

as databases, Web services, RSS feeds, email servers, file system, and other externalities that are beyond your control. A failure in any one of these systems means that your application can also no longer run successfully. It is vitally important that your applications can gracefully handle such problems. The rest of this section discusses and illustrates possible error handling approaches in ASP.NET and C#.

## .NET Exception Handling

When an error occurs, something called an *exception* is raised, or *thrown* in the nomenclature of .NET. That is, when an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error. When thrown, an exception can be handled by the application or by ASP.NET itself. In the .NET exception handling model, exceptions are represented as objects. The ancestor class for all exceptions is `Exception`. This class has several subclasses, such as `ApplicationException` and `SystemException`, which in turn have many subclasses such as `IOException`, `SecurityException`, and `NullReferenceException`. Every `Exception` object contains information about the error.

When an exception is raised but not handled by the application, ASP.NET displays the *default error page*. This page displays the exception message, the exception type, the line that it occurred on, as well as a stack trace, as shown in Figure 5.1. A *stack trace* displays every call, from the original page request down to the line that triggered the exception.

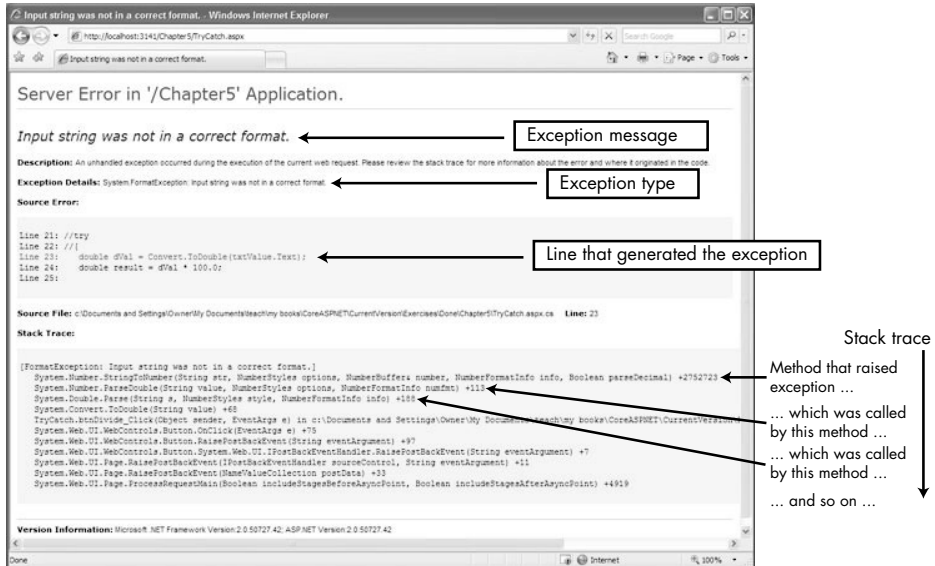


Figure 5.1 ASP.NET exception display

Although this ASP.NET default error page is quite useful when developing and debugging an application, you might not always want to display this page when an exception occurs. Instead, you might want to *handle* the exception. There are three different ways or levels where you can do so.

1. At the class level
2. At the page level
3. At the application level

## Exception Handling at the Class Level Using a try...catch Block

All .NET languages provide a mechanism for separating regular program code from exception handling code. In C#, this is accomplished via the try...catch block. If a runtime error occurs during the execution of any code placed within a try block, the program does not crash but instead tries to execute the code contained in one of the catch blocks. In the following example, there are two catch blocks. If an exception occurs, the system searches the associated catch blocks in the order they appear in the code until it locates a catch block that handles the exception. For this reason, the more specific catch exception must appear before the more general catch exception.

```
try
{
    double dVal1 = Convert.ToDouble(txtValue1.Text);
    double dVal2 = Convert.ToDouble(txtValue2.Text);
    double result = dVal1 / dVal2;

    labMessage.Text = txtValue1.Text + "/" + txtValue2.Text;
    labMessage.Text += "=" + result;
}
catch (FormatException ex1)
{
    labMessage.Text = "Please enter a valid number";
}
catch (Exception ex2)
{
    labMessage.Text = "Unable to compute a value with these values";
}
```

### CORE NOTE

---

If your Web application is going to be localized for different languages, you might want to place the actual error messages in local-specific resource files and instead reference the resource keys in the error messages. For more information, see the discussion on localization in Chapter 16.

