm1
Submit1_ServerClick(object sender,System.Ever
Solution Explorer - WebApplication1
Solution 'WebApplication1' (1 project)
WebApplication1
References
AssemblyInfo.cs
Global.asax
Web.config
WebApplication1.vsdisco
WebForm1.aspx
Solution Explo... Running Docu...
Watch 1
Watch 1 Output Command Window Call Stack Breakpoints
Ready

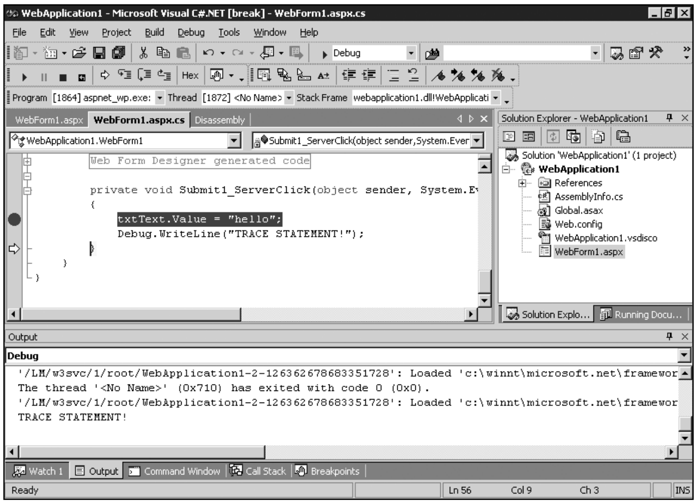
```
            //
            InitializeComponent();
        }

        Web Form Designer generated code

        private void Submit1_ServerClick(object sender, System.E
        {
            txtText.Value = "hello";
        }
```

Name	Value	Type
txtText.Value	"goodbye"	string
txtText	{System.Web.UI.HtmlControls.HtmlInputText}	System.Web.UI.HtmlControls.HtmlInputText
System.Web.UI.HtmlControls.HtmlInputCo	{System.Web.UI.HtmlControls.HtmlInputText}	System.Web.UI.HtmlControls.HtmlInputControl
EventServerChange	{System.Object}	System.Object
MaxLength	-1	int
Size	-1	int
Value	"goodbye"	string

Figure 7.4 The watch window.

Tracing

Tracing simply enables you to write out a series of text messages as the program is executing so that you can see what is happening at any point during the program's life cycle. This can be very useful in creating a log of events that can be inspected later to ensure that the program is operating as you expect in all aspects. Figure 7.5 shows an output from a `Debug.Trace` statement in the output window.


WebApplication1 - Microsoft Visual C#.NET [break] - WebForm1.aspx.cs
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Program [1864] aspnet_wp.exe: Thread [1872] <No Name> Stack Frame webapplication1.dll!WebApplicati
WebForm1.aspx WebForm1.aspx.cs Disassembly
WebApplication1.WebForm1
Submit1_ServerClick(object sender,System.Ever
Solution Explorer - WebApplication1
Solution 'WebApplication1' (1 project)
WebApplication1
References
AssemblyInfo.cs
Global.asax
Web.config
WebApplication1.vsdisco
WebForm1.aspx
Solution Explo... Running Docu...
Output
Debug
Watch 1 Output Command Window Call Stack Breakpoints
Ready Ln 56 Col 9 Ch 3 INS

```
        Web Form Designer generated code

        private void Submit1_ServerClick(object sender, System.E
        {
            txtText.Value = "hello";
            Debug.WriteLine("TRACE STATEMENT!");
        }
    }
}
```

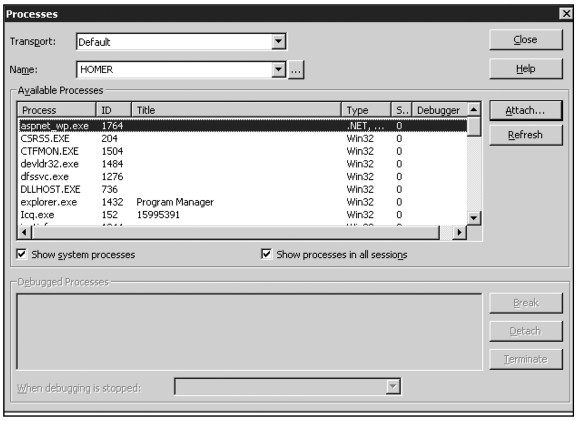
```
'/LM/w3svc/1/root/WebApplication1-2-126362678683351728': Loaded 'c:\winnt\microsoft.net\framewor
The thread '<No Name>' (0x710) has exited with code 0 (0x0).
'/LM/w3svc/1/root/WebApplication1-2-126362678683351728': Loaded 'c:\winnt\microsoft.net\framewor
TRACE STATEMENT!
```

Figure 7.5 Tracing and the output window.

