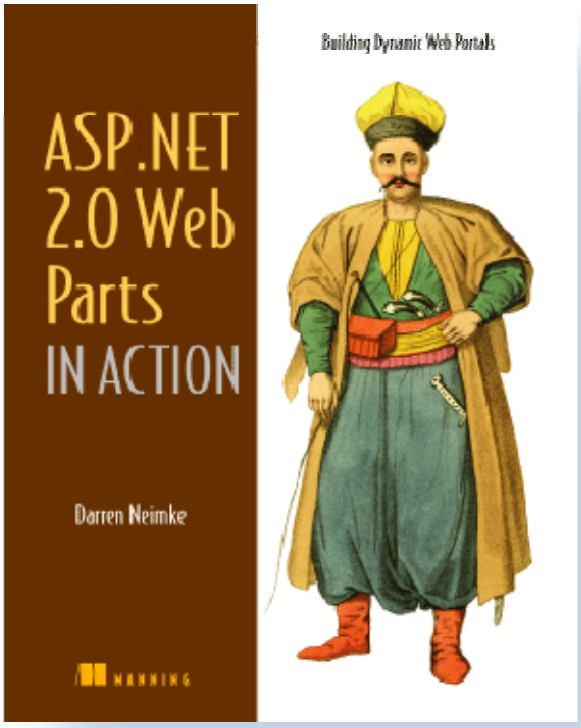
An Excerpt
ASP.NET 2.0 Web Parts in Action
By Darren Neimke



Excerpt of

ASP.NET 2.0 Web Parts in Action

Chapter 9: Portal Management

For more on this book click here
and check out Manning's other .NET and Microsoft titles here

CHAPTER 9

Portal management

9.1 Introduction 257
9.2 Preparing for deployment 258
9.3 Recovering from errors gracefully 268
9.4 When all else fails 271
9.5 Summary 281

9.1 INTRODUCTION

It seems like ages since we started our journey into the world of web parts and portals and we've covered a lot of ground along the way. Having taken on the challenge of creating a web portal for the Adventure Works business, we set about liaising with the end-users of the portal so that we could understand their requirements and ensure that we were building suitable features for them. Well, the good news is that our work is almost complete and soon it will be time to deploy the code onto the company's web server so the HR employees can begin using their new portal application. Before we can deploy our portal though, we need to start the planning that will help us to decide how the portal will be deployed. In addition, we need to work out how to support the portal after it has been deployed.

By the end of this chapter, we will not only have deployed our portal, but we will also have set a strategy for effective management of the portal when it is no longer under our control. Having this strategy in place frees us to be creative in chapter 10, when we look at the newer areas of portal development that are emerging.

When we build software applications, we always go through the well-known Software Development Lifecycle (SDLC) process. The SDLC defines the steps and processes that we must pass through to create quality software applications. This is

essentially a linear progression from planning stages though to development, finishing with the testing and deployment stages. Because of the linear nature of the SDLC, it is also commonly referred to as the "lifecycle" of application development.

While the majority of the tasks we've embarked upon so far have been associated with the development stage of the lifecycle, we must now turn our attention to the last two phases of the SDLC lifecycle—testing and deployment.

So what exactly will happen when our application leaves the development environment, and what can we do to ensure that we are able to manage and provide support for the portal when it leaves our hands? This chapter answers those questions.

9.2 PREPARING FOR DEPLOYMENT

Picture this situation: we've finished developing our portal application, so we deploy the application files onto the company's servers and release it to the users in the HR department. For the first week everything goes according to plan and, aside from a few requests for enhanced functionality, there have been no major hiccups and the application is running smoothly. However, in the second week, we start getting phone calls from the users complaining that, at times, our application seems to run slowly and sometimes stops working altogether.

This is the worrisome scenario we face every time we deploy our applications into a production environment. How will we diagnose our application to track down an obscure and hard to locate bug? You might think we could simply run the code in our development environment and observe the bug there by using debugging techniques, but remember that this particular bug took a week to begin showing its beady eyes. In reality there's no guarantee that the bugs that affect our applications in one environment will be reproducible in another environment, because many factors often differ between environments. For example, we typically develop our applications on a machine running a desktop operating system such as Windows XP; but when deployed, these same applications run on a machine running a server operating system such as Windows 2003 Server. In reality, there are hundreds of factors that vary between our development environment and the environments that we deploy our code in.

Of course when our portal is in the development phase of the SDLC lifecycle, it is easy to diagnose the cause of errors occurring in the application, because the code is running on our development machine. In addition, we can use the debugging tools in Visual Studio 2005 to connect to the running application and step through the execution of a page to locate errors and attain the information necessary to help us track down the cause of problems. We saw how to attach the Visual Studio debugger to a web page and view the state of variables in chapter 2. Recall that we used the debugger when we attached it to a web page to look at the state of a `GenericWebPart` at runtime. However, as mentioned, things are not so simple when our applications are beyond reach.

Let's now take a step into the world of application monitoring where we can find out what tools are at our disposal for diagnosing and tracking down errors when our applications are deployed. On this trek, we'll see two things. First, we use code instrumentation to diagnose existing problems. And second, there are ways to monitor the health of the applications so that we might even detect errors before they actually occur.

9.2.1 Code instrumentation

To assist in the task of tracking down bugs in applications, we can add code that will log information about our application when it is running. This practice of adding logging information is known as *instrumenting* our code. We'll now take a look at the `Trace` class in the .NET Framework which can be used to display diagnostic information about the state of our code at runtime. We'll also see how ASP.NET provides built-in support for optionally displaying this diagnostic information at the bottom of each page. Displaying tracing information within a page provides a simple, interactive debugging experience when we are looking for problems in our code at runtime.

The whole point of instrumenting code is to get help diagnosing problems when they arise in our applications. When we add instrumentation it is for the purpose of understanding what's happening in our code in a quantitative way, without having to run it on our own machine. By instrumenting our code and logging the results to an output file or database, we can then analyze the data to diagnose the cause of problems and then work out how to take corrective action.

ASP.NET provides us with the `Trace` class for instrumenting our code using trace statements. `Trace` class also allows the output of those statements to be written to a specific target location such as a log file, the Windows Event Log, or even to simply be displayed at the bottom of the screen. In the following snippet of code, trace statements are used to send messages to the default tracing output location to record when code enters and leaves the `AddHyperlink` method in our `FavoritesWebPart`:

```
public void AddHyperlink(HyperlinkData hyperlink) {

    HttpContext ctx = HttpContext.Current;
    ctx.Trace.Write("Entering AddHyperlink: " + hyperlink.URL);

    ... method code here

    ctx.Trace.Write("Exiting AddHyperlink");
}
```

The `Trace` can be accessed directly from code within our page, but for code contained within controls such as our `FavoritesWebPart` we must first get a handle to an instance of the current page context and access the `Trace` instance directly from the context object. The `Trace` class writes the output of our trace statements into

objects known as *listeners*, which take the output and write it to a specific output device. Once we have tracing statements in our code, we can configure our pages to have these statements appear at the bottom of the page at runtime. This is useful for accessing important debug information quickly and is accomplished simply by setting the trace directive of the page to true as shown in the following snippet:

```
<%@ Page Trace="true" %>
```

Now when we run a page and add a hyperlink to our `FavoritesWebPart`, our tracing statements will appear at the bottom of the page as shown in figure 9.1.