---



There may be times when you want to execute some block of code regardless of whether an exception occurred. The classic example is closing a database connection no matter whether the SQL operation was successful or generated an exception. In such a case, you can use the optional `finally` block, as shown in the following partial example.

```
try
{
    // Open a database connection

    // Execute SQL statement
}
catch (DbException ex)
{
    // Handle database exception
}
finally
{
    // Close database connection if it exists
}
```

## The Cost of Exceptions

Throwing exceptions is relatively expensive in terms of CPU cycles and resource usage. As a result, one should try to use exceptions to handle only exceptional situations. If your code relies on throwing an exception as part of its normal flow, you should refactor the code to avoid exceptions, perhaps by using a return code or some other similar mechanism instead.

For instance, I was once hired to update an existing ASP.NET application written by a large programming team. In this application, there were pages that needed to interact with a method in a business object that handled customer requests. The method was passed the customer's email and then the business object's data members were populated from a database if the email existed in the database. The problem was that if the email didn't exist, the business object threw an exception. As a result, the code for using this method had to look similar to the following.

```
try
{
    SomeBusinessObject.Login(email);

    // Other code dependent upon a successful login
}
catch (Exception ex)
{
    // Display message that email was not found
}
```

Incorrect user input is not really an exceptional situation. In fact, it is so common that you should design your code to routinely handle it without raising an exception. The better approach would have been to refactor the `Login` method so that it returns some type of flag (such as a `bool` or a `null`) if the customer email does not exist, as shown in the following.

```
bool okay = SomeBusinessObject.Login(email);
if (! okay)
{
    // Display error message on page
}
else
{
    // Other code dependent upon a successful login
}
```

Similarly, the earlier `string-to-int` exception example can be rewritten so as to avoid the exception, by using the `Int32.TryParse` method, as shown in Listing 3.9 from Chapter 3.

## Possible Exception Handling Strategies

If you design your code so that exceptions are thrown only in truly exceptional situations, you might wonder what your program should do when one of these exceptional exceptions occurs. There are four possibilities.

- “Swallow” the exception by catching, and ignore the exception by continuing normal execution.
- Completely handle the exception within the `catch` block.
- Ignore the exception by not catching it (and thus let some other class handle it).
- Catch the exception and rethrow it for some other class to handle it.

The first approach is almost never appropriate. It is rare indeed to have a program that can blithely ignore a runtime error that would have crashed the program had it not been trapped by the `try` block. The second approach seems at first glance to make the most sense. If the class that generates the exception can gracefully and sensibly handle the exception, why shouldn't it? Remember that here we are not referring to routine or expected exceptions that are simply the by-product of a poor design, but are referring to truly unexpected exceptions.

If you remember the cost of exceptions, you (the developer) may want to know when an exception occurs in a production application so that you can change the code to prevent it from occurring in the future. In this case, you might not want to catch the exception but instead let some other class “higher” in the calling stack

handle it, perhaps by recording the exception to some type of exception log. Even if you are not recording an exception log, you should remember that *in general, you should not catch exceptions in a method unless it can handle them*, such as by logging exception details, performing some type of page redirection, retrying the operation, or performing some other sensible action.

Sometimes, developers catch an exception only to rethrow it so that it is still available to some other class “higher” in the calling stack, as shown in the following.

```
try
{
    // Other code that causes an exception
}
catch (Exception ex)
{
    // Do something with exception

    // Rethrow exception
    throw;
}
```

The cost of rethrowing an exception is quite close to the cost incurred raising it in the first place. Nonetheless, this approach may make sense if you want to decorate the exception message with some type of diagnostic information, as shown in the following.

```
catch (Exception ex)
{
    string myMessage = "Error in Class XXXX";

    // Throw new exception with your additional info
    throw new Exception(myMessage, ex);
}
```

This way, the exception can still get processed by some other “higher” class, but now it has additional information from your class about the exception.