Attaching to Processes

At some point you might need to attach to a process running somewhere on the computer and debug it from within the ASP.NET page. This could be the ASP.NET process, aspnet_wp.exe, or another running service, or any other program running on the server. This can be accomplished quite easily.

Under the Debug menu you will find the Processes selection. Clicking this brings up the dialog box in Figure 7.6.



Transport: Default

Name: HOMER

Available Processes

Process	ID	Title	Type	S..	Debugger
aspnet_wp.exe	1764		.NET, ...	0	
CSRSS.EXE	204		Win32	0	
CTFMON.EXE	1504		Win32	0	
devldr32.exe	1484		Win32	0	
dfssvc.exe	1276		Win32	0	
DLLHOST.EXE	736		Win32	0	
explorer.exe	1432	Program Manager	Win32	0	
Icq.exe	152	15995391	Win32	0	

[✓] Show system processes [✓] Show processes in all sessions

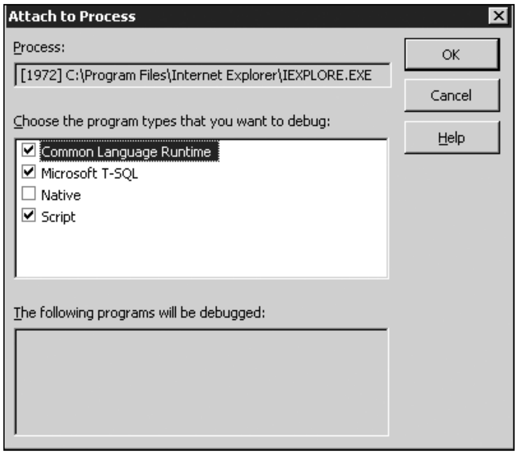
Debugged Processes

When debugging is stopped:

Figure 7.6 Attaching to a running process.

By default, you will see only the processes that you have started in some way. If you click the Show System Processes check box, you have access to everything that is running on the current machine. Click the process to be debugged, and then click the Attach button. Now, if that process hits a breakpoint or any other type of debugger event, that event pops up in the Visual Studio .NET IDE. You can also force the program to break by clicking the Break button at the bottom of the window. At any time, you can stop debugging the process by clicking the Detach button. Terminate kills the process altogether.

After clicking the Attach button, you are given a choice of what type of debugging you want to do on the process you've selected. Figure 7.7 shows an example of this dialog box.



Process:
[1972] C:\Program Files\Internet Explorer\IEXPLORE.EXE

Choose the program types that you want to debug:

[✓] Common Language Runtime
[✓] Microsoft T-SQL
[ ] Native
[✓] Script

The following programs will be debugged:

Figure 7.7 Choosing the debug type.

If you are attaching to a .NET process written using the Common Language Runtime (CLR), choose the Common Language Runtime option from the dialog box. If you'll be breaking into a process that uses Microsoft Transact SQL language, check Microsoft T-SQL as your debug type. Native enables you to debug a standard Win32 application. This enables you to debug a Win32 program at an assembly language level. Finally, the Script type gives you the capability to debug standard Visual BasicScript and JavaScript. This is especially useful for debugging an instance of Internet Explorer.

Setting It All Up

There really isn't a whole lot to mention here. It is extremely simple to use the Visual Studio .NET IDE for debugging ASP.NET pages. In most cases, it will be a plug-and-play affair. The default debug build of any ASP.NET project will have everything set up for you to begin. However, even though that probably will be the case, we will discuss what is absolutely required for the debugging to work, just in case something goes wrong.

Take a look at the web.config file contained in your project. This is an XML file that contains specific configuration information for your ASP.NET project. One line in this file will look similar to the following:

```
<compilation defaultLanguage="vb" debug="true" />
```

The `defaultLanguage` parameter will be based on the default language of your ASP.NET project. But what we are concerned about here is the `debug` parameter. If you are running in a debugging environment and want to be able to access the spiffy features of the Visual Studio .NET IDE, this `debug` parameter must be set to `true`, as it is in the previous line. If it is set to `false`, none of the features will work. This is what you want for a release build of your project.