In figure 9.1 we see that the tracing output is now being added to the bottom of our page, and that our tracing statements appear within the Trace Information section of that output. By reading this information we can confirm that code entered and exited the `AddHyperlink` correctly, and we can also see the text of the hyperlink that is being added. The From First and From Last columns, appearing in the tracing output can be used to determine how long it takes each section of code within the page's lifecycle to execute. The From First column tells us the time in seconds since the first trace message was output, and the From Last column tells us how many seconds have elapsed since the last trace message was output. By reading the output we can locate code that is slow running and then use our own development environment to look for ways to improve the performance of that particular piece of code.

Another useful member of the `Trace` class is the `Warn` method, which is similar to the `Write` method except that its output is displayed in the Trace Information section in red text. It is customary to use the `Warn` method to write tracing output for extreme conditions that our code encounters such as displaying the text of exceptions that are being handled.

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.000414298465307742	0.000414
aspx.page	Begin Init	0.000487771490510665	0.000073
aspx.page	End Init	0.00861338522074733	0.008126
aspx.page	Begin InitComplete	0.00871619158300846	0.000103
aspx.page	End InitComplete	0.0208641042367116	0.012148
aspx.page	Begin LoadState	0.0209669105989728	0.000103
aspx.page	End LoadState	0.0243947205580598	0.003428
aspx.page	Begin ProcessPostData	0.0244712665995259	0.000077
aspx.page	End ProcessPostData	0.0301032673147006	0.005632
aspx.page	Begin PreLoad	0.0301848419282339	0.000082
aspx.page	End PreLoad	0.0302971467043996	0.000112
aspx.page	Begin Load	0.0303591657598941	0.000062
aspx.page	End Load	0.0305100229219077	0.000151
aspx.page	Begin ProcessPostData Second Try	0.0305790261052732	0.000069
aspx.page	End ProcessPostData Second Try	0.0306449562723754	0.000066
aspx.page	Begin Raise ChangedEvents	0.0307030642162621	0.000058
aspx.page	End Raise ChangedEvents	0.0307815658135322	0.000079
aspx.page	Begin Raise PostBackEvent	0.0308407912178783	0.000059
	Entering AddHyperlink: http://Microsoft.com	0.0309656674242117	0.000125
	Exiting AddHyperlink	0.0311251849047854	0.000160

Figure 9.1 With page tracing turned on, tracing statements appear in the trace information section at the bottom of the web page.

For example, in our data access layer code we typically handle exceptions using try…catch code blocks, such as the one shown in the following snippet of code:

```
try {
    ... attempt an operation here
} catch( Exception ex ) {
    ... catch any exceptions here
} finally {
    ... perform clean-up tasks here
}
```

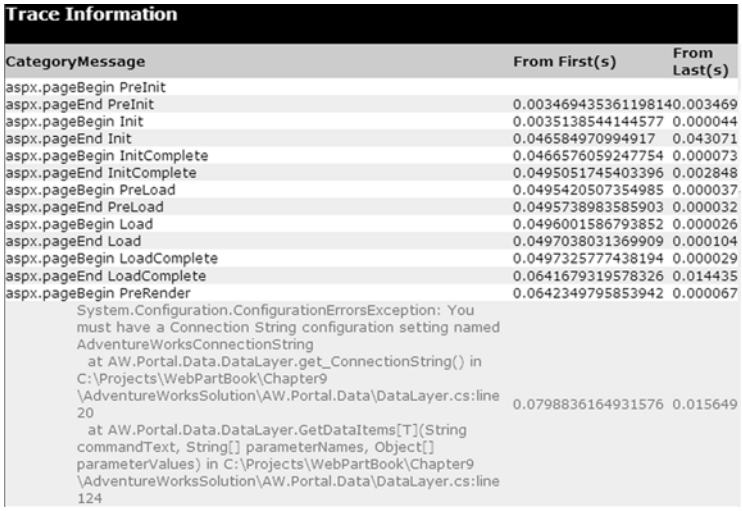
Using a try…catch…finally block ensures that any exceptions that might occur are trapped, and therefore do not cause the execution of our page to fail. Catching exception information also helps us to ensure that sensitive information such as a connection string is not displayed on the page to our users. For example, if there were a problem with one of the stored procedures that we wrote for our portal, we wouldn't want users to see that information displayed on the web page, but we would need to write it out in a tracing statement so we could know to fix the problem. To do this we can add a `Trace.Warn` call to the catch block to display details about the exception as shown in listing 9.1:

Listing 9.1 Using Trace.Warn will cause statements to appear in red text within the tracing output, and is the standard to use when instrumenting exceptions.

```
try {

    ... method code here

} catch (Exception ex) {

    HttpContext ctx = HttpContext.Current;
    ctx.Trace.Warn(ex.ToString());

}finally {
    ... perform clean-up tasks here
}
```

Now when an exception is thrown we will be able to view a detailed message about the cause of the exception in our tracing output. For example, we might forget to correctly configure our `AdventureWorks` connection string in the configuration file. If so, when we deploy the application, our web parts will appear on the pages, but they will not display any content. When we view the `Trace` output for the page, the exact nature of the problem will be revealed to us as shown in figure 9.2.

Now when we run the page with tracing turned on, we can clearly see that the reason our web parts are not displaying any content is because we haven't added a connection string configuration setting named `AdventureWorksConnectionString` to our web configuration file.



Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.003469435361198140	0.003469
aspx.page	Begin Init	0.0035138544144577	0.000044
aspx.page	End Init	0.046584970994917	0.043071
aspx.page	Begin InitComplete	0.0466576059247754	0.000073
aspx.page	End InitComplete	0.0495051745403396	0.002848
aspx.page	Begin PreLoad	0.0495420507354985	0.000037
aspx.page	End PreLoad	0.0495738983585903	0.000032
aspx.page	Begin Load	0.0496001586793852	0.000026
aspx.page	End Load	0.0497038031369909	0.000104
aspx.page	Begin LoadComplete	0.0497325777438194	0.000029
aspx.page	End LoadComplete	0.0641679319578326	0.014435
aspx.page	Begin PreRender	0.0642349795853942	0.000067
	System.Configuration.ConfigurationErrorsException: You must have a Connection String configuration setting named AdventureWorksConnectionString at AW.Portal.Data.DataLayer.get_ConnectionString() in C:\Projects\WebPartBook\Chapter9\AdventureWorksSolution\AW.Portal.Data\DataLayer.cs:line 20 at AW.Portal.Data.DataLayer.GetDataItems[T](String commandText, String[] parameterNames, Object[] parameterValues) in C:\Projects\WebPartBook\Chapter9\AdventureWorksSolution\AW.Portal.Data\DataLayer.cs:line 124	0.0798836164931576	0.015649

Figure 9.2 This trace output contains the text for two configuration exceptions, which is the result of tracing code contained within the core ASP.NET code.

