Bonus Issue

3

UNDERSTANDING SHAREPOINT JOURNAL

Ayman Mohammed El-Hattab

SharePoint Troubleshooting

UNDERSTANDING SHAREPOINT JOURNAL

SharePoint Troubleshooting

Rubina Ranasgt. 10
N-0190 Oslo, Norway
Phone +47 91 39 85 86 • Web http://www.understandingsharepoint.com/

Credits

About the Author

Ayman El-Hattab is a SharePoint developer and speaker. He is a Microsoft Certified Solution Developer as well as a Microsoft Certified Technology Specialist in SharePoint. Ayman writes articles about SharePoint and its related technologies for online magazines and speaks at numerous user groups and other offline communities. He is the founder of SharePoint4Arabs.com and organizes events for EGYSUG. You will also find him as an active participant in the MSDN forums.

About the Technical Reviewer

Bjørn Christoffer Thorsmæhlum Furuknap is a senior solutions architect, published author of *Building the SharePoint User Experience*, speaker, and passionate SharePointaholic. He has been doing software development professionally for 16 years for small companies as well as multinational corporations. He has also been a teacher at a college-level school, teaching programming and development to aspiring students, a job that inspired him to begin teaching what he has learned and learns every day.

About *Understanding SharePoint Journal*

Understanding SharePoint Journal is a periodical published by UnderstandingSharePoint.com. The journal covers few topics in each issue, focusing to teach a deeper understanding of each topic while showing how to use SharePoint in real-life scenarios.

You can read more about *USP Journal*, as well as get other issues and sign up for regular updates, discounts, and previews of upcoming issues, at http://www.understandingsharepoint.com/journal.

Other Credits

A great big thanks to Kim Wimpsett for doing the copyedit. The quality of work in this issue is greatly attributed to her skill.

Table of Contents

About the Author i
About *Understanding SharePoint Journal* i
Other Credits i
Something Went Wrong! 1
Debugging Your SharePoint Code 3
- Fundamental Terms 3
- Debugging ASP.NET Applications 4
- ASP.NET Debugging vs. SharePoint Debugging 5
- Debugging Assemblies Located in GAC 5
- Attaching a Debugger to Worker Processes 6
- Tips and Tricks 8
- Debugging Inline Code in Your Custom Pages 8
- Debugging Custom Timer Jobs 10
- Cached Assemblies! 13
- Forcing the Execution of Timer Jobs 13

Unified Logging Services (SharePoint Trace Logs) 13
- Recommended Configurations 14
- Notes 14
- Logging to SharePoint Trace Logs 15
- Tracing Service Lost Trace Events 16

Windows Event Logs 17
Post-deployment Troubleshooting 17
- Debug Calls vs. Trace Calls 18
- Global Exception Handling 18

SharePoint Troubleshooting Toolbox 19
- Debug Config Feature 19
- ULS Logs Viewers 19
- Debugger Feature for SharePoint 20
- WSPBuilder Extensions for Visual Studio 20
- Assembly Binding Log Viewer 21
- SharePoint Logging Library 22
- SharePoint Manager 2007 24
- .NET Reflector 24

Introduction

Asking the right question at the right time to the right people can trigger a landslide.

In software development, a *death march* is a project that is destined to fail. Typically, bugs are the reason why you are subjected to such lengthy and tiresome projects—complete with crabby stakeholders, grouchy quality engineers, missed deadlines, and late nights.

Believe it or not, most teams consume an average of 50 percent of their development life cycle troubleshooting and debugging. Bugs can truthfully make your life joyless, because if enough of them creep into your released software, customers might stop using your services, or you could even lose your job.

Usually, the most complex part of debugging is finding the bug in the source code. Once it is found, correcting it is relatively easy. For me, I don't consider debugging to be separate step in software development; instead, it's a fundamental and integral part of the entire life cycle.

Despite the existence of luminous books dedicated to SharePoint development, none of them dives into troubleshooting in enough depth. The initiative for this issue came out of my late nights as a software engineer trying to troubleshoot SharePoint solutions to ship high-quality products on time.

Since the release of WSS 3.0 and SharePoint 2007, I was lucky to learn tips, tricks, and techniques that really speed up debugging, and my hope is that presenting this information in this bonus issue will help you learn how to author applications with fewer bugs in the first place and that when you're asked to debug, you can do it much faster and smarter. You can get even more tips, tricks, and techniques at my blog at http://www.aymanelhattab.com. If you have any questions related to .NET/SharePoint development, feel free to drop me an email at ayman.elhattab@gmail.com.

Chapter 1

SharePoint Troubleshooting

Give a man a fish, and you feed him for a day; teach a man to fish, and you feed him for a lifetime.

Earlier versions of SharePoint did make some use of the .NET Framework, but SharePoint 2007 has been rebuilt from ground up on top of it, meaning SharePoint applications are in fact tremendously powerful ASP.NET applications. Although SharePoint exhibits a great deal of power, that doesn't mean you won't be authoring ASP.NET code anymore. Quite the contrary—anything you can do with SharePoint, you can do with ASP.NET, and vice versa. That being said, there are some differences when it comes to debugging and troubleshooting with SharePoint that you should know.

ICON KEY

- Valuable information
- Test your knowledge
- Exercise
- Caution

Something Went Wrong!

Generally speaking, an error message will be your first signal that something is not behaving as expected in your application. Unluckily, SharePoint uses an annoying error page to illustrate that a problem occurred. If you spend any time writing custom SharePoint code, at some point you will probably receive a screen like the one shown in Figure 1.

FIGURE 1. GENERIC ERROR PAGE

This error page means that an assembly has thrown an unhandled System.Exception. Sometimes the user-friendly error page will note the nature of the error if the code that threw the exception used something more precise than System.Exception or if it included a message in the Exception class constructor.

We need to get a more detailed error message than the irksome "Unexpected error has occurred" to be able to analyze the error.

Exercise

1. Open the web configuration file (web.config) from the virtual directory containing your SharePoint application located at [Local_Drive]:\Inetpub\wwwroot\wss\VirtualDirectories\[YourVirtualDirectory].

2. Search the web configuration file for customErrors, and set the mode attribute of the customErrors element to Off or RemoteOnly.

Setting customErrors to Off shows the full error to every client every time. This is typically used for the duration of development since there are no clients using it.

Setting customErrors to RemoteOnly permits you to display custom errors only to remote clients. This means that if you are browsing your SharePoint site locally, you will get fully detailed error messages. However, everyone else will receive the standard SharePoint page.