Inline Debugging of ASP.NET Pages

This is so very easy—you're going to be extremely happy about this. If you've ever debugged a program using Visual Studio 6.0 (Visual Basic, Visual C++, and so on) you will feel right at home with what you are about to learn about Visual Studio .NET. Let's discuss these great features using a sample project mentioned earlier in the chapter. As usual, both Visual Basic .NET and C# versions of the code will be provided for you to see.

This sample project will consist of an ASP.NET page and a Visual Basic .NET/C# component so that you can see how easily the two interact and how they can be debugged simultaneously. The project itself will simply ask the user for a valid email address and then send a form letter to that address. Start out by creating the project. We called ours Chap5Visual Basic for the Visual Basic .NET version and Chap5CS for the C# version.

The first thing to do is create the ASP.NET page that the user will see. Listing 7.1 contains the main ASP.NET page that contains the input form on which the user can enter the email address where the mail will be sent. Here the page name is left as `WebForm1`, the default name provided when the project was created.

Listing 7.1 **ASP.NET Page for Debugging Example**

```
<%@ Page Language="vb" AutoEventWireup="false"
➥Codebehind="WebForm1.aspx.vb" Inherits="Chap7VB.WebForm1"%>
<html>
    <HEAD>
        <title>Email Test Page</title>
    </HEAD>
    <body MS_POSITIONING="GridLayout">
        <form id="Form1" method="post" runat="server">
            Please enter the email address to send to:
            <br>
            <input type="text" id="txtEmail" runat="server"
            ➥NAME="txtEmail">
            <br>
            <input type="submit" id="btnSubmit" value="Send Email"
            ➥runat="server" NAME="btnSubmit">
        </form>
    </body>
</html>
```

This page would work in a Visual Basic .NET project. To have it work in a C# project, change the first line to the following:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs" AutoEventWireup="false"
➥Inherits="Chap7CS.WebForm1" %>
```

This is a very simple page. It consists of two elements: a text box for the email address and a Submit button to send the form to the server. Now you will create the server-side code for this project. It will be contained in two parts: First, you will look at the code-behind file that is associated with this file. Here you will verify that you have a valid email address. Second, you will create a Visual Basic .NET/C# component that will actually send the email. Listing 7.2 contains the code-behind file for the C# project, and Listing 7.3 contains the code-behind file for the Visual Basic .NET project.

Listing 7.2 **Listing for Debugging Example (C#)**

```
using System;

namespace Chap7CS
{
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.HtmlControls.HtmlInputText txtEmail;
        protected System.Web.UI.HtmlControls.HtmlInputButton btnSubmit;

        public WebForm1()
        {
            Page.Init += new System.EventHandler(Page_Init);
        }

        private void Page_Init(object sender, EventArgs e)
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
                this.btnSubmit.ServerClick += new
                ➥System.EventHandler(this.btnSubmit_ServerClick);
        }

        private void btnSubmit_ServerClick(object sender,
        ➥System.EventArgs e)
        {
            if(txtEmail.Value.IndexOf("@") == -1 ¦¦
                txtEmail.Value.IndexOf(".") == -1)
                Response.Write("The supplied email address is not
                ➥valid.");
        }
    }
}
```

Listing 7.3 **Listing for Debugging Example (Visual Basic .NET)**

```
Public Class WebForm1
        Inherits System.Web.UI.Page
        Protected WithEvents txtEmail As
        ➥System.Web.UI.HtmlControls.HtmlInputText
        Protected WithEvents btnSubmit As
        ➥System.Web.UI.HtmlControls.HtmlInputButton