## Exception Handling at the Page Level

ASP.NET allows the developer to handle errors on a page basis via the page’s `Page_Error` event handler. The `Page_Error` event handler is called whenever an uncaught exception occurs during the execution of the page. In the following example, the sample `Page_Error` event simply displays the error message and then clears the exception.

```
public partial class PageExceptionTest : System.Web.UI.Page
{
```

```
protected void Page_Load(object sender, EventArgs e)
{
    BuggyMethod();
}

private void BuggyMethod()
{
    // Deliberately throw an exception to simulate
    // uncaught exception
    throw new
        ApplicationException(
            "Your buggy code caused an exception.");
}

private void Page_Error(object sender, EventArgs e)
{
    Exception ex = Server.GetLastError();
    Response.Write("<h1>An error has occurred</h1>");
    Response.Write("<h2>" + ex.Message + "</h2>");
    Response.Write("<pre>" + ex.StackTrace + "</pre>");
    Context.ClearError();
}
}
```

The result in the browser is shown in Figure 5.2.

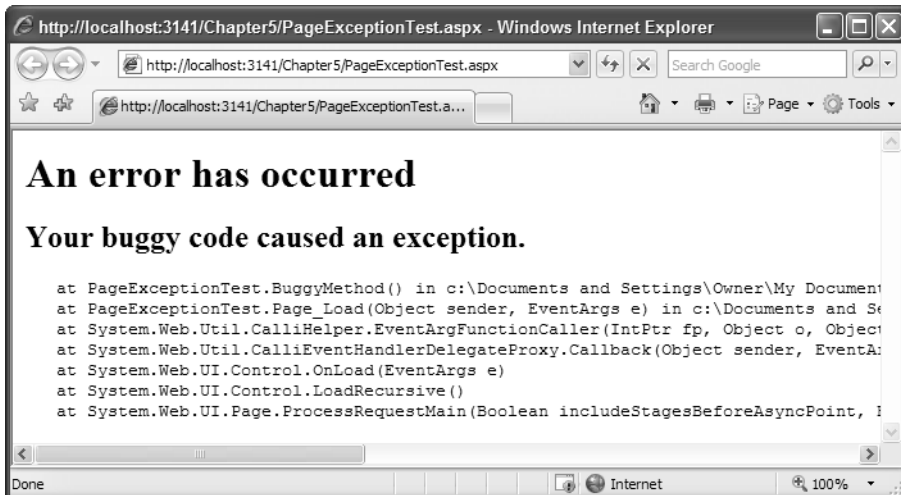


Figure 5.2 ASP.NET exception display

You might wonder why this example uses the `Response.Write` method instead of the usual ASP.NET practice of displaying dynamic text content in a control such as a `Label` control. The reason is that you cannot use controls in the `Page_Error` method because it is called *before* any control instances have been created. The `Context.ClearError` method is also necessary to prevent the default ASP.NET error page (shown in Figure 5.1) from displaying.

The `Page_Error` handler is typically used to handle exceptions not caught and handled by `try...catch` blocks. However, it is often preferable to *not* use the `Page_Error` handler, and instead use the application-wide `Application_Error` handler, which is covered next, for these situations.

## Exception Handling at the Application Level

There are two different ways that you can handle an exception at the application level: using a `Application_Error` event handler and using the ASP.NET error page redirection mechanism.

### Using the `Application_Error` Handler

ASP.NET allows the developer to handle errors on an application-wide basis via the `Application_Error` event handler. This handler resides in the application's `Global.asax` file and is often the preferred location to handle uncaught exceptions in an application. The reason why the `Application_Error` event handler is generally preferred over the `Page_Error` event handler is that you often want to do the same thing for all unhandled exceptions in your application: for instance, log them to some type of error log, display a custom message depending upon the role of the user, or send an email to the Web master. Rather than have the same type of error-logging code on every single page, it makes sense to centralize this code into a single spot. This single spot is the `Application_Error` handler.

The following example illustrates an `Application_Error` handler in the `global.asax` file that outputs any exceptions it receives to the Windows Event Log. This log can be viewed by the Event Viewer snap-in (see Figure 5.3 on page 266) that is available via the Administrative Tools option in the Windows Control Panel.

This example creates a new event source category named `WebErrors` if it doesn't already exist, and then outputs the event to this source. Notice as well that it uses the `System.Diagnostics` namespace.