3. Search the file for SafeMode, and set the CallStack and AllowPageLevelTrace attributes of the SafeMode element to True.

Now you will receive the regular ASP.NET error page when something goes wrong, as shown in Figure 2.

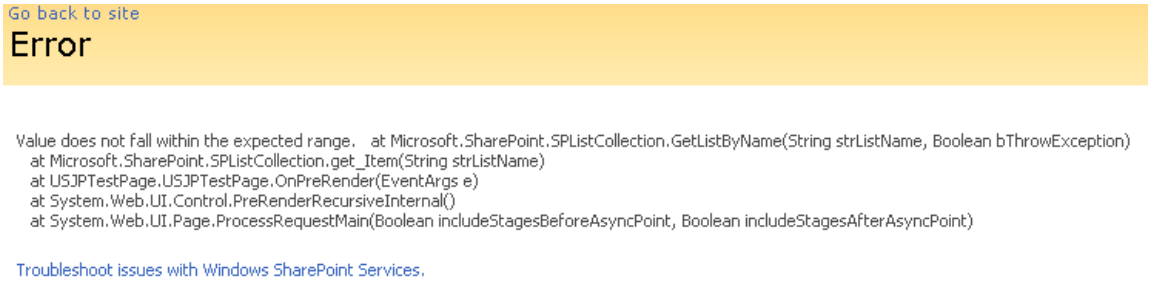

FIGURE 2. ASP.NET ERROR PAGE WITH A STACK TRACE

That's more descriptive, isn't it?

Debugging Your SharePoint Code

Now you have the standard ASP.NET error page with a stack trace, but you need to trace through your code to diagnose the issues rather than guessing the root causes.

Fundamental Terms

Before diving in, I'll define some terms that I mention frequently throughout this issue.

Application Pools

A classic example that I always utilize to discuss application pools is a web hosting company that hosts many websites on a common server, and therefore all the websites hosted on this server share the system resources. If one of the hosted websites had a memory leak, this potentially takes memory away from the other hosted sites, causing errors and inconveniences.

Prior to IIS 6.0, at least, that was the case. Too many server resources were shared between websites, and it was too easy for an error to creep in and damage the entire server process.

Fortunately, IIS 6.0 introduced a new concept called *application pools*. Application pools form boundaries around applications, separating them from each other even though they are being hosted on the same server.

Each application pool is given its own set of server resources (including memory). That way, the hosting company can now host each website in its own application pool to accomplish security, reliability, and availability and to prevent the leaky site from affecting other sites.

Worker Process

IIS uses http.sys, which is part of the networking subsystem of Windows, to listen to requests and route them to the appropriate request queue. Each request queue corresponds to a particular application pool, and each pool is served by *worker processes* that handle requests, run application code, and return responses.

Web Garden

An application pool that uses more than one worker process is called a *web garden*. You can have multiple worker processes serving one request queue, which makes IIS not only faster but also more reliable because when a process fails, the other keeps going.

Debugging ASP.NET Applications

Now that I have defined the terms that might not be familiar to you, let's see how we can debug traditional ASP.NET applications. Visual Studio 2005 introduced a built-in development web server for testing and debugging your web applications without having IIS installed on your development machine. The server is named WebDev.WebServer.exe, and it gets installed by Visual Studio into the framework version's directory.

When you set the breakpoints in your source code and press F5 to run your application, WebDev.WebServer.exe starts a process to execute your web application, and the code breaks whenever it hits a breakpoint. To investigate which process is running for debugging, run the web application from Visual Studio. You will get a pop-up notification like the one shown in Figure 3.

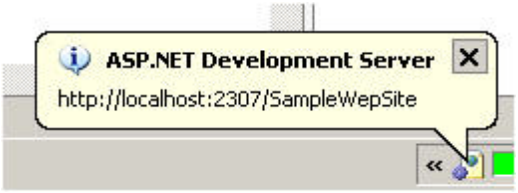

FIGURE 3. STUDIO INTEGRATED DEVELOPMENT WEB SERVER

Double-click the icon, and a pop-up window will denote that the Visual Studio integrated engine has started a process to execute your application.

FIGURE 4. DEVELOPMENT SERVER PROCESS DETAILS

ASP.NET Debugging vs. SharePoint Debugging

Unfortunately, debugging SharePoint applications is not as straightforward as pressing F5 from Visual Studio. As I stated earlier, a SharePoint application is just a powerful ASP.NET application hosted under IIS. In other words, whenever we create a new SharePoint web application, behind the scenes a new IIS website gets created.

So, if we have several running processes in IIS and we need to debug the application, first we need to attach to the correct process in Visual Studio; this in fact applies to any application hosted under IIS, not only SharePoint.

> Note
>
> The following debugging instructions, tips, and tricks apply to any custom SharePoint code you write within custom web parts, pages, event receivers, or user controls, but they don't apply to custom timer jobs, which we will cover in detail later in this issue.

Debugging Assemblies Located in GAC

There is just one more thing you need to do prior to attaching a debugger to the correct worker process so you can debug your assemblies located in the GAC. In fact, it's a common myth among SharePoint developers that to debug assemblies that have been deployed to the GAC, you need to copy the debug

symbols (the PDB file) to the GAC as well. This was true in the early days of .NET, but this is not true anymore.

To permit Visual Studio to debug your assemblies that have been deployed to the global assembly cache, you need to do the following:

Exercise

1. Select Tools > Options > Debugging.
2. You'll find an option labeled Enable Just My Code (Managed only), as shown in Figure 5, which is selected by default. Deselect this option to be able to debug the assemblies located in the GAC.