Private Sub btnSubmit_ServerClick(ByVal sender As System.Object, ByVal e
➥As System.EventArgs) Handles btnSubmit.ServerClick
        If txtEmail.Value.IndexOf("@") = -1 Or _
            txtEmail.Value.IndexOf(".") = -1 Then
            Response.Write("The supplied email address is not valid.")
        End If
    End Sub
End Class
```

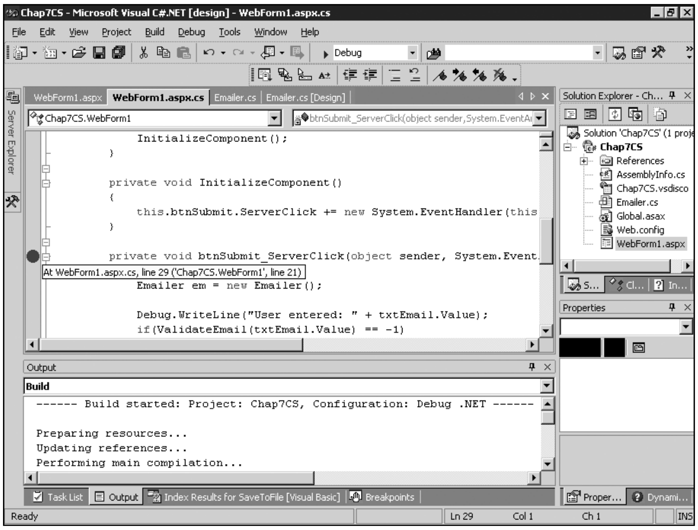
These examples are also extremely simple, but they demonstrate the features of the Visual Studio .NET IDE quite effectively.

In this example, you are listening to the `ServerClick` event of the Submit button, named `btnSubmit`. When the user clicks the Submit button, this event is fired. At this point, you can inspect what is in the text box on the form. If it does not contain an `@` symbol or a `.`, it cannot be a valid email address and you then report this back to the user with a `Response.Write` of an error message.

Next you will look at how the previously mentioned debugging tools can aid you in tracking down a problem in this simple page.

Setting a Breakpoint

Let's start out by setting a breakpoint on the `ServerClick` event of the Submit button. This is the `btnSubmit_ServerClick` function in either piece of code. To set a breakpoint, simply move the mouse cursor to the gray left margin in the code editor window, and click. This drops a red dot into the margin, signifying that a breakpoint has been set at that specific line. Figure 7.8 shows exactly where to set this breakpoint and what the margin will look like after clicking.



```
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            this.btnSubmit.ServerClick += new System.EventHandler(this
        }

        private void btnSubmit_ServerClick(object sender, System.Event.
At WebForm1.aspx.cs, line 29 ('Chap7CS.WebForm1', line 21)
            Emailer em = new Emailer();

            Debug.WriteLine("User entered: " + txtEmail.Value);
            if(ValidateEmail(txtEmail.Value) == -1)
```