Seeing tracing in action here makes it clear how useful it is as a tool for detecting the source of problems in our applications. As for prescriptive guidance about where to place tracing statements within an application's code, the obvious place is to find the strategic locations most likely to cause problems at runtime, and target them so we can work out how to diagnose those errors when they occur. At a minimum, I would suggest instrumenting the following code:

- Exception handling blocks
- Calls to external systems which may be expensive, so that we can monitor how long those operations are taking
- Calls to complex business logic operations to help detect erroneous logic

There are many additional topics about managing tracing within our applications worthy of investigation. One such topic is configuring listeners so that the output of tracing statements is directed to areas other than the bottom of the page—such as the Event Log or into a database. As such I highly recommend reading up on ASP.NET Tracing to learn more about options that exist to help with diagnosing problems in web applications.

9.2.2 Health monitoring

Tracing is certainly useful when diagnosing issues that already exist within our application, but ideally we'd like to be a little more pro-active. We'd like to monitor the application in order to detect certain types of issues before they actually become problems. This activity is known as Health Monitoring and involves keeping track of

performance counters in code, and viewing these vital statistics periodically to keep an eye peeled for signs that errors are occurring.

Typical use

ASP.NET 2.0 contains an event-based health monitoring system known simply as Health Monitoring that we can tap into to keep track of the health of our applications. We can use the Health Monitoring system to monitor the health of our applications and send notifications as thresholds when certain types of events are raised. For example, periodically the ASP.NET process will be forced to recycle, which will trigger an application restart. Most of the time, these restarts are expected. Here's an example: an administrator changes a web configuration file and thereby restarts the application. However, there are other times when application restarts are symptomatic of an application experiencing extreme difficulty. In this case, we want to receive notifications about the events. For instance, large numbers of application restarts are a typical symptom that a denial-of-service attack is taking place. Using the Health Monitoring service we could configure our application to listen for application restarts, and configure a rule that would cause a notification to be sent after a certain threshold is breached within a given time.

Monitoring system is highly configurable, and therefore allows us to choose which logging provider to use as the output of the notification alert. Table 9.1 displays a list of the standard logging providers configured to work with ASP.NET applications, and details where they target their output:

Table 9.1 Standard logging providers that are provided for the Health and Monitoring service in ASP.NET.

Provider Class	Implements an event provider that …
System.Web.Management.EventLogWebEvent-Provider	Logs ASP.NET health-monitoring events into the Windows Application Event Log
System.Web.Management.SimpleMailWeb-EventProvider	Sends e-mail for event notifications
System.Web.Management.SqlWebEventProvider	Saves event notifications to an SQL database
System.Web.Management.TraceWebEvent-Provider	Sends ASP.NET health-monitoring events as trace messages
System.Web.Management.WmiWebEvent-Provider	Maps ASP.NET health-monitoring events to Windows Management Instrumentation (WMI) events

In addition to the standard logging providers listed in table 9.1, we can also create our own custom providers. Even more, we can extend and customize the Health Monitoring system at a very granular level. In the next section we'll take a look at the two most common areas of extensibility with the Health Monitoring system: Custom Providers and Custom Events. While learning about these topics we'll also take the opportunity to learn how to configure the Health Monitoring system for our application.

Custom providers

The main reason for creating a custom logging provider is to handle situations when there are specific requirements for how notifications should be delivered. We've seen that, for common notification sinks such as the Windows Event Log, Email, or a SQL Server database, there are already pre-built providers available, but there will also be times when we need to target other types of notification consumers—such as a mobile phone device. When we are targeting a mobile device, we could create a logging provider that would receive notifications of critical behavior and have it send alerts to the cell phone of an application administrator to advise him that things are not quite right. When we need to target a notification consumer that is not supported by the standard logging providers listed in table 9.1, we need to write our own provider and add to it the logic for dispatching the notifications to the device we are targeting.

We create a custom logging provider by creating a class that derives from the `WebEventProvider` class and overriding the `ProcessMessage` method. We place our custom logic for dispatching the event notification in the `ProcessMessage` method. It is this method that will be called by ASP.NET whenever an event that is mapped to our provider is fired. This method receives an argument named `raisedEvent`, which contains information about the Health Monitoring event that occurred. This information includes the event type, event code, and a message describing the event. Once we have created a custom logging provider, we can configure it for use via the provider's element of the `healthMonitoring` section of the web configuration file, as seen in the following code snippet:

```
<healthMonitoring enabled="true">
    <providers>
        <add name="MySmsProvider"
type="SmsWebEventProvider,SmsWebEventProvider" />
    </providers>
</healthMonitoring>
```

In this snippet we are telling the Health Monitoring system that a custom event provider named `MySmsProvider` is available for use within the application.

Events

Notifications are dispatched to the logging providers through events. For example, when an ASP.NET application is restarted, an application restart event is raised by ASP.NET and handled by whatever logging provider is configured to handle events of that kind. If the `SqlWebEventProvider` was configured as the current provider for application restart events, an entry would be written into an SQL Server database each time the restart count reached a threshold that we had configured for the application.

The Health Monitoring system is composed of a large hierarchy of standard events that are raised by ASP.NET as it goes about the job of processing web requests. These events are broken down into the following categories: Request, Error, Audit, and Miscellaneous. In addition to the standard Health Monitoring events, we can

also create custom events to notify the Health Monitoring service of things that we want to keep an eye on in our applications. You configure which events should be handled within the application through the `eventMappings` element of the `healthMonitoring` section of the web configuration file, as seen in the following code snippet:

```
<healthMonitoring enabled="true">
    <eventMappings>
        <add name="WebServiceCallEvent"
          type="ExternalCallWebEvent, ExternalCallWebEvent" />
    </eventMappings>
</healthMonitoring>
```

In this snippet we are registering a custom event named `WebServiceCallEvent` with the Health Monitoring system. Note that the type of class configured for this event is a custom class named `ExternalCallWebEvent`. Listing 9.2 shows the code for the `ExternalCallWebEventClass`:

Listing 9.2 By creating and raising custom events, we can get fine-grained control over our health monitoring activities.

```
public class ExternalCallWebEvent : WebBaseEvent {

    public ExternalCallWebEvent (string message,
      object eventSource, int eventCode) :
        base(message, eventSource, eventCode) { }
}
```

The `WebBaseEvent` class that the `ExternalCallWebEvent` inherits from is the base class for all Health Monitoring events. We use this custom event by creating an instance of the `ExternalCallWebEvent` class whenever we detect certain conditions in our application and then calling the `Raise` method of the `WebBaseEvent` class. The following snippet shows how to raise an `ExternalCallWebEvent` event in code:

```
ExternalCallWebEvent e = new ExternalCallWebEvent(
    "Some notification message here",
    this,
    WebEventCodes.WebExtendedBase + 1
    );

e.Raise();
```

Judging from the name of the class, you can imagine that we might typically raise this particular custom event in response to an abnormal condition relating to making a call to another website. The example of Health Monitoring included in the Adventure Works sample for chapter 9 is based on a web part called `OPMLWebPart`. Figure 9.3 shows a picture of how the `OPMLWebPart` appears when displayed in a browser at runtime.