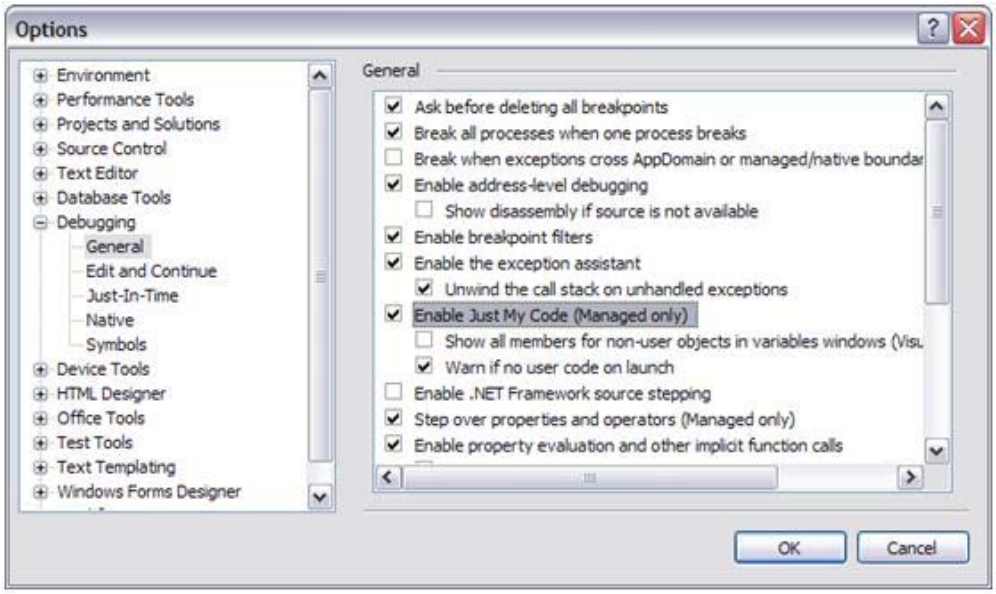

FIGURE 5. THE ENABLE JUST MY CODE (MANAGED ONLY) OPTION

> **Note**
>
> If you keep the default setting, Visual Studio will skip loading the symbols, and you will not be able to step through your code.

Attaching a Debugger to Worker Processes

Exercise

After enabling Visual Studio to debug assemblies deployed to GAC, now we can attach a debugger to worker processes:

1. In Visual Studio, select Debug > Attach to Process, as shown in Figure 6.

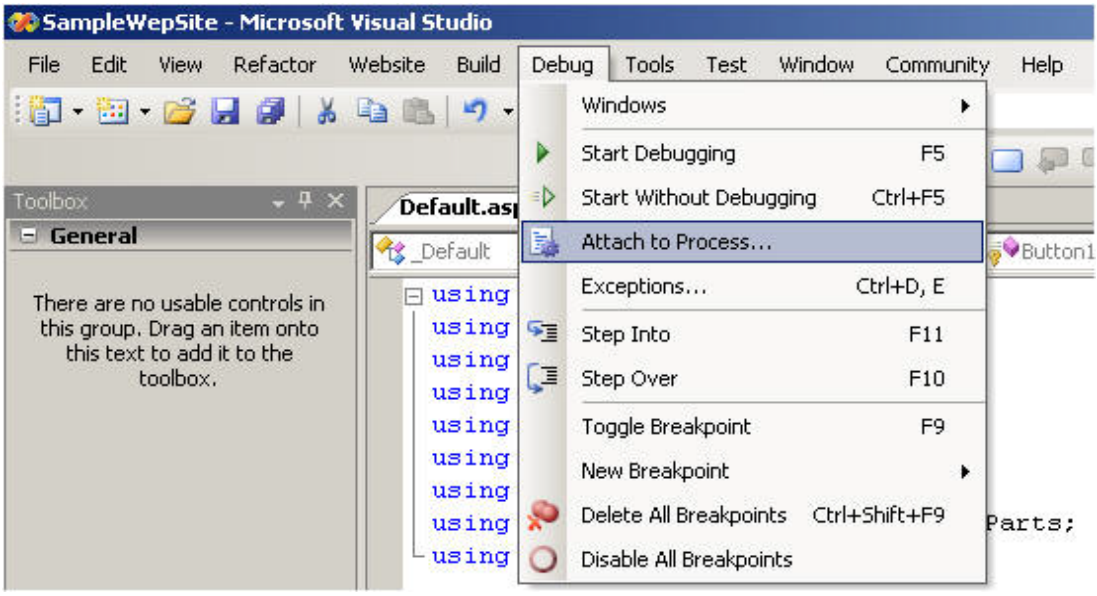

FIGURE 6. ATTACHING A DEBUGGER TO PROCESSES

2. A window will pop up listing the available processes. Find and select the w3wp.exe process, and click the Attach button.

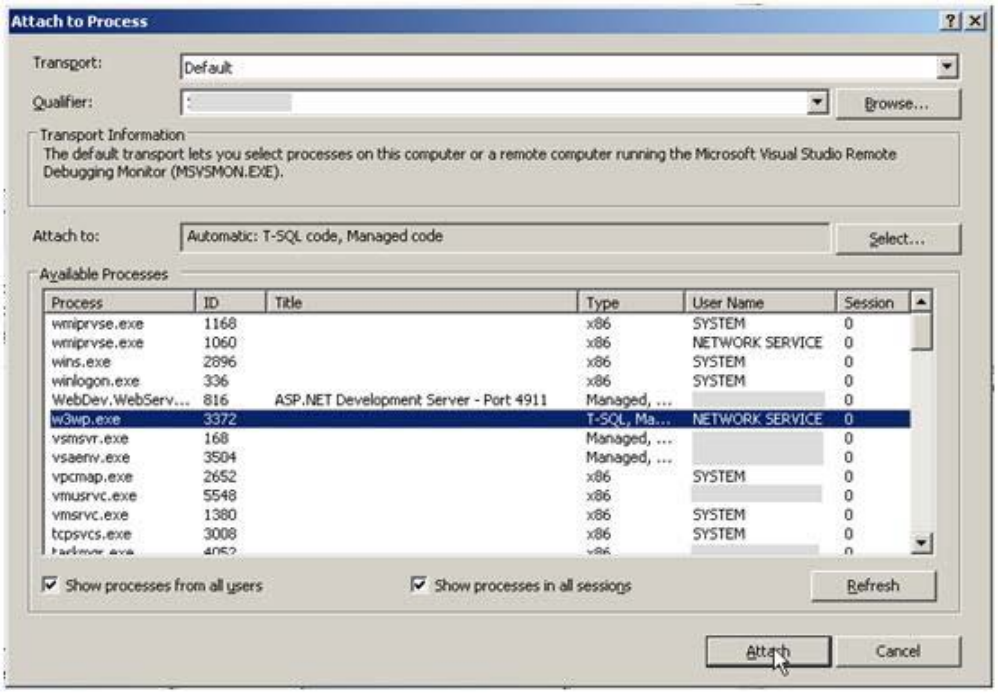

FIGURE 7. ATTACH TO PROCESS WINDOW

⌘ Tip

Tips and Tricks

Here are some tips for attaching debuggers to worker processes:

1. If no worker processes are listed, select the “Show processes from all users” and “Show processes in all sessions” checkboxes.
2. If you still cannot find a w3wp.exe to attach to, navigate to your application’s site through your preferred web browser, and then return to Visual Studio, click Refresh, locate the w3wp.exe process, and click the Attach button.
3. If you have multiple SharePoint web applications and those web applications have their own application pools, you will likely find multiple instances of w3wp.exe in the list of available processes in the Attach to Process window. Therefore, you can attach to all those instances by holding down the Ctrl key and selecting all the w3wp.exe instances, or you can choose just one as follows:
    a. Open the command prompt window, and run IISAPP to get a list of the current instances of w3wp.exe.