```
------ Build started: Project: Chap7CS, Configuration: Debug .NET ------

Preparing resources...
Updating references...
Performing main compilation...
```

Figure 7.8 Setting the breakpoint in the example program.

Now go ahead and run the program. When Internet Explorer appears, enter some gibberish into the text box that does not contain either an `@` symbol or a `.`. Now click the Submit button. When this occurs, the Visual Studio .NET IDE should pop up with
the breakpoint line highlighted in yellow. The execution of your program has paused, and now you can use some of the other debugging features. You will look at the watch window next.

Watch Window

At the bottom of your screen, you will see a debugging window with some tabs below it. Click the one labeled Watch 1. If this tab is not available, you will find the same entry under the Debug menu as part of the Windows suboption. Figure 7.9 shows the menu entry.

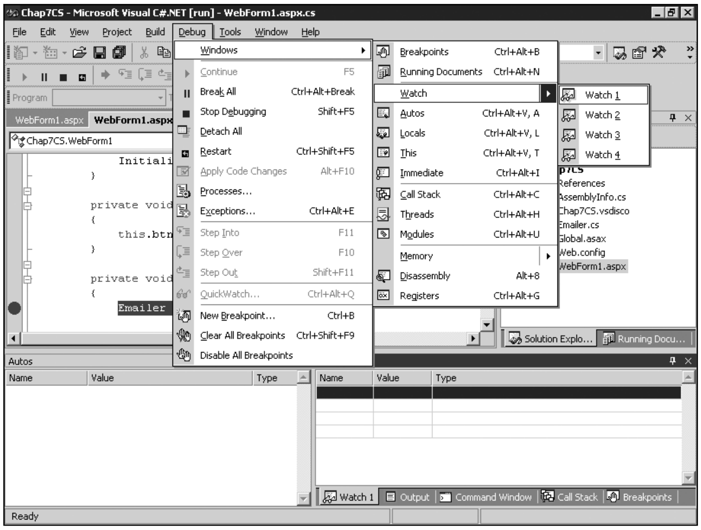


Figure 7.9 The watch window option under the Debug menu.

The watch window enables you to type in a specific variable name, control name, or other object and then see what value it contains and what type it is. For now, type in `txtEmail`. You will see that you can expand this entry into all the control's properties to see what they each contain. To save space, you might want to see only the `Value` property—in that case, you could enter `txtEmail.Value` as your watch name.

With the `txtEmail` control expanded, you can see that the `Value` property contains whatever you entered in the control on the client side. This can be extremely useful when debugging all types of controls to see what values they contain. It is always useful to start here when debugging almost any problem to make sure that the data isn't to blame. For example, you might be chasing after a problem only to find that the `Value` property is empty for some reason or that it does not contain the data that you think it does.

The other thing that you can do with the watch window is change the value of a variable. If you click the property value in the Value column in the watch window, you can enter a new value as you see fit. This might be helpful if you want to test a certain case in your logic that might be difficult to hit. For example, if an error is supposed to occur if a value equals –1, at this point you could change the value to –1 and continue execution to make sure that the code path is operating properly.

That's about it for the watch window. This is a feature that you will use quite a bit in your debugging. Remember that you can enter any variable or any object into the window and view any or all of its specific properties. Also note that you can change the values of any of these properties at any time.

The Command Window

If you have used Visual Basic, this will be a familiar sight. The command window was called the immediate window in Visual Basic, but its features are identical. This window enables you to issue commands to debug or evaluate expressions on the fly.

To display the window, either click the Command Window tab located at the bottom of your screen, or choose Windows, Immediate from the Debug menu at the top of your screen. Figure 7.10 shows where you can find the option under the Debug menu.

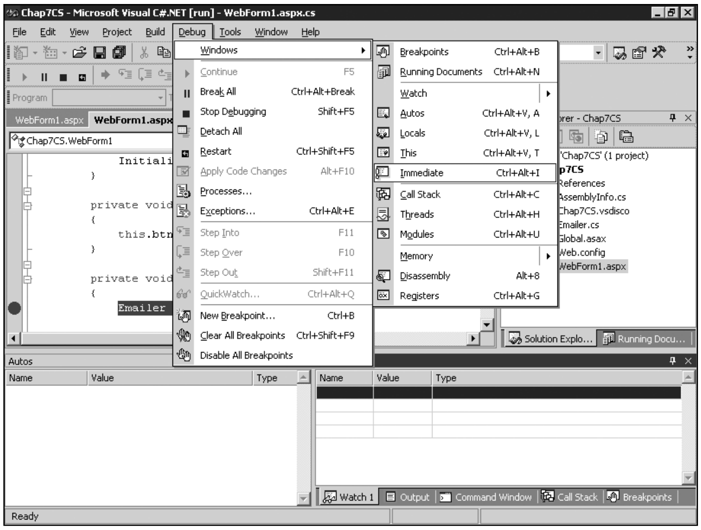


Figure 7.10 The immediate window option under the Debug menu.

To view the contents of a variable or object, just type its name into the command window. For example, typing `txtEmail.Value` while at the breakpoint displays the contents of the text box upon submission to the server.

Similar to the watch window, you can change the value of variables or object properties. To change the value of the form text box, you could enter `txtEmail.Value = "newvalue"`, which would set the string to `"newvalue"`.

What makes the command window a bit more exciting than the watch window is its capability to execute functions. For example, if you want to execute the `ValidateEmail` function in your code listing at any time, you could do it right from the command window: Just click in the window and call the function with the appropriate parameter. For example, if you type `ValidateEmail("test@myhost.com")`, you will see that it returns `0`. If you type `ValidateEmail("asdfasdf")`, it returns –1. So, you can test any function that you write right here without having the executing program call it explicitly—a great debugging aid.

Tracing

Tracing is a very simple method of debugging problems. Tracing can be used to print text statements during the execution of your code. This can be used as a log to see exactly what is happening at any point in your code. As you can see in the previous examples, in the `ServerClick` function of the Submit button, you are calling `Debug.WriteLine` with the value of the form's text box. This call spits out the value of the text box to the debug stream. The easiest place to see the debug stream is in the output window at the bottom of the Visual Studio .NET IDE.

This window can be displayed by either clicking the Output tab or choosing Output under the View menu and then choosing Other Windows. This window shows you all the debug statements that you have inserted into your code, as well as anything else that gets written to the debug stream by any other components that might be associated with the running project. Keep this in mind if you see a flood of messages that you didn't write into your code.