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

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

<script runat="server">
```

```
...

void Application_Error(object sender, EventArgs e)
{
    // Construct the error string
    string msg = "Url " + Request.Path + "Error: " +
        Server.GetLastError().ToString();

    // Need to catch exception just in case you do not
    // have permission to access Event Log
    try
    {
        // Create the WebErrors event source if you need to
        string logName = "WebErrors";

        if (!EventLog.SourceExists(logName))
            EventLog.CreateEventSource(logName, logName);

        // Add a new error event to the log
        EventLog log = new EventLog();
        log.Source = logName;
        log.WriteEntry(msg, EventLogEntryType.Error);
    }
    catch (Exception ex)
    {
        // Not much you can do with this except swallow it
        // or output it to debugger
        Debug.WriteLine(ex.Message);
    }
}

</script>
```

In some situations (such as when a site is hosted on a third-party server), you might not have permission to add to or access the server event log. In such a case, you must use some other mechanism, such as sending an email or logging the exception to some other type of file or database.

To send an exception via email, you can use the `MailMessage` class in the `System.Net.Mail` namespace. The next example, following Figure 5.3, illustrates how this class could be used.

---

## CORE NOTE

To test this example, a valid SMTP server must be available to your Web server.

---



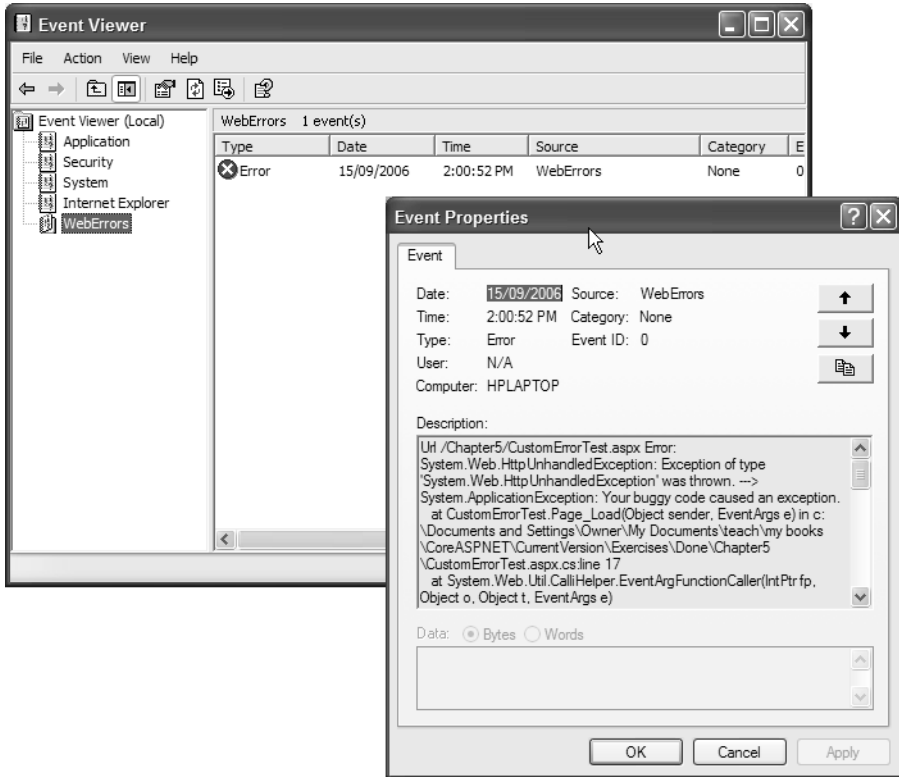


Figure 5.3 Viewing an exception in the Event Viewer

```

<%@ Application Language="C#" %>

<%@ Import Namespace="System.Net.Mail" %>

<script runat="server">

...

void Application_Error(object sender, EventArgs e)
{
    // Construct the mail message
    MailMessage mail = new MailMessage();
    mail.To.Add(new MailAddress(
        "administrator@buggysoftware.com"));
    mail.From = new MailAddress(
        "autogenerated@buggysoftware.com");
  
```