FIGURE 8. IISAPP OUTPUT

    b. Note the PID of the instance that corresponds to your web application.
    c. Now return to Visual Studio, and attach to the w3wp.exe instance with an ID equivalent to the PID you noted in the previous step.

Debugging Inline Code in Your Custom Pages

Have you ever tried to debug the inline code of your custom pages? If you try the previous instructions, you will not be able to step through your code.

You must perform an extra step before debugging your inline code.

Exercise

Open the web configuration file of the virtual directory containing your web application. Refer to the previous section for more help in finding this file.

1. Locate the Compilation element.
2. Change the debug mode from false to true, as shown in Figure 9 and Figure 10.

```
<compilation batch="false" debug="true">
```

```
    <script runat="server" language="C#">
        protected void Page_Load(object sender, EventArgs e)
        {
            Response.Write("Hello SharePointers");

        }
    </script>
```

FIGURE 9. BEFORE CHANGING THE DEBUG MODE

```
    <script runat="server" language="C#">
        protected void Page_Load(object sender, EventArgs e)
        {
            Response.Write("Hello SharePointers");

        }
    </script>
```

FIGURE 10. AFTER CHANGING THE DEBUG MODE

Alternatively, if you have a few custom application pages with inline code, you can set the debug element of the Page directive on a page-by-page level to true rather than turning it on for the whole web application, as shown here:

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

So, Why Set debug="true"?
Setting the debug attribute of the Compilation element or the Page directive to true instructs ASP.NET to generate debug symbols when compiling pages.

So, unless you are trying to debug inline code within a ghosted page, there is no need to set the debug attribute of the Compilation element to true. I am stressing this because I have seen many online tutorials improperly state that you must set the attribute to true in order to step through your code. This is not true!

Note

Although it makes sense to set the debug="true" during development inline code, it is a common mistake to leave this development-time setting on in production environments; this leads to slower code execution because of the additional debug paths being enabled. This will also cause performance issues because much more memory will be used by the application at runtime.

Debugging Custom Timer Jobs

SharePoint timer jobs are tasks executed on a scheduled basis by the Windows SharePoint Services timer service (owstimer.exe). They are analogous to scheduled tasks, and you can create them in any version of Windows, but SharePoint timer jobs come with many more benefits. SharePoint relies on timer jobs for a number of its functionality areas, and you can even broaden those functionalities and create your custom timer job to introduce new features.

For instance, last month I developed a custom timer job that runs on a daily basis to query all the SharePoint lists and libraries in all the site collections and insert corresponding records into a custom database table for reporting purposes.

Unfortunately, SharePoint timer jobs are tricky when it comes to debugging; it is not about attaching a debugger to w3wp.exe like what we did earlier in this issue. Troubleshooting my custom timer jobs has caused me a lot of headache; that is why I am determined to share with you all the tricks that I have learned in the course of developing a reporting timer job.

To debug custom timer jobs, follow these steps:

1. Insert an always-failing assertion statement at the beginning of the Execute method. You should use Debug.Assert(false) rather than Trace.Assert(false) since debug calls are detached from the release builds; then you don’t have to remove them from your code before moving to staging or production.

```
public override void Execute(Guid targetInstanceId)
{

    System.Diagnostics.Debug.Assert(false);
    System.Diagnostics.Debug.WriteLine(SPContext.Current.Site.WebApplication.Name);

}
```

FIGURE 11: ASSERT STATEMENT

2. Every time the timer job starts, you will get a pop-up window like the one shown in Figure 12. This window hinders the execution of the timer job until you close the message box. Note that timer jobs run in parallel, so this window will never halt the execution of other timer jobs running simultaneously. For now, leave this window alone.

FIGURE 12. ASSERTION FAILED

3. Select Debug > Attach to Process, and attach a debugger to the Windows SharePoint timer service (owstimer.exe). Make sure that the "Show process from all users" checkbox is selected if owstimer.exe is not listed.