Execution Control

Before we start talking about the call stack, let's take a brief journey into the execution control features of the debugger. These features enable you to control exactly what is executed in your program—and in what order. They can also be used to trace deeply into certain portions of the code or skip over them, if that level of detail is unnecessary.

All these features can be found under the Debug menu at the top of your screen. All are also associated with keyboard shortcuts that vary depending on how you have configured Visual Studio .NET. The keyboard shortcuts are the easiest method of using these
features because they enable you to move through many lines of code in a very quick fashion. We recommend learning the keyboard shortcuts and using them while debugging your own code.

The three options are Step Into, Step Over, and Step Out. Step Into enables you to step one level deeper into the code at the current point of execution or, at the very least, move to the next statement. If you are about to call a function in your code, using Step Into continues execution at the first line of the called function.

Step Over does the opposite of Step Into. If you are about to call a function, using Step Over at this point does just that—it steps over execution of the function to the very next line of the function that you are currently executing. Now keep in mind that this does not mean that it will not execute the function—it just will not allow you to dig into it. It executes the function and moves on to the next line. This is quite useful when you know that a function or block of code is working correctly and you do not want to spend the time hitting every single line.

Step Out enables you to jump out of the current function that you are debugging and go one level up. Again, similar to the Step Over feature, this does not skip the execution of the remaining lines of code; they just execute behind your back, and your debugging cursor moves to the next line in the previous function you were in.

So how do you know which function you were previously in? That leads to the next debugging feature, the call stack.

Call Stack

The call stack shows the current function you are in and all functions that preceded it. When you call a function from a function from a function from a function, you have a call stack that is four levels deep, with the current function on the top. You can view the call stack window by clicking the Call Stack tab at the bottom of your screen or by choosing Call Stack from the Debug menu under Windows.

Continuing the previous example, stop execution again on the `btnSubmit_ServerClick` function and then trace into the `ValidateEmail` function. Now you have called a function from a function. Take a look at the call stack window. The top two levels should show you the `ValidateEmail` function, followed by the `btnSubmit_ServerClick` function. You will also see quite a few other functions that are called by the ASP.NET system processes.

Now go ahead and double-click the `btnSubmit_ServerClick` function. A green highlight appears over the point in the function that you currently are in. In this case, the call to `ValidateEmail` is highlighted because this is the exact position that you are currently at in that function.

This feature can be of use when you are debugging code that might not be all your own. If you are many levels deep into a function stack, you might need to know where you came from. By using this, you can trace back to the calling stack and see exactly who called you and with what information. After you have traced back to the call stack, you can use the watch window or the command window to inspect the local variables from the previous functions. This can be handy when you want to find where certain data values are coming from if they are wrong.

Feature Summary

That about wraps up the baseline features of the Visual Studio .NET IDE. Next you will look at how to add a C# or Visual Basic .NET component to your project and debug that simultaneously with your ASP.NET pages. The process is extremely stream-lined and quite
seamless, as you will soon see.

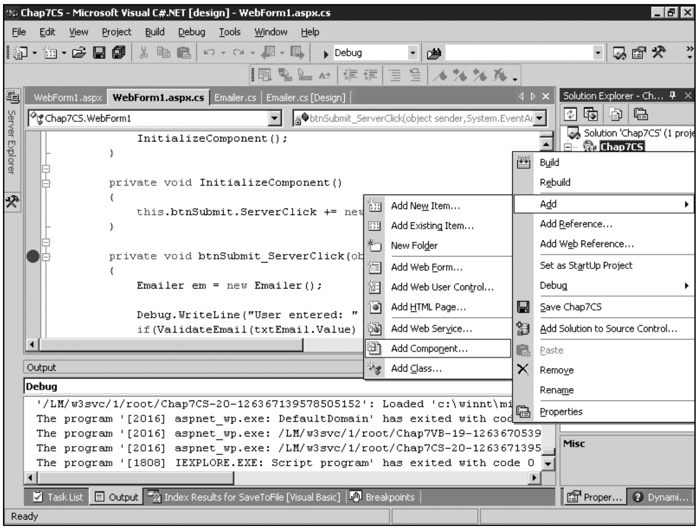
Inline Debugging of Components

If you tried debugging Visual Basic components within an ASP page in the previous version of Visual Studio, you will remember that it can be a pain to deal with. You need the Visual Interdev IDE open for debugging the ASP page, and you need the separate Visual Basic IDE open to debug the components at the same time.

The new Visual Studio .NET IDE makes this process remarkably simpler. You can add the component to your ASP.NET project, and debugging of that component can be done within the same IDE in sequence with your ASP.NET code. Let's look at how this is done. To do this, you will add a component to the previous project that actually sends out the email to the address provided.

Adding the Component

You will now add the component to the ASP.NET application. Just right-click your mouse on the project name (Chap5VB or Chap5CS, if you've named them what we called them) and choose Add Component under the Add submenu. Here, choose either a Visual Basic .NET component class or a C# component class, depending on which type of project you are currently doing. Name the component Emailer, for lack of a better name. Figure 7.11 shows the menu option to choose after right-clicking the project name.