```
mail.Subject = "Critical Application Exception";
mail.IsBodyHtml = true;

string body = "<html><head></head><body>";
body += "<h1>" + Request.Path + "</h1>";
body += "<h2>" + DateTime.Now + "</h2>";
body += Server.GetLastError().ToString();
body += "</body></html>";
mail.Body = body;

// Send the email
SmtpClient mailer = new SmtpClient();
mailer.Host = "host name or IP address goes here";
try
{
    mailer.Send(mail);
}
catch (Exception ex)
{
    // Not much you can do with this except output
    // it to debugger
    Debug.WriteLine(ex.Message);
}
}
</script>
```

As an alternative to hardcoding the email addresses and the IP address, you could define these items in the `appSettings` section of the `Web.config` file. The `appSettings` section can be used to define any custom application-specific values. For instance, you can add the following items to `appSettings` section.

```
<configuration>
  <appSettings>
    <add key="Error_ToEmail"
        value="administrator@buggysoftware.com"/>
    <add key="Error_FromEmail"
        value="autogenerated@buggysoftware.com"/>
    <add key="Error_SmtpHost"
        value="host name or IP address goes here"/>
  </appSettings>
  ...
</configuration>
```

You can access these values via the `ConfigurationManager.AppSettings` method (in the `System.Configuration` namespace). You can change your `Application_Error` method to use this method to retrieve these values, as shown in the following.

```

string to = ConfigurationManager.AppSettings["Error_ToEmail"];
mail.To.Add(new MailAddress(to));
string from =
    ConfigurationManager.AppSettings["Error_FromEmail"];
mail.From = new MailAddress(from);
...
mailer.Host = ConfigurationManager.AppSettings["Error_SmtpHost"];

```

Finally, another way of handling an application error is to output the exception information to your own custom exception log file. In the example in Listing 5.1, the `Application_Error` handler outputs each exception to a text file in the `App_Data` folder of the application if the `Error_ShouldLogErrors` flag in the `Web.config` file is set to `true`. This flag is defined in the `appSettings` section of the `Web.config` file, as shown here.

```
<add key="Error_ShouldLogErrors" value="true" />
```

---

#### Listing 5.1 Global.asax

---

```

<%@ Application Language="C#" %>

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

<script runat="server">

    void Application_Error(object sender, EventArgs e)
    {
        try
        {
            // Get the error log flag from Web.config
            string sLogErrors =
                ConfigurationManager.AppSettings[
                    "Error_ShouldLogErrors"];
            bool logErrors = Convert.ToBoolean(sLogErrors);

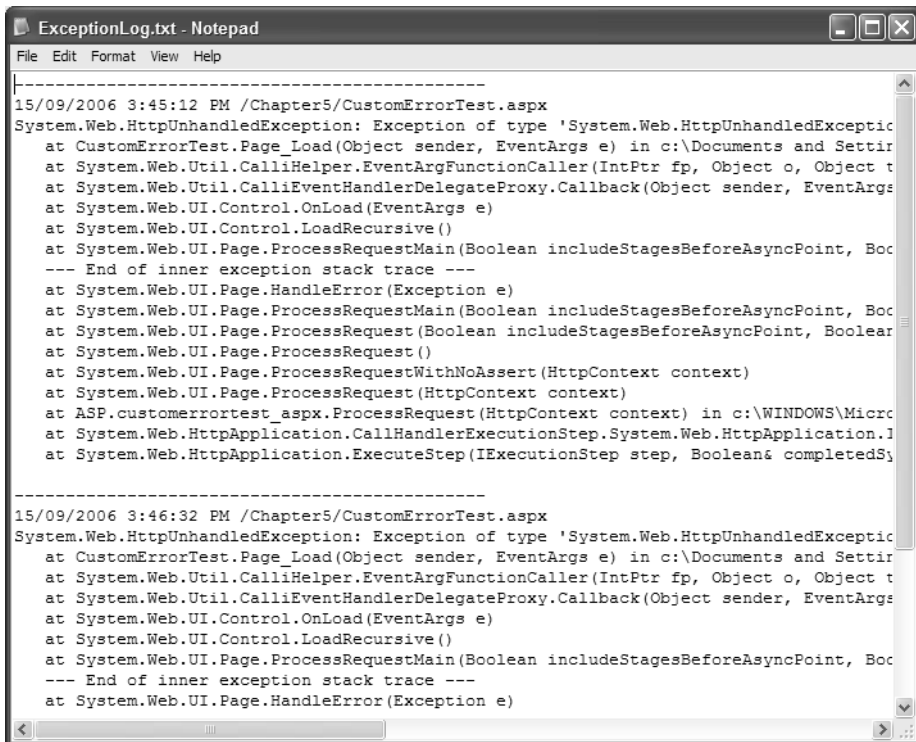
            // Only log errors if Web.config file tells you to
            if (logErrors)
            {
                // Keep the exception log file in App_Data
                string path =
                    Server.MapPath("~/App_Data/ExceptionLog.txt");