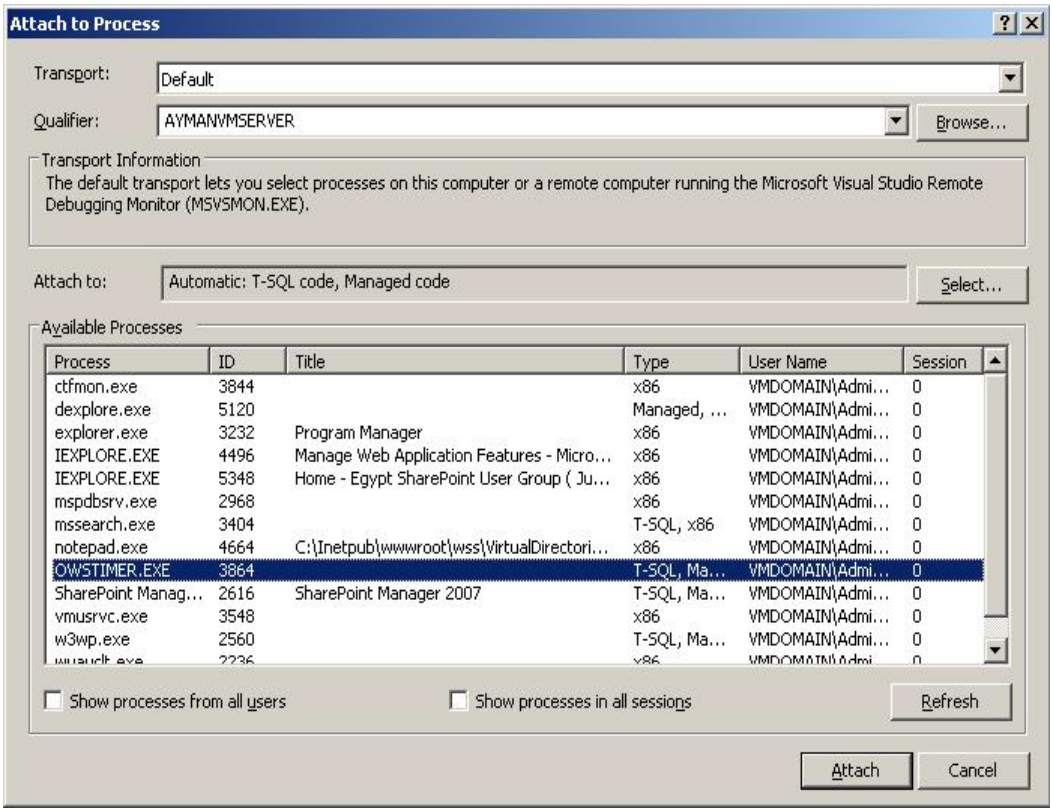

FIGURE 13. ATTACHING A DEBUGGER TO THE SHAREPOINT TIMER SERVICE

4. Click the Ignore button in the assertion window.

There you go!

```
public override void Execute(Guid targetInstanceId)
{

    System.Diagnostics.Debug.Assert(false);
    System.Diagnostics.Debug.WriteLine(SPContext.Current.Site.WebApplication.Name);

}
```

FIGURE 14. STEPPING THROUGH CUSTOM TIMER JOB CODE

You might be wondering why you should insert the assert statement at the beginning of the Execute method of the timer job. It is true that you can just attach a debugger to owstimer.exe and wait for the subsequent cycle of the timer job, but it could be tough to find out whether the application has been effectively attached to the process since the jobs may not fire for several minutes. You could be left with Visual Studio attached to the owstimer.exe

process with breakpoints set and wondering whether the job is running or be wondering whether the breakpoints are not being hit because of some problem with loading the symbol files.

Cached Assemblies!

⌘ Tip

Any time you update your custom timer job class and deploy the assembly to the global assembly cache, you must restart all the timer services in the farm. If you don't restart the timer service, it will run the old copy of your timer job class. Yes, the timer service caches the assemblies. Restarting the SharePoint timer service is the only way to refresh the assembly cache and force it to use your updated assembly. You can achieve that from the command line using the following commands:

```
net stop sptimerv3

net start sptimerv3
```

Forcing the Execution of Timer Jobs

You will probably need to execute your timer jobs outside of their scheduled times when you are developing, troubleshooting, or testing your custom timer jobs. Unhappily, this is not possible from Central Administration, but you can work around that using the SharePoint object model. For more information, refer to the following blog post:

http://www.sharepoint4arabs.com/AymanElHattab/Lists/Posts/Post.aspx?ID=61

Unified Logging Services (SharePoint Trace Logs)

Every now and then you will experience uninformative error screens while creating a site collection, activating a feature, or even deploying solution packages. For instance, a couple of months ago I got an irritating "File not found" error while attempting to create a site collection from an out-of-the-box site definition. It was very hard to troubleshoot because it didn't involve any custom code on my side.

SharePoint trace logs come to the rescue when you need to troubleshoot errors taking place before and after custom code; these trace logs are very nifty during the deployment and provisioning operations because they are the main sources of information about everything happening in SharePoint.

Most SharePoint components and products use the Unified Logging Services (ULS). These ULS logs are normal text files located at 12\Logs. You can configure ULS via the "Diagnostic logging" link under Logging and Reporting in the operations section of SharePoint Central Administration to be in command of the verbosity of events captured in the log files. You can also modify the settings per category or for all the categories, as shown in Figure 15, thus modifying the number of events captured. The list entries are sorted from most critical to least critical.

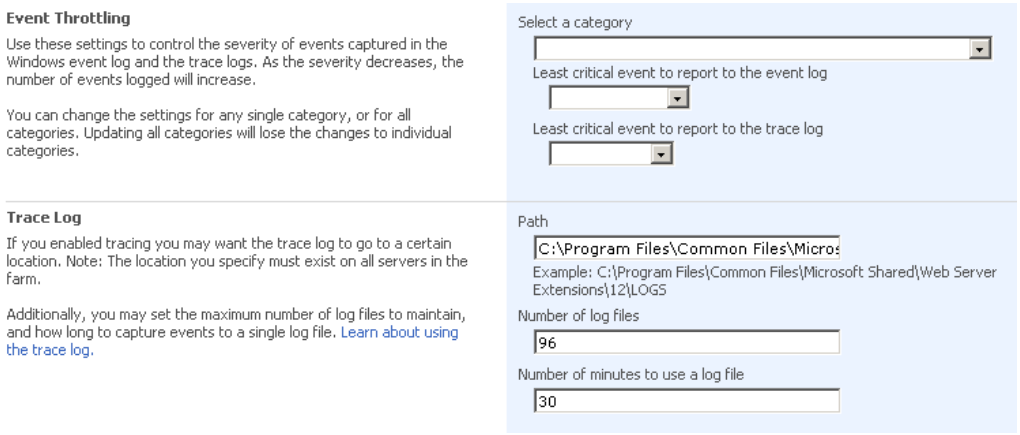

FIGURE 15. CONFIGURING DIAGNOSTIC LOGGING THROUGH CENTRAL ADMINISTRATION

Recommended Configurations

Event Throttling settings: I always just set all the categories to Verbose at development time because it is difficult to specify the categories needing to change when a problem occurs. In production environments, you should throttle from Audit Failure and above to capture warnings and errors.

"Number of log files" and "Number of minutes to use a log file": The default for these two settings gets you around two days; however, I recommend just a couple of log files and 5–10 minutes for each. This will save you a lot of reading compared to the default setting.

Notes

⌘ **Tips**

The following are some tips that you should consider when configuring SharePoint diagnostic logging:

- If you want to tweak the verbosity setting per category, make sure to leave the default value of the General category for trace logging as it is

(Verbose) because entries about the provisioning and deployment processes are logged under the General category. Also, know that updating all categories will make you lose the changes you made to individual ones.

Creating custom categories for diagnostic logging: http://blogs.prexens.com/Lists/Posts/Post.aspx?List=7a299699%2Df8da%2D4559%2D920c%2Dbda481608691&ID=11

- You can create your own diagnostic logging categories and get them to appear in the categories drop-down list by implementing a couple of interfaces. Before googling and investigating the SDK to create your custom categories, I recommend starting with the Prexens team blog post titled "How to create custom categories for SharePoint diagnostic logging with IDiagnosticsManager and IDiagnosticsLevel." It will save you lots of time.
- For more information about parsing and reading SharePoint trace logs, please refer to the "Troubleshooting Toolbox" section later in this issue.