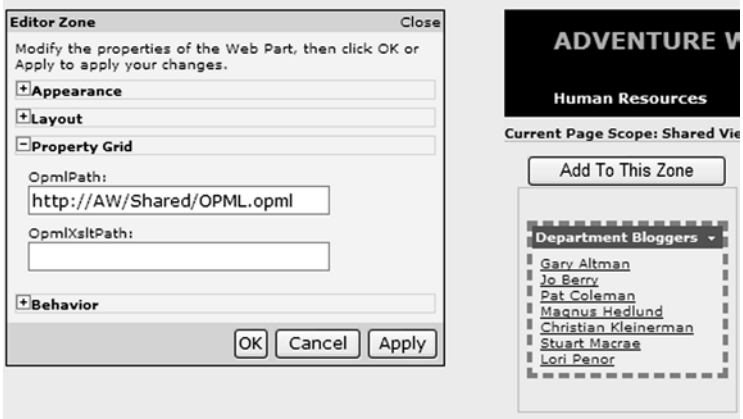


Figure 9.3 The OPMLWebPart reads an OPML file from an external website. Calls to external resources are ideal candidates for health monitoring.

The `OPMLWebPart` allows a user to configure a URL to third-party site and have an OPML formatted XML file returned from it. Making a call to another website is certainly something we'd want to keep our eye on. Imagine the case when the third-party site takes a long time to respond to our requests, or worse still, stops responding altogether. In such a case we'd want to be notified so that we could take corrective action before our users start reporting errors from our application. Within the code for the `OPMLWebPart`, health monitoring code records the time it takes to receive a response from the third-party site, and if it is longer than two seconds, the web part starts raising `ExternalCallWebEvent` health events. This code can be seen in listing 9.3.

Listing 9.3 By monitoring the time it takes to load the OPML file we can raise an alert when it starts taking excessive time to access the external resource.

```
XmlDocument tmp = new XmlDocument();
Stopwatch watch = new Stopwatch();

watch.Start();
tmp.Load(this.OpmlPath);
watch.Stop();

if (watch.ElapsedMilliseconds >= 2000) {
    ExternalCallWebEvent e = new ExternalCallWebEvent(
        string.Format(
            "The call to {0} took {1} milliseconds to complete.",
            this.OpmlPath,
            watch.ElapsedMilliseconds
        ),
        this,
```

Check time elapsed for making external call

```
            WebEventCodes.WebExtendedBase + 1
            );
                                    Raise custom health
                                    monitoring event
        e.Raise();   <-------------
    }
    return tmp.OuterXml;
```

In the code we create an instance of the `Stopwatch` class to keep track of how long it takes to load the XML from an external site. If that time exceeds 2000 milliseconds, we format a health monitoring event and raise it from within the system. The `Stopwatch` is a special diagnostic class that can be used to perform very accurate timings from within code, and is designed to be used especially for measuring elapsed times.

For our custom event to be logged, it must first be sent to a provider—which in our case will be the `MySmsProvider` configured earlier. To do this we configure the Health Monitoring service to handle our `WebServiceCallEvent`, and tell it which provider to use as the sink for those events. The following snippet shows how to connect our custom event to the SMS provider:

```
<healthMonitoring enabled="true">
    <rules>
        <add name="My SMS Rule"
            eventName="WebServiceCallEvent"
            provider="MySmsProvider"
            />
    </rules>
</healthMonitoring>
```

The rule we have configured here connects our custom `WebServiceCallEvent` to the `SMSProvider` so that whenever the `WebServiceCallEvent` is raised within our application, it will be sent to the SMS provider and our administrators will be alerted that something is wrong and the application requires their attention.

Health Monitoring is a substantial topic and this section has only scratched the surface. I highly recommend spending some time to research Health Monitoring and other techniques for keeping tabs on applications. Just remember that when an application is in a production environment and it starts to misbehave, you will be grateful that you took the time to instrument your code appropriately.

Of course, no matter how hard we try to detect errors ahead of time there will still be problems. We need to have a way to manage those sticky situations, so that the user is not stopped in his tracks with no options for recovery. After all, who wants to take the managing director's midnight phone call when he's grumbling about a broken web page? In the next section we will look at various strategies for assisting users when all is not going according to plan and errors arise.

9.3 RECOVERING FROM ERRORS GRACEFULLY

Earlier in this chapter we learned that we could instrument our code to help detect the root cause of problems, and saw how to use Health Monitoring as an early warning system to alert us when our application's health is waning. With all of these safety measures in place, we can now deploy our application, comfortable in the knowledge that we can detect errors before our users do, right? Well, unfortunately this is not the case, and in reality even the best applications succumb to errors at times. What we need is a way to handle errors when they occur at runtime and provide a way for our application to handle these errors gracefully, so that users are not left looking at an ASP.NET Error page such as the one shown in figure 9.4.

This is typical of what users see when something goes wrong with an ASP.NET page and an unhandled exception occurs. Displaying such a page to users is disconcerting