```
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            this.btnSubmit.ServerClick += new

        }

        private void btnSubmit_ServerClick(ob
        {
            Emailer em = new Emailer();

            Debug.WriteLine("User entered: "
            if(ValidateEmail(txtEmail.Value)
```

```
'/LM/w3svc/1/root/Chap7CS-20-126367139578505152': Loaded 'c:\winnt\mi
The program '[2016] aspnet_wp.exe: DefaultDomain' has exited with cod
The program '[2016] aspnet_wp.exe: /LM/w3svc/1/root/Chap7VB-19-1263670539
The program '[2016] aspnet_wp.exe: /LM/w3svc/1/root/Chap7CS-20-1263671395
The program '[1808] IEXPLORE.EXE: Script program' has exited with code 0
```

Figure 7.11 Adding a new component to the project.

Now that you have added the component, it needs some code. Listing 7.4 is the C# version of the emailer, and Listing 7.5 is the Visual Basic .NET version. Reference whichever one is applicable for your project.

Listing 7.4 **Code for Emailer Component (C#)**

```
using System.Web.Mail;

namespace Chap7CS
{
    public class Emailer : System.ComponentModel.Component
    {
        private System.ComponentModel.Container components = null;

        public Emailer(System.ComponentModel.IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }

        public Emailer()
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            components = new System.ComponentModel.Container();
        }

        public void SendFormEmail(string toAddr)
        {
            MailMessage mm = new MailMessage();

            mm.To = toAddr;
            mm.From = "admin@domain.com";
            mm.Body = "This is a test message.  Exciting, isn't it?";
            mm.Subject = "Chapter 7 Test Message";
            SmtpMail.SmtpServer = "smtp.domain.com";
            SmtpMail.Send(mm);
        }
    }
}
```

Listing 7.5 **Code for Emailer Component (Visual Basic .NET)**

```
Imports System.Web.Mail

Public Class Emailer
    Inherits System.ComponentModel.Component

    Public Sub New(ByVal Container As System.ComponentModel.IContainer)
        MyClass.New()

        Container.Add(Me)
    End Sub

    Public Sub New()
        MyBase.New()

        InitializeComponent()
    End Sub

    Private components As System.ComponentModel.Container

    <System.Diagnostics.DebuggerStepThrough()> Private Sub
    ➥InitializeComponent()
      components = New System.ComponentModel.Container()
    End Sub

    Public Sub SendFormEmail(ByVal toAddr As String)
        Dim mm As MailMessage = New MailMessage()

        mm.To = toAddr
        mm.From = "admin@domain.com"
        mm.Body = "This is a test message.  Exciting, isn't it?"
        mm.Subject = "Chapter 7 Test Message"
        SmtpMail.SmtpServer = "smtp.domain.com"
        SmtpMail.Send(mm)
    End Sub
End Class
```

The code here is pretty simple. Each version contains a function called `SendFormEmail` that takes the email address to send to as a parameter. Then you use the `MailMessage` and `SmtpMail` objects from the `System.Web.Mail` assembly to form the email message and send it out using a valid SMTP server. To get this to work in your environment, be sure to replace the `SmtpMail.SmtpServer` value with the SMTP server of your local network.

You will need to modify your `btnSubmit_ServerClick` function to create an instance of this component and call the `SendFormEmail` method to make it happen. Listing 7.6 gives the code for the modified `btnSubmit_ServerClick` in C#, and Listing 7.7 gives the same code in Visual Basic .NET.

Listing 7.6 **Modified Code for *btnSubmit_ServerClick* (C#)**