Logging to SharePoint Trace Logs

You can also write code that incorporates logging to MOSS logs. Writing to the same trace log alleviates the need for developers to log their development information in other places such as the Windows Event Log, which is more commonly used by system administrators.

```
try
{
    // SPSite site = SPContext.Current.Site;
}
catch (Exception ex)
{

    Microsoft.Office.Server.Diagnostics.PortalLog.LogString
    ("Exception Happened : {0} --> {1} ", ex.Message,
    ex.StackTrace);
    // throw new SPException("Unknown Error Occured. Please
    contact the adminsitartor");
}
```

FIGURE 16. USING THE PORTALLOG CLASS TO LOG MESSAGES TO ULS

Regrettably, this method has several limitations:

You cannot set the log event level (for example, Low, Medium, and Critical) because it will always display the error level in the trace logs as High.

In addition, you cannot manage the trace category because it will always display the category as General.

The logger is located at 12\ISAPI \Microsoft.Office.Server.dll, and therefore it's available only with the MOSS install, not WSS 3.0.

What if you need to write into the logs while regulating the error level and the error category?

SharePoint Logging Library
http://sploggginglibrary.codeplex.com/

I recently shared a library at CodePlex that I have used frequently during the past year. The SharePoint Logging Library makes use of the TraceProvider class that was introduced in WSS SDK and helps you log debug information, warnings, or exceptions into SharePoint logs. For additional information about the SharePoint Logging Library, refer to the "Troubleshooting Toolbox" section later in this issue.

Tracing Service Lost Trace Events

Sometimes you will encounter something very weird in the ULS logs. The only entries that are logged are "Tracing service lost trace events" log entries. To resolve this issue, just restart your Tracing service. You can do that from the command line using the following commands:

`net stop sptrace` (to stop the tracing service)

`net start sptrace` (to start the tracing service)

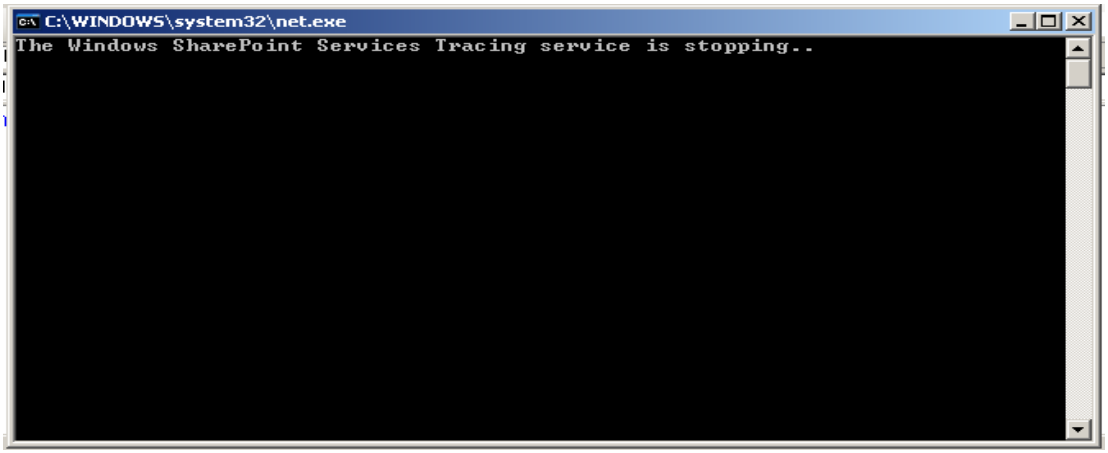

FIGURE 17. STOPPING THE TRACING SERVICE THROUGH THE COMMAND LINE

This will do the trick for you.

Note

Sometimes you need to delete all the trace log files, especially when the files get larger and larger. This is a little bit tricky because Windows will delete all the files except for the file that is currently in use by the tracing service. To delete this locked file, stop the tracing service, delete the file, and start it again.

Windows Event Logs

You should consider the Windows Event Log part of your common troubleshooting practice because it sometimes comprises valuable entries that SharePoint is unable to log in the trace logs. You can also configure the severity level from the Central Administration site in the same way you do for ULS.

It is possible to log errors to the System event log from your custom code, but you need to promote the privileges as shown in Figure 18 since your server might not allow code running in the security context of the logged-in users to write entries to the Event Log. The SPSecurity.RunWithElevatedPrivileges method executes the code that writes those entries with full control rights even if the logged-in user does not have sufficient permissions.

```
try
{
    SPSite site = SPContext.Current.Site;
}
catch (Exception ex)
{

    SPSecurity.RunWithElevatedPrivileges(delegate()
    {
        System.Diagnostics.EventLog.WriteEntry("Your Web Part Name",
            ex.Message);
    });

}
```

FIGURE 18. LOGGING ENTRIES TO THE WINDOWS EVENT LOG

Post-deployment Troubleshooting

Post-deployment troubleshooting is one of the key design points to consider in any large-scale application. SharePoint applications must be designed so that

they produce the necessary information to keep them running smoothly for years after they are put into production.

In this issue, I have shown you two ways to achieve that; the first one was logging information and exceptions into SharePoint trace logs using the SharePoint Logging Library, and the second was writing entries into Windows Event Log. In this section, I will discuss two other techniques that really help you monitor the application after going live.

Debug Calls vs. Trace Calls

DebugView
http://www.microsoft.com/technet/sysinternals/utilities/debugview.mspx

System.Diagnostics.Debug and trace statements are very valuable in this case. Used in conjunction with DebugView, you can conclude what has gone wrong. Debug calls are removed from the release builds, so you should use them in your development environment, but trace calls remain in release code and so will add overhead to the code execution. However, trace calls are very helpful when you cannot attach a debugger (for instance, in the staging and production environments). Figure 19 shows the trace output using DebugView.