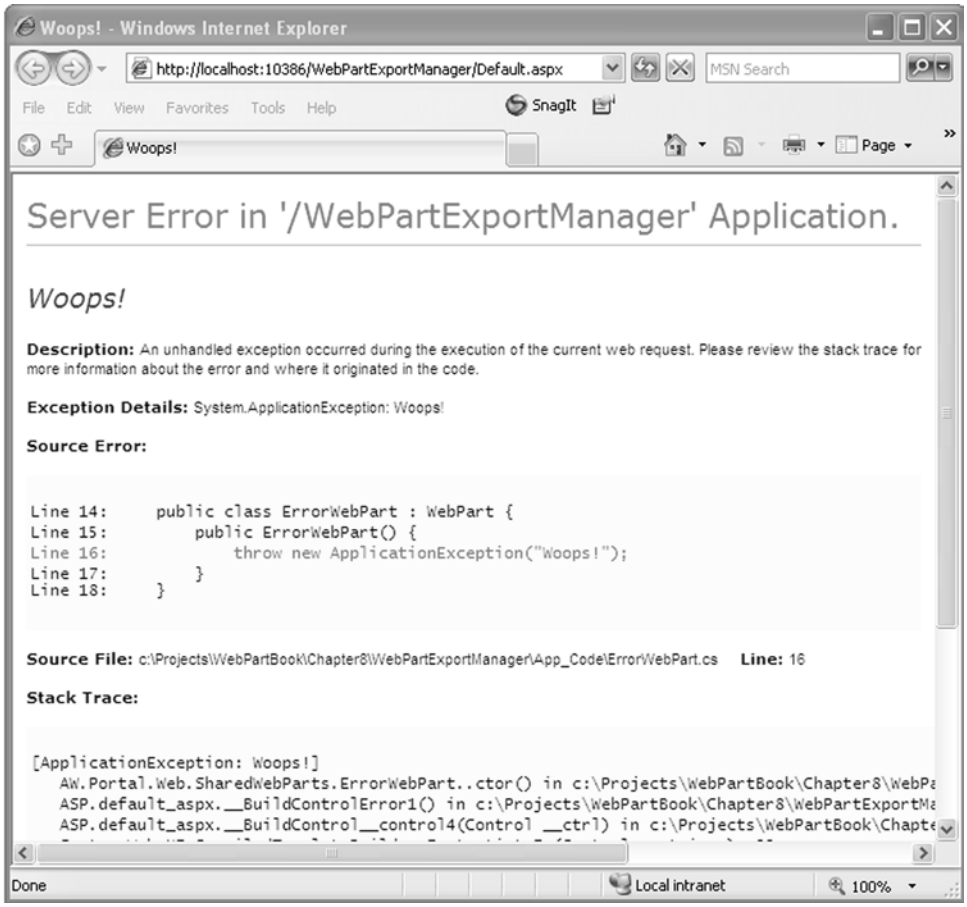


Figure 9.4 The dreaded ASP.NET error page will be displayed by default whenever an unhandled exception gets thrown.

because the page shows code, detailed error messages, and other technical information that users would simply not understand. Showing this level of technical information could be even worse if it were viewed by a visitor with mischievous intentions—such as a hacker. A hacker viewing technical details shown in figure 9.4 might gain valuable insight into database connection strings and other information that could then be used to hack the website. Having said all that, the ability to view such detailed information about errors is very useful when we are developing the application, as it helps us find errors and apply fixes faster. So we need to have the ability to flip a switch and have this page displayed in the development environment, but to display a custom error page when the application is deployed into other environments. Thankfully, ASP.NET provides us with this ability.

9.3.1 Providing a custom error page

To display a custom error page whenever unhandled errors occur, we can simply use the configuration settings of our application to tell ASP.NET which page to redirect to when errors occur. We do this by using the `customErrors` configuration element in the web configuration file. The following snippet shows an example of the `customErrors` element being set so that `ErrorPage.aspx` will be displayed whenever an unhandled error occurs within the application.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <customErrors mode="On"
        defaultRedirect="ErrorPage.aspx" />
  </system.web>
</configuration>
```

In this example, the mode attribute is set to On while the `defaultRedirect` attribute contains the URL of the page where we want to display custom error information. Typically, such a page would inform the users that an error has occurred and that the system administrator has already been notified. Finally, it is useful to provide users with a link from this page that allows them to navigate back into the site—such as a link to the home page or a link back to the page that the users have just come from. Listing 9.4 shows a sample custom error page that tells users what happened.

Listing 9.4 A standard page can be used to display to users in place of the dreaded error page. This page can contain useful information and provide users with a way to continue their browsing experience.

```
<html>
  <body>
    <p>
        We are sorry for the inconvenience but an error has occurred.  A
detailed error
        message has been sent to the system administrator.  Please
```

```
        click <a href="javascript: window.history.go(-1);">here</a> to
return to the page
        that you were just at, or click <a href="Default.aspx">here</a> to
go our Home Page.
    </p>
  </body>
</html>
```

When critical errors occur on our website, users will no longer be subjected to the confusing and highly technical standard ASP.NET error page. Instead, users are shown a page with which they are more familiar, and which helps them to recover and continue using our application. In the case of the page shown in listing 9.4, we have explained that an error has occurred and we are providing users with links which will allow continued use of the site. However, even though we've provided users with a more friendly error page, we still need a way to log the failure and notify an administrator that it has occurred so that we take steps to fix the problem behind the failure.

9.3.2 Logging the failure

The simplest place to put notification code is in the Global Application class. To add a Global Application class to our application, we simply right-click on the Visual Studio solution tree from within our web application and use the Add New Item menu option to add a new Global Application file to our application. This adds a file named `Global.asax` to our project. In the Global Application class, we can write code that will run when certain application level events occur within the application. For example, whenever a new page is requested within the application, the `BeginRequest` event is fired and we can write code in the Global Application class to handle that event. Likewise, the Global Application class allows us to handle the Application's Error event, so we can handle this event and write code that runs when an unhandled error occurs. Listing 9.5 is an example of how to handle the Application's Error event from within the `Global.asax` file and use it to send an email to an administrator notifying them of the error.

Listing 9.5 By logging unhandled errors from within the Applicaion_Error event handler we can get notifications when unexpected failures occur.

```
void Application_Error(object sender, EventArgs e) {

    Exception ex = Server.GetLastError();

    if (ex != null) {
        string body =
          "The following error has occurred: " + ex.ToString() ;
        string subject = "Application Error.";
        string from = "ErrorHandler@AdventureWorks.com";
        string to = "Adminstrator@AdventureWorks.com";
```

```
            SmtpClient mailClient = new SmtpClient("http://localhost");
            mailClient.Send(from, to, subject, body);
        }
}
```

The code in listing 9.5 uses the `GetLastError` method of the Server object to access an instance of the actual error that occurred. Having the underlying exception object at our disposal provides access to a great deal of information about the error, such as the message of the exception and also a full stack trace of what was happening at the time the error occurred. We can use this information to diagnose what might have caused it.

NOTE There's an excellent article on MSDN which discusses an extensible strategy for logging and reporting on errors that can be found at the following URL: http://msdn.microsoft.com/asp.net/default.aspx?pull=/library/en-us/dnaspp/html/elmah.asp

Now that our application has logging code, Health Monitoring code, and a custom error page, you might think that the application will be resilient in all kinds of disasters and that we have little to worry about. But if you're taking off your shoes to put your feet up on the desk, take heed of the following scenario.

9.4 WHEN ALL ELSE FAILS

A major selling point of web portals is that they facilitate the creation of modular user interfaces where we can deploy new web parts any time without having to re-publish an entire website. We saw this in chapter 8 with the custom dialog catalog, which allows us to add new web parts to our portal by simply dropping an assembly in the bin folder and then adding an entry to an XML file. Along with this ease of deployment comes the danger of deploying web parts that are not fully tested and contain errors. Think about what it would mean to deploy a web part which included a critical error and which threw an exception the instant it was added to the page. Once users added this web part to their pages, they would no longer be able to visit that page because the error page would be displayed instead. And because the users could not access their pages, they could not remove the web part. What to do?

We've seen in chapters 5 and 6 that the `WebPartPersonalization` class has a method which allows us to reset the entire set of personalization data for a user for a single page by using its `ResetPersonalizationState` method like so:

```
wpm.Personalization.ResetPersonalizationState()
```

So when a user has broken his page, we can fix it by finding a way to load the broken pages into the current context and then calling `ResetPersonalizationState` on the page. There are two issues with this. First, how can we load the page into the current context given that it won't load? Second, users will not be happy losing all their

settings for a page that is heavily personalized, so ideally we need to provide a way that allows page resets to occur at a more granular level. We'll now investigate how to provide the users a self-management facility that allows them to go in and fix their own broken pages when this type of scenario occurs.

9.4.1 Self-maintenance of web parts

In a presentation at the PDC conference in Los Angles in late 2005, Mike Harder, who is a Software Design Engineer on the ASP.NET team for the web parts feature, showed us how to provide user self-management. We'll now implement Mike's solution into our Adventure Works portal. First, let's summarize the steps a user performs so that we can see where we are headed.

The user

- Adds a broken web part to his page
- Is instantly taken to a self-help administration page to fix the problem
- Removes the broken web part from the page
- Is returned to the original page and the error web part is no longer present

To implement this solution we will create an administration page that allows users to fix pages themselves. The page we'll create will display a list of web parts for a given page, and will allow the user to delete one or more web parts from that page. Figure 9.5

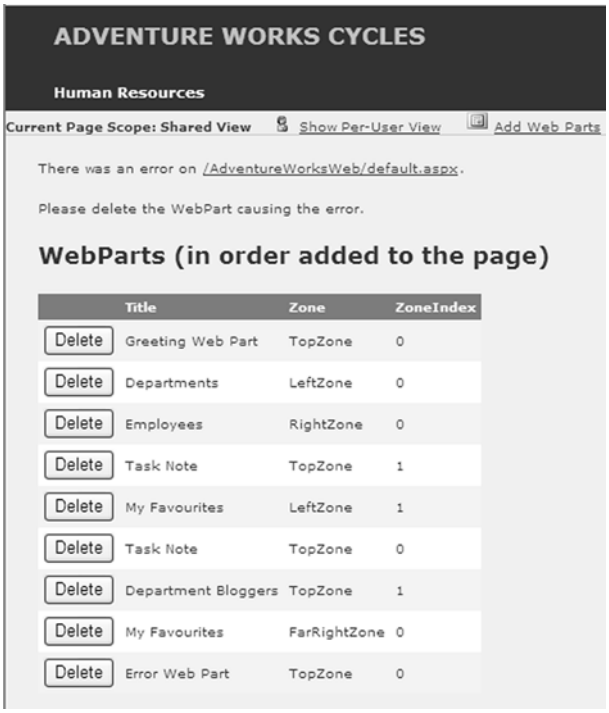


Figure 9.5
This grid displays all web parts for a given page and provides the user with a way to remove web parts which have caused the underlying page to stop working.

shows the administration page that users will use to remove troublesome web parts from their pages.

To get things started, add a new page to our portal named `WebPartAdministration.aspx` and set it as the custom error page for the application by making the following `customErrors` entry in the web configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <customErrors mode="On"
        defaultRedirect="WebPartAdministration.aspx" />
  </system.web>
</configuration>
```

As we see in figure 9.5, our administration page shows all web parts for a given page and allows them to be deleted from that page. In order to execute personalization operations on another page instance, we will be creating a helper class which can silently and invisibly instantiate a page in the background. Once we have that instance, we can access the web part manager on the instance to perform personalization operations. This helper class will also expose properties that allow us to view the web parts on the invisible instance. Figure 9.6 illustrates the relationship between the `WebPartAdministration.aspx` page and the helper class.

The helper class we create will be called `WebPartsAdministrator`, and as we see from figure 9.6 it can create an invisible instance of the web page that caused the user to arrive at the error handling page. The `WebPartsAdministrator` helper class also provides a method named `GetWebParts` that we can access from the administration page and use to return a listing of the web parts contained on the page with the errors. The web parts returned by the `GetWebParts` method will be bound to a list and displayed to the user—as shown in figure 9.5.

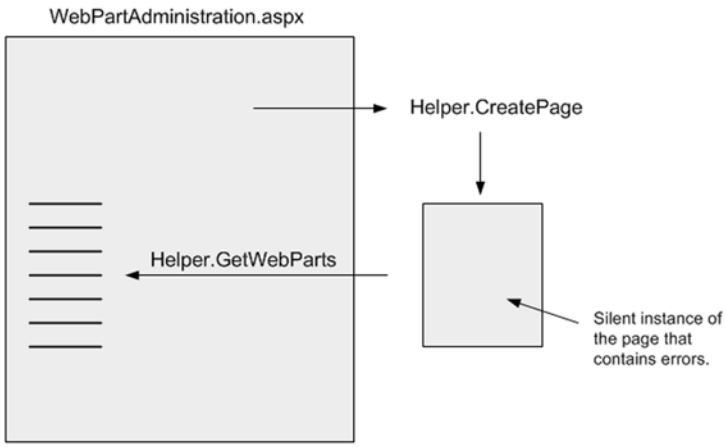


Figure 9.6 The relationship between our error handling page and the helper class used to communicate with the page that has errors.

Silently running a web page

The most technically challenging part of this solution lies in creating the logic within the helper class to silently instantiate the underlying page that contains the errors. Luckily the ASP.NET framework provides us with many useful methods and classes, making the task quite simple. The first bit of help from ASP.NET arrives when we call the `GetWebParts` method of the `WebPartsAdministrator` class. When `GetWebParts` method is called, our helper class creates an instance of a page based on a virtual path we provide and returns the web parts for that page; this can be seen in listing 9.6.

Listing 9.6 The GetWebParts method executes a target page and uses a callback to read web parts off of its web part manager instance.

```
public static WebPartCollection GetWebParts(
        string path, HttpContext context) {

    Page page =
     (Page)BuildManager.CreateInstanceFromVirtualPath(
       path,
       typeof(Page)
       );

    WebPartCollection webParts = null;
    page.PreLoad += delegate {
        webParts = WebPartManager.
            GetCurrentWebPartManager(page).WebParts;
    };

    ExecutePage(page, path, context);

    return webParts;
}
```

Listing 9.6 shows that the `GetWebParts` method dynamically instantiates the underlying page instance by using the `CreateInstanceFromVirtualPath` method of the `BuildManager` class. This method takes the path of the ASP page that we wish to create an instance of, and also a type argument, which indicates the base type for the page class. In our case we are passing the `aspxerrorpath` that we are passed by the ASP.NET error redirection, and simply specifying `Page` as the base class.