```
private void btnSubmit_ServerClick(object sender, System.EventArgs e)
{
    Emailer em = new Emailer();

    Debug.WriteLine("User entered: " + txtEmail.Value);
    if(ValidateEmail(txtEmail.Value) == -1)
        Response.Write("The supplied email address is not valid.");
    else
    {
        em.SendFormEmail(txtEmail.Value);
        Response.Write("The email was sent successfully.");
    }
}
```

Listing 7.7 **Modified Code for *btnSubmit_ServerClick* (Visual Basic .NET)**

```
Private Sub btnSubmit_ServerClick(ByVal sender As System.Object, ByVal e
➥As System.EventArgs) Handles btnSubmit.ServerClick
    Dim em As Emailer = New Emailer()

    If txtEmail.Value.IndexOf("@") = -1 Or _
        txtEmail.Value.IndexOf(".") = -1 Then
        Response.Write("The supplied email address is not valid.")
    Else
        em.SendFormEmail(txtEmail.Value)
        Response.Write("The email was sent successfully.")
    End If
End Sub
```

Debugging the Component

Now we get to the cool part. You can debug this component while you debug the ASP.NET page and its respective code-behind file. To prove this, set a breakpoint on the `btnSubmit_ServerClick` in the code-behind file and then start the program.

When Internet Explorer appears, enter a valid email address in the appropriate box, and click the Submit button. Immediately, the breakpoint on the `btnSubmit_ServerClick` function should fire and the program is paused on that line. Now step to the point where the current function is about to call the `Emailer.SendFormEmail` function. At this position, do a Step Into. You will see that the source code to the Emailer component appears with the code pointer at the top of the `SendFormEmail` function.

From here, you can use all the techniques mentioned earlier to inspect variables, set trace statements, modify variable values, and so on. It couldn't be easier! Say goodbye to multiple programs being open simultaneously and other configuration issues that make your life difficult.

Remote Debugging

Every time we have a discussion with someone regarding debugging ASP pages, we always ask if that person has ever tried to set up ASP debugging on the local machine. The response usually is "yes." We then follow up with the question of if that person has ever gotten it to work. The number who answer "yes" to that question is much lower. Finally, we ask if that person has ever gotten ASP debugging to work remotely. We have yet to find someone who has gotten it to work properly and consistently.

With ASP.NET and Visual Studio .NET, that all changes. It finally works. And it couldn't possibly be easier to install, configure, and use.

Installation

When you install Visual Studio .NET on your server, all you need to do is install both Remote Debugging options listed under the Server Components option during the install
procedure.

Setup

To configure remote debugging, the only thing you need to do is place your user account into the newly created Debugger Users group both on the client machine and on the server machine. This can be done using the standard user configuration tools that are part of whichever Windows operating system you are using.

Using It

This is the easiest part of all. To use the new remote debugging features, simply create a project on your client computer that points to the project running on the server. This can be done by choosing Open Project from the Web from the File menu. Just type in the name of the server where the project resides, and you will be presented with a list of projects currently residing on that server. Choose the appropriate one.

If you are not connecting to an existing project, you can create a brand new project on the server, and remote debugging will still take place.

Next, set a breakpoint on the line where you want to stop, or simply start the application running. It will connect to the server and bring up an Internet Explorer window, as usual. The big difference here is that the application is running entirely on the server. When you hit your breakpoint, you are hitting it on the server, in the server's memory space. The same goes for components and anything else that you might be debugging. Everything that can be debugged in Visual Studio .NET locally can now be debugged remotely on any server where the remote debugging options have been installed.

And that's it! It's almost completely automatic. I wish it was this easy in Visual Studio 6.0—it's a huge time saver and a powerful tool.

Summary

In this chapter, you looked at many of the debugging features found in the new Visual Studio .NET IDE. You should now be familiar with things such as the watch window, the command window, breakpoints, variable inspection, and variable modification as applied both to debugging ASP.NET pages and Visual Basic .NET and C# components. With these concepts in mind, you are prepared to start debugging your own projects, and you are prepared for what is explained in the remainder of this book.

In the next chapter, we discuss how you can use the Windows NT and Windows 2000 Event Log to aid in tracking down troublesome code in your projects.