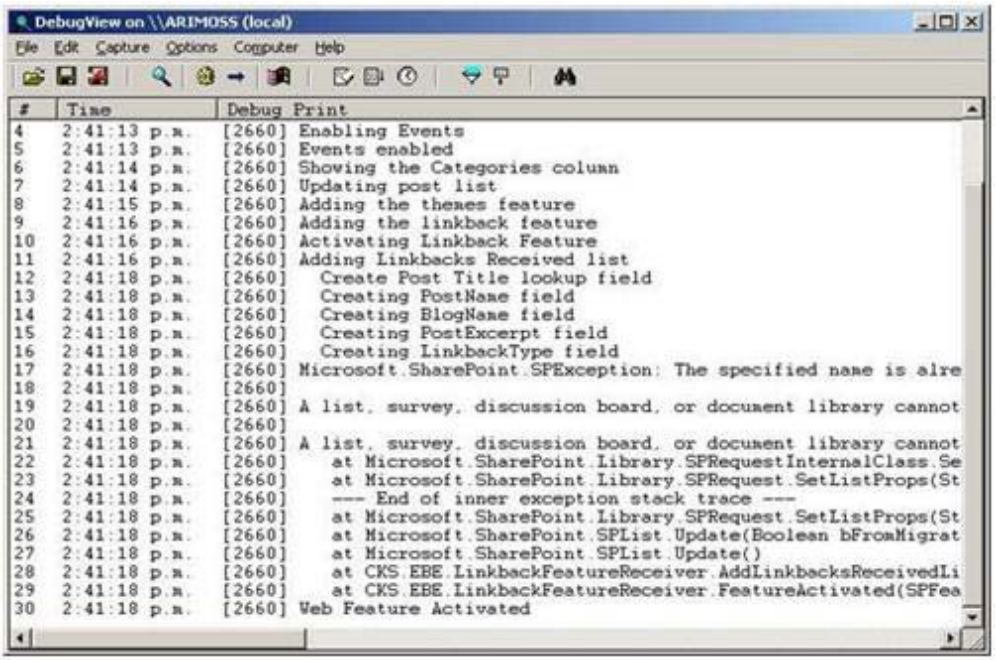

FIGURE 19. DEBUGVIEW IN ACTION

Global Exception Handling

In many cases, you might want some form of centralized exception handling rather than surrounding every piece of code with Try Catch blocks. You also might want some way to log and handle unhandled exceptions and to forward

the end user to an error page that has the same look and feel of your site collection rather than the silly error page that comes with SharePoint. In fact, nothing is worse than an error being thrown at the face of the end user and not being able to figure out what caused the problem because the exception is unhandled and it is not revealed anywhere in the log files.

A great way to do centralized logging and exception handling and provide a friendly error page is to employ a global exception handler at the application level. This will catch any unhandled exceptions, log them, clear them, and redirect the user to a fully branded page. You can also extend your global exception handler to send notification mails to the site collection administrator informing them that an unhandled exception has occurred. Implementing a global exception handler in SharePoint is a bit different from regular ASP.NET applications. For more information, refer to the following blog post: http://www.sharepoint4arabs.com/AymanElHattab/Lists/Posts/Post.aspx?ID=62

SharePoint Troubleshooting Toolbox

Since the release of WSS 3.0 and MOSS 2007, I have used a mixture of tools on the Web, which were of major benefit when debugging my code. My current SharePoint troubleshooting toolbox comprises the following tools.

Debug Config Feature

http://www.codeplex.com/features

http://www.u2u.info/Blogs/Patrick/Lists/Posts/Post.aspx?ID=1766

Debug Config is a handy SharePoint feature that, when activated, automatically tweaks your web configuration file and switches it to development mode by turning on StackTrace and turning off customErrors using some code within the feature receiver. This code utilizes the SPWebConfigModification class, which can be used to dynamically modify SharePoint web configuration files.

ULS Logs Viewers

http://hristopavlov.wordpress.com/sptraceview/

http://www.codeplex.com/SPLogViewer

http://codeplex.com/wssmosslogfilereader

When you choose to increase the number of events captured by the ULS, you will find the log files very cluttered and hard to use, and SharePoint provides no built-in viewer. To fill the gap and smooth the progress of troubleshooting, you can use third-party tools. For me, I use the following free ones:

- SPTraceView
- LogViewer
- WSS/MOSS Log File Reader

Debugger Feature for SharePoint

http://blogs.msdn.com/sharepoint/archive/2007/04/10/debugger-feature-for-sharepoint.aspx

A nifty feature by a brilliant developer, Jonathan Dibble, the Debugger Feature for SharePoint can be installed and activated on a SharePoint site to automate the "attaching to debugger" process. Once activated, the debugger feature adds an Attach Debugger menu item to the Site Actions menu, as shown in Figure 20.

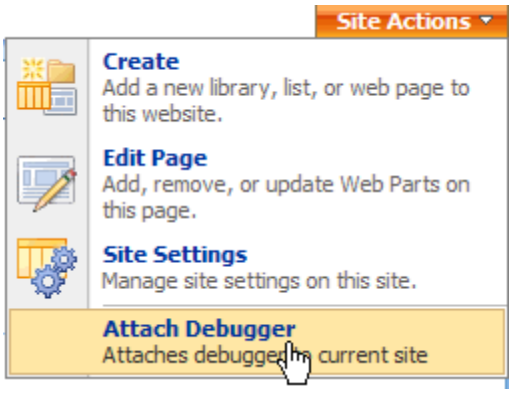

FIGURE 20. DEBUGGER FEATURE FOR SHAREPOINT

The feature provisions a simple page, which executes the System.Diagnostics.Debugger.Launch statement, causing an exception to be thrown and the debugger to be auto-attached. In some cases, the debugger cannot attach, such as when the application pool is running under a different user account. When that happens, the page will at least give you the correct PID, so you can then attach the debugger yourself. It's extremely helpful and faster than doing the same thing from Visual Studio.

WSPBuilder Extensions for Visual Studio

http://www.codeplex.com/wspbuilder

One of my favorite and commonly used features of WSPBuilder Extensions for Visual Studio, besides generating the solution packages, is the Attach to IIS Worker Processes shortcut. By clicking the project, choosing WSPBuilder, and then clicking Attach to IIS Worker Processes, you can effortlessly attach a debugger to all the available worker processes.