Once we have the page instance returned from the `BuildManager`, we hook the `PreLoad` event for the page. Doing this allows us to access the web part manager instance attached to the dynamic page during its execution so that we can access the web parts and personalization services from it.

Now that we have an event handler registered for the `PreLoad` event, we execute the page and our handler will be called at the appropriate part of the lifecycle of the dynamic page. In addition, our code will run just as it would when we write event handlers in the code behind normal pages. The code that executes the page is contained within a private helper named `ExecutePage`, which is shown in listing 9.7.

Listing 9.7 Executing the page is achieved via a call to Server.Execute which executes another page within the context of the current page.

```
private static void ExecutePage(Page page, string path,
                                HttpContext context) {
    string originalPath = context.Request.Path;
    context.RewritePath(path);

    try {
        context.Server.Execute(page, TextWriter.Null, false);
    } catch {}

    context.RewritePath(originalPath);
}
```

Notice—the first thing that we do in the `ExecutePage` method is to rewrite the path of the request by using the `Context.RewritePath` method. This ensures that code within the dynamic page will see the request URL as if the page really was the page requested by the user. This helps to ensure that the dynamic page runs exactly as it would for a normal request. After rewriting the path we use `Server.Execute` to execute the dynamic page, which processes the target page within the context of the current page. Notice in our code we are passing in a null `TextWriter`, so that even though the dynamic page is executed, it doesn't display anywhere. We also trap and gobble up any exceptions so they do not affect our administration page.

At the time the page is executed, it will run through its lifecycle in just the same way it would if it were being called directly from a browser request. So our event handler will receive a notification when the page is in its pre-loading phase. At that time we can access all the personalization and web part data directly from the web part manager in our `PreLoad` event handler and return them to our administration page where they will be displayed.

The last thing we do in the `ExecutePage` method is to rewrite the path of the request back to the original path, so that the remainder of the requested page will process in the right context.

The code for deleting a page is very similar to the code we used when we listed web parts. The only difference is that in our `PreLoad` event handler we write code that deletes a selected web part. Listing 9.8 shows the code for the `DeleteWebPart` method of our `WebPartsAdministrator` class.

Listing 9.8 Deleting a web part is achieved in a manner similar to attaining a list of web parts except that different code is run in the PreLoad callback.

```
public static void DeleteWebPart(string path, string ID,
                                 HttpContext context) {
    Page page = (Page)BuildManager.CreateInstanceFromVirtualPath(
        path, typeof(Page));
```

```
    page.PreLoad += delegate {
        WebPartManager webPartManager =
            WebPartManager.GetCurrentWebPartManager(page);
        WebPart webPart = webPartManager.WebParts[ID];
        webPartManager.DeleteWebPart(webPart);
    };

    ExecutePage(page, path, context);
}
```

Notice that we again use an inline code block as our event handling code for the `PreLoad` event by using the delegate keyword in C# and assigning a chunk of code within curly braces. This form of attaching inline code blocks as event handlers is called anonymous methods and is an elegant way to attach event handling code without having to write separate methods for handling events. Writing our event handling code as an anonymous method also allows us to refer to the ID argument passed in to the `DeleteWebPart` method directly, which is something that we could not do if the event handling code was in a separate method.

Databinding in the administration page

Now that we've finished the `WebPartsAdministrator` helper class, we can get to work on the visual elements in the `WebPartAdministration.aspx` page we created earlier. This page will list the web parts for a given page and allow the user to delete individual web part items; listing 9.9 shows the mark-up code that we'll use to do this.

Listing 9.9 The methods for obtaining a web part list and deleting web parts are encapsulated within an ObjectDataSource which can then be bound directly to a GridView, providing the user with a way of working with the data.

```
<p>
    There was an error on <asp:HyperLink ID="TargetPage" runat="server" />.
</p>

<asp:ObjectDataSource EnableViewState="False"
  ID="WebPartsDataSource" runat="server"
  TypeName="AW.Portal.Web.WebPartsAdministrator"
  SelectMethod="GetWebParts"
  DeleteMethod="DeleteWebPart">
        <DeleteParameters>
            <asp:QueryStringParameter
                  DefaultValue="&quot;&quot;"
                  Name="path"
```

ObjectDataSource supports data binding with business object

Parameters map to method arguments

```
                QueryStringField="aspxerrorpath"
                Type="String" />
            <asp:ControlParameter
                ControlID="WebPartsGridView"
                Name="ID"
                PropertyName="SelectedValue"
                Type="String" />
            <portal:HttpContextParameter
                Name="context" />
        </DeleteParameters>

        <SelectParameters>
            <asp:QueryStringParameter
                DefaultValue="&amp;quot;&amp;quot;"
                Name="path"
                QueryStringField="aspxerrorpath"
                Type="String" />
            <portal:HttpContextParameter
                Name="context" />
        </SelectParameters>
</asp:ObjectDataSource>
```

Display web parts using a GridView control

```
<asp:GridView ID="WebPartsGridView" runat="server"
    AutoGenerateColumns="False"
    DataSourceID="WebPartsDataSource" DataKeyNames="ID">
    <Columns>
        <asp:CommandField ButtonType="Button" ShowDeleteButton="True" />
        <asp:BoundField DataField="Title"
          HeaderText="Title" SortExpression="Title" />
          <asp:TemplateField HeaderText="Zone">
            <ItemTemplate>
              <asp:Label ID="Label1"
              Text='<%# Eval("Zone.DisplayTitle") %>'
              runat="server" />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="ZoneIndex" HeaderText="ZoneIndex"
            SortExpression="ZoneIndex" />
    </Columns>
</asp:GridView>
```

Three controls on our administration page are being used to fetch data and present it to the user. The first of these controls is a hyperlink control with an ID of `TargetPage` that is going to display a link back to the page which caused the error. Users can click on this link when they have removed the errant web part to return to the previous page. This provides a seamless browsing experience for users as they can

fix their problem and return to their work without having to manually track an administration page or make a phone call to resolve their issue.

A `GridView` control is used to display a grid of all web parts for the target page to the user and is presented in such a way that the user can see the Name of the web part, the Zone that contains the web part, its index within that zone, and a delete button for each web part.

ObectDataSource and command parameter controls

The final control on the page is an `ObjectDataSource`. This control is bound to the custom `WebPartsAdministration` business class that we built, and is configured to use the `GetWebParts` method as its `SelectCommand` and the `DeleteWebPart` method as its `DeleteCommand`. Command parameters are used to tell these methods where to go for their argument data. The `ObjectDataSource` control is a specific type of the new `DataSource` control and is used to perform two-way data binding between data sources and data-bound controls. There are also specific `DataSource` controls for performing data binding to SQL Server data sources and XML data sources; however, binding directly to business objects is convenient as it allows us to ensure that certain business rules can be enforced when dealing with application data.

In the following code segment, notice that the `GetWebParts` method expects two arguments and the `DeleteWebPart` method expects three. Also note how these are all passed in through command parameters configured within the `ObjectDataSource`. In the same code segment, note the flexibility we have with command parameters for binding to many different targets. Specifically, using these command parameter objects, we can bind method arguments directly to Querystring values, Form values, Cookies, Session items, and even Controls on the web page. An example is where we bind the ID method argument of the `DeleteWebPart` method directly to the `SelectedValue` property of the `GridView` control.

In fact, to highlight just how flexible the command parameters are, you may notice that the `HttpContextParameter` is a custom parameter class because it has a control prefix of "portal:" instead of the standard "asp:" control prefix used for the in-built ASP.NET controls. The following fragment shows the entire code for the custom `HttpContextParameter` class:

```
public class HttpContextParameter : Parameter {
    protected override object Evaluate(
        HttpContext context, Control control) {
        return context;
    }
}
```

This class simply overrides the `Evaluate` method of the `Parameter class` and returns the `HttpContext` that it is passed as its output.

The only task remaining in our page is to set the `Text` and the `NavigateUrl` properties of the `TargetPage` hyperlink, so that users can navigate back to where they came from, as shown in listing 9.10.

Listing 9.10 Our custom error handling page will receive an argument contained within the querystring.This can be used to route users back to their original location when they have finished administering their web parts and have fixed their problem.

```
protected override void OnLoad(EventArgs e) {
    base.OnLoad(e);

    TargetPage.Text = Request.Params["aspxerrorpath"];
    TargetPage.NavigateUrl = Request.Params["aspxerrorpath"]; ;

}
```

In this code we simply read the value of the `aspxerrorpath` that is passed to our page by ASP.NET and set it as the redirect path.

You've just seen how to manage personalization data at the level of individual web parts. It's now time to take a quick look at the `PersonalizationAdministration` class to see how we can work with personalization data at a wider level.

9.4.2 Managing personalization data

The `PersonalizationAdministration` class contains static members that provide us with the ability to perform queries over personalization data. Using this class, we could run queries to determine which users have not used the system recently and then use that information to decide how to reset their personalization data. This kind of functionality may be useful for large and active sites that do not wish to retain too much personalization data for inactive members.

Here's an example of a query that is used to return all personalization data older than 200 days:

```
DateTime inactiveSince = DateTime.Now.AddDays(-200) ;

PersonalizationStateInfoCollection inactiveUserResult =
   PersonalizationAdministration.GetAllInactiveUserState(inactiveSince) ;
```

A site administrator could run such a query and then use the results to decide which users should have personalization information deleted. To assist with such a task the administrator could write another query which allowed her to view all personalization data for a specific user before she deleted the data for that user. Figure 9.7 shows us how these queries might look when presented as a couple of administration web parts.

In the first web part, the user can enter the username for a specific user and have a collection of all his personalization data returned. In the second web part, an inactivity period can be entered and all personalization data for individual users and paths older than that date would be displayed in the grid. Notice how the data in the results grid displays the user's name and the path of the personalization data, and it also displays the size of the personalization data for that personalization instance. An administrator

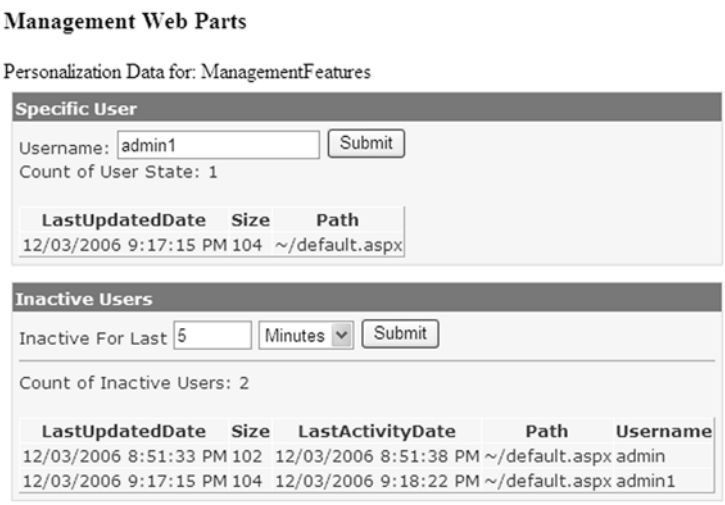


Figure 9.7 Using the PersonalizationAdministration class allows us to run queries over personalization data and provides a way to perform administrative queries such as checking on the amount of stale personalization data in the system.

might use web parts such as these to locate inactive users for a given path and then query all personalization data for that user to see whether she has some current personalization data instances before deleting that data.

The code for finding the personalization data for a specific user looks like this:

```
PersonalizationStateInfoCollection userResult =
  PersonalizationAdministration.FindUserState(
    null, UserNameTextBox.Text
    );
```

When the administrator has decided to delete the personalization data for a specific user, he can simply call the `ResetUserState` method of the `PersonalizationAdministration` class like so:

```
PersonalizationAdministration.ResetUserState(null, UserNameTextBox.Text);
```

The `PersonalizationStateInfoCollection` class that is returned from the `PersonalizationAdministration` queries is a collection of `PersonalizationStateInfo` instances. The `PersonalizationStateInfo` class is an abstract class, and so each of the items will actually be an instance of either the `UserPersonalizationStateInfo` class or the `SharedPersonalizationStateInfo` class. The `PersonalizationStateInfo` class itself contains only three properties useful to us: `LastUpdatedDate`, `Path`, and `Size`. When looping through a collection of personalization data from a query such as `GetAllInactiveUserState`, we must therefore cast each item to a `UserPersonalizationStateInfo` object before we can get at the Username of the user associated with the

personalization data. The following code snippet shows an example of how to get at the `UserPersonalizationStateInfo` specific properties from a specific `PersonalizationStateInfo` instance:

```
((UserPersonalizationStateInfo) userResult[0]).Username;
((UserPersonalizationStateInfo) userResult[0]).LastActivityDate;
```

It's these properties that are displayed to the administrator in the bottom grid in figure 9.7.

9.5 SUMMARY

At the beginning of this chapter we asked this question: how can we support and manage our web application when it is deployed and no longer directly under our control? By the end of the chapter, you have the answer. You can choose from a number of methods to keep an eye on the health of your code. Code instrumentation and health monitoring are two methods, and creating personalization queries and housing them within management web parts is another.

Often, instrumenting code and adding management features are unglamorous tasks. They are seldom found outside of commercial enterprise applications. For this reason, I wanted to take the time to discuss these practices here and show that with just a little foresight, a great deal of management capability can be injected into your applications—even if they are small and being written by a hobbyist!