

Technical Notes on Error Logging Modules & Handlers (ELMAH)

Revision 236

Written by [Atif Aziz](#)

Principal Consultant, [Skybow AG](#)

Applies to [Microsoft ASP.NET 1.1](#) and [ELMAH v1.0 \(Build 5527\)](#)

See also: [GotDotNet Workspace for ELMAH](#)

Table of Contents

Purpose of This Document	3
Overview	3
Quick Start	7
A Note on the Implementation	19
Architectural Overview	19
Error: The Phantom Exception	20
ErrorLog and ErrorLogEntry	24
SqlErrorLog	25
MemoryErrorLog	27
Binding to the Log Implementation	28
The Handlers to the ErrorLog	30
ErrorMailModule	32

Purpose of This Document

ELMAH was originally introduced with the [MSDN](#) article “[Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components](#).” The goal of the article was to demonstrate how HTTP modules and handlers can be used in ASP.NET to provide a high degree of componentization that goes just beyond the classical reuse through controls. It was not the goal of the article to discuss the implementation details of ELMAH or the background to some of its design. That is the purpose of this document in the form of technical notes and using a casual voice.

Please bear in mind that this is a “living” document that will be expanded as needed.

Overview

ELMAH provides two HTTP modules and a set of HTTP handlers that can be used as a foundation for a complete error logging, notification and display solution for web applications. You can enable ELMAH for one or more web applications or for all web applications running on a machine. All you have to do is deploy a single assembly and make changes to the configuration file. There is no need to recompile or re-deploy an application.

The primary goal of ELMAH 1.0 is to demonstrate, by way of example, how HTTP handlers and modules can be used as a very high-level form of componentization, enabling entire sets of functionalities to be developed, packaged and deployed as a single unit and independent of web applications.

Here’s how it works. There are two independent modules, called [ErrorLogModule](#) and [ErrorMailModule](#). Both of these subscribe to the [Error](#) event of [HttpApplication](#) in order to listen for unhandled exceptions that bubble out of the main web application code. Exceptions that are handled and swallowed by anyone along the stack are never seen by these modules, including those cleared using the [ClearError](#) method on an [HttpContext](#) instance.

The [ErrorMailModule](#) is the simpler of the two, so let’s talk about that one first. When it receives the [Error](#) event, it creates an e-mail message, writes out the error details in the body

and sends it off to a designated address. The solution comes with a standard implementation that formats the error as an HTML document, as shown in Figure 1. You can also provide your own implementation in case you don't like the default formatting.