                // Append the following to this file
                StreamWriter sw = File.AppendText(path);
                sw.WriteLine("-----");
                sw.WriteLine(DateTime.Now + " " + Request.Path);
                sw.WriteLine(Server.GetLastError().ToString());
                sw.WriteLine();
            }
        }
    }

```

```
        sw.Flush();
        sw.Close();
    }
}
catch (Exception ex)
{
    // Not much you can do with this except output
    // it to debugger
    Debug.WriteLine(ex.Message);
}
}
}
</script>
```

The resulting text file can be seen in Figure 5.4.

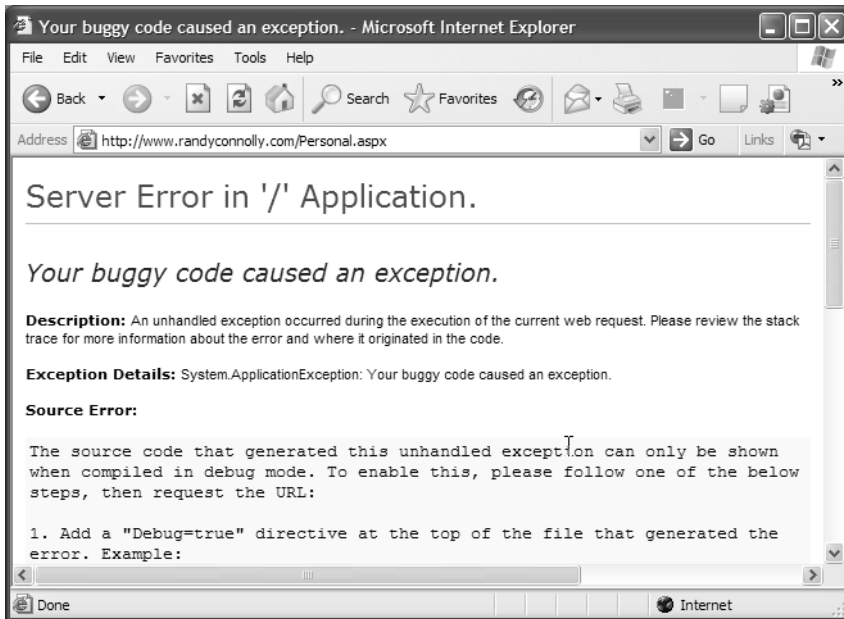