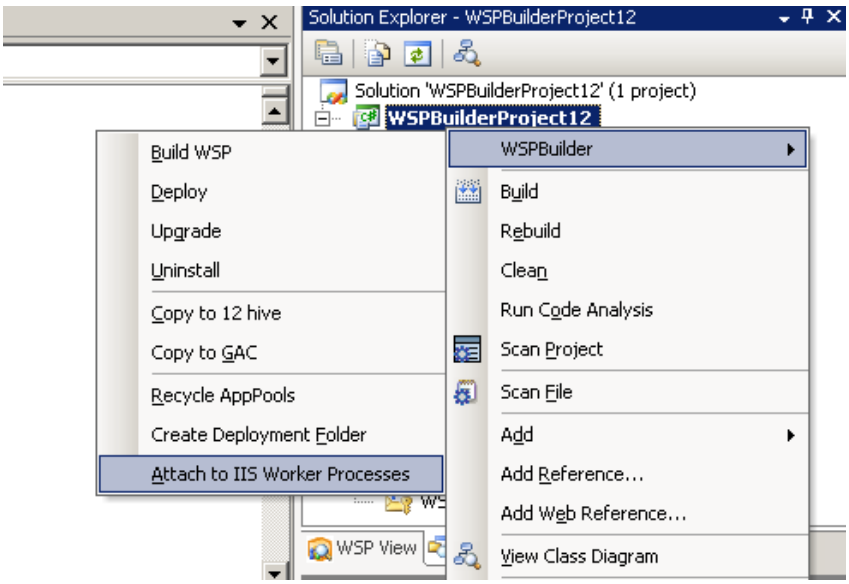

FIGURE 21. USING WSPBUILDER EXTENSIONS

Assembly Binding Log Viewer

Have you ever deployed your pages to SharePoint and then had some binary files refuse to load and throw the following exception?

FIGURE 22. FILE NOT FOUND EXCEPTION NOT SPECIFYING THE FAILING ASSEMBLY

File Not Found. at System.Reflection.Assembly._nLoad(AssemblyName fileName, String codeBase, Evidence assemblySecurity, Assembly locationHint, StackCrawlMark& stackMark, Boolean throwOnFileNotFound, Boolean forIntrospection)

Sometimes, it's effortless to resolve this error by simply adding <%@Register> directives to reference some assemblies in your page markup; other times you will not be able to figure out what went wrong. Solving the problem can be tiresome because the exception message never specifies the assembly that is failing to load. Luckily, the Microsoft .NET Framework SDK comes with a very handy tool named Assembly Binding Log Viewer that can assist you diagnosing this issue.

The Assembly Binding Log Viewer keeps track of all assembly binding attempts that happen on your system. Although this tool ships with the .NET Framework, it's not very commonly known. The following blog post will help you get started with the Assembly Binding Log Viewer and understand how you can use it to resolve this kind of exceptions. http://www.sharepoint4arabs.com/AymanElHattab/Lists/Posts/Post.aspx?ID=63.

SharePoint Logging Library

http://sploggginglibrary.codeplex.com

The SharePoint Logging Library is an open source library available at CodePlex. It helps you logging debug information, warnings, or exceptions into SharePoint trace logs. The best part about SharePoint Logging Library is that it automatically detects the namespace, class, and method names and writes them hand in hand with the messages you specify in your code, while you still have full control over the severity level and the logging category.

```
namespace WebApplication1
{
    public class MyWebPage : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            try
            {
                int x =30;
                SharePointLogger.LogStartOfMethod();
                SharePointLogger.LogWarning("Hello SharePointers");
                SharePointLogger.LogInformation("x = " + x);
                SharePointLogger.LogMessageWithSeverityAndCategory("Hi",
                SharePointLogger.TraceSeverity.High, "My Category");
                SharePointLogger.LogEndOfMethod();
            }
            catch (Exception ex)
            {
                SharePointLogger.LogException(ex);
            }
        }
    }
```

FIGURE 23. USING SHAREPOINT LOGGING LIBRARY IN SHAREPOINT CUSTOM CODE

Figure 24 shows what you get in the trace log files when this code executes.

```
07/23/2009 16:13:36.74    w3wp (0x1304)                0x13D0
    SharePoint Logging Library        General                0
    Verbose    Beginning of WebApplication1.MyWebPage.Page_Load method

07/23/2009 16:13:36.74    w3wp (0x1304)                0x13D0
    SharePoint Logging Library        General                0
    Warning    {WebApplication1.MyWebPage.Page_Load} : Hello
SharePointers

07/23/2009 16:13:36.74    w3wp (0x1304)                0x13D0
    SharePoint Logging Library        General                0
    Verbose    {WebApplication1.MyWebPage.Page_Load} : x = 30

07/23/2009 16:13:36.74    w3wp (0x1304)                0x13D0
    SharePoint Logging Library        My Category            0    High
    {WebApplication1.MyWebPage.Page_Load} : Hi

07/23/2009 16:13:36.75  w3wp (0x1304)              0x13D0 SharePoint Logging Library
    General              0      Verbose End of WebApplication1.MyWebPage.Page_Load
method
```

FIGURE 24. TRACE LOG FILES AFTER USING SHAREPOINT LOGGING LIBRARY

SharePoint Manager 2007

http://www.codeplex.com/spm

SharePoint Manager 2007 gives you a visual interface to the object model of SharePoint, allowing you to explore what properties are available, see how a site is structured, monitor timer jobs, and do a ton of other stuff.

With SPM, you can browse your entire SharePoint installation, together with servers, sites, services, lists, items, timer jobs, and anything else. It's really useful when it comes to SharePoint troubleshooting.

.NET Reflector

.NET Reflector is not a SharePoint tool per se, but it is still incredibly useful for developers. It allows you to see the source code of .NET code. That's right, campers; SharePoint is a .NET application, so you can see the source code of SharePoint as clear as the sky on a sunny day.

After you have started .NET Reflector, you need to load whatever assembly you want to inspect. A prime candidate is the Windows.SharePoint.dll file located in the [12]\ISAPI folder. Either open the assembly from the Open menu or drag and drop the assembly from a folder into the .NET Reflector window. Click the expand icon (the small plus sign) in front of the assembly name, and you can keep browsing down into the namespaces and classes inside the assembly. Double-click a property or method, and a second pane opens showing the source code of the method. You can also right-click an entire class and choose Disassemble to open an entire class for inspection.

Final Thoughts and Additional Resources

Thank you, Furuknap.

I really had fun joining the *USPJ* authoring team and writing this little bonus issue of the journal, and I really hope that you had fun reading it. In this issue, I introduced you to some of the troubleshooting utilities, tools, tips, and tricks that can make your life easier in view of the fact that SharePoint troubleshooting can really be a nightmare for those who are new to the platform if it is not performed properly.

Check out my blog at http://www.aymanelhattab.com, where you will find shorter articles, tips and tricks, questions and answers, and downloadable content. You can also follow me on Twitter http://www.twitter.com/aymanelhattab or drop me an e-mail at ayman.elhattab@gmail.com if you have any questions related to SharePoint or its related technologies.

To get information about the upcoming issues of *USP Journal*, make sure you sign up for the mailing list on the USP Journal website, http://www.understandingsharepoint.com/journal. Members of the list receive special offers, discounts, and previews of new issues.

So, until I talk to you again, good SharePoint troubleshooting to you!