```
-----
15/09/2006 3:45:12 PM /Chapter5/CustomErrorTest.aspx
System.Web.HttpUnhandledException: Exception of type 'System.Web.HttpUnhandledException'
  at CustomErrorTest.Page_Load(Object sender, EventArgs e) in c:\Documents and Settings\...
  at System.Web.Util.CalliHelper.EventArgFunctionCaller(IntPtr fp, Object o, Object t...
  at System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs...
  at System.Web.UI.Control.OnLoad(EventArgs e)
  at System.Web.UI.Control.LoadRecursive()
  at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean...
  --- End of inner exception stack trace ---
  at System.Web.UI.Page.HandleError(Exception e)
  at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean...
  at System.Web.UI.Page.ProcessRequest(Boolean includeStagesBeforeAsyncPoint, Boolean...
  at System.Web.UI.Page.ProcessRequest()
  at System.Web.UI.Page.ProcessRequestWithNoAssert(HttpContext context)
  at System.Web.UI.Page.ProcessRequest(HttpContext context)
  at ASP.customerrortest.aspx.ProcessRequest(HttpContext context) in c:\WINDOWS\Micro...
  at System.Web.HttpApplication.CallHandlerExecutionStep.System.Web.HttpApplication.I...
  at System.Web.HttpApplication.ExecuteStep(IExecutionStep step, Boolean& completedS...
-----
15/09/2006 3:46:32 PM /Chapter5/CustomErrorTest.aspx
System.Web.HttpUnhandledException: Exception of type 'System.Web.HttpUnhandledException'
  at CustomErrorTest.Page_Load(Object sender, EventArgs e) in c:\Documents and Settir...
  at System.Web.Util.CalliHelper.EventArgFunctionCaller(IntPtr fp, Object o, Object t...
  at System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs...
  at System.Web.UI.Control.OnLoad(EventArgs e)
  at System.Web.UI.Control.LoadRecursive()
  at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean...
  --- End of inner exception stack trace ---
  at System.Web.UI.Page.HandleError(Exception e)
```

Figure 5.4 Event logging to a text file

## Using Custom Error Pages

If you do not use the `Context.ClearError` method in the `Page_Error` or the `Application_Error` handler, ASP.NET redirects to the default ASP.NET error page. By default, ASP.NET only shows this detailed error page to local users (i.e., the developer). Remote users see a different ASP.NET error page that does not contain all the exception details, as shown in Figure 5.5.



**Figure 5.5** Default error page for remote users

You can replace the default ASP.NET error page with your own custom page, as shown in Figure 5.6.

To use a custom error page, you can change the settings of the `<customErrors>` element in the `Web.config` file. In this element, you can specify the custom page that is to be displayed, as shown in the following.

```
<system.web>
  <customErrors mode="On"
    defaultRedirect="FriendlyErrorPage.aspx" />
  ...
</system.web>
```

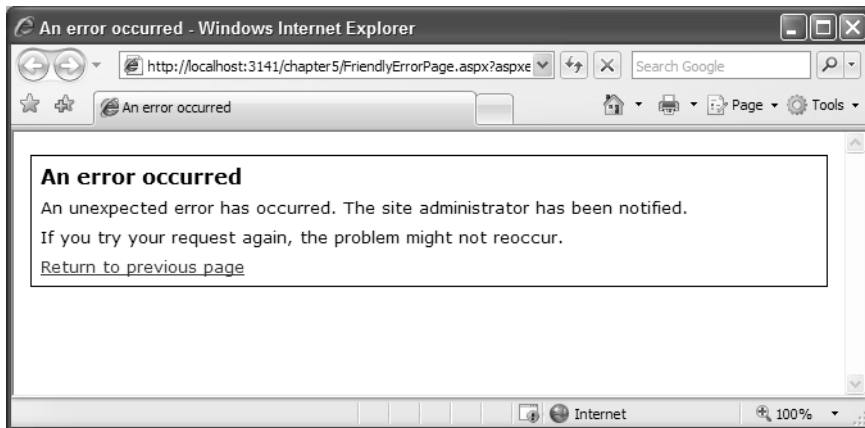


Figure 5.6 Custom error page

Setting the `mode` attribute to `On` means that the ASP.NET details page is not shown, even to local users. If you want local users to still see the default details page, change the mode to `RemoteOnly`.

## Handling Common HTTP Errors

ASP.NET allows you to create custom error pages for different HTTP error codes. For example, a common feature of many Web sites is to provide custom HTTP 404 (requested page not found) and HTTP 500 (server error) error pages. You can specify custom pages for HTTP error codes within the `<customErrors>` element, as shown in the following.

```
<customErrors mode="On"
  defaultRedirect="FriendlyErrorPage.aspx" >
  <error statusCode="404" redirect="custom404.aspx" />
  <error statusCode="500" redirect="custom500.aspx" />
</customErrors>
```

## Using the Validation Server Controls

Data is at the heart of most Web applications. Typically, Web applications require users to fill in form data, which is then used as a basis for other application actions. User data is also often persisted in some form, such as to databases or to XML files. Unfortunately, users can be quite peculiar. They can forget to enter certain important fields in a form, type in something preposterous into a field, or even willfully try to subvert an application's security by entering a rogue script into a form text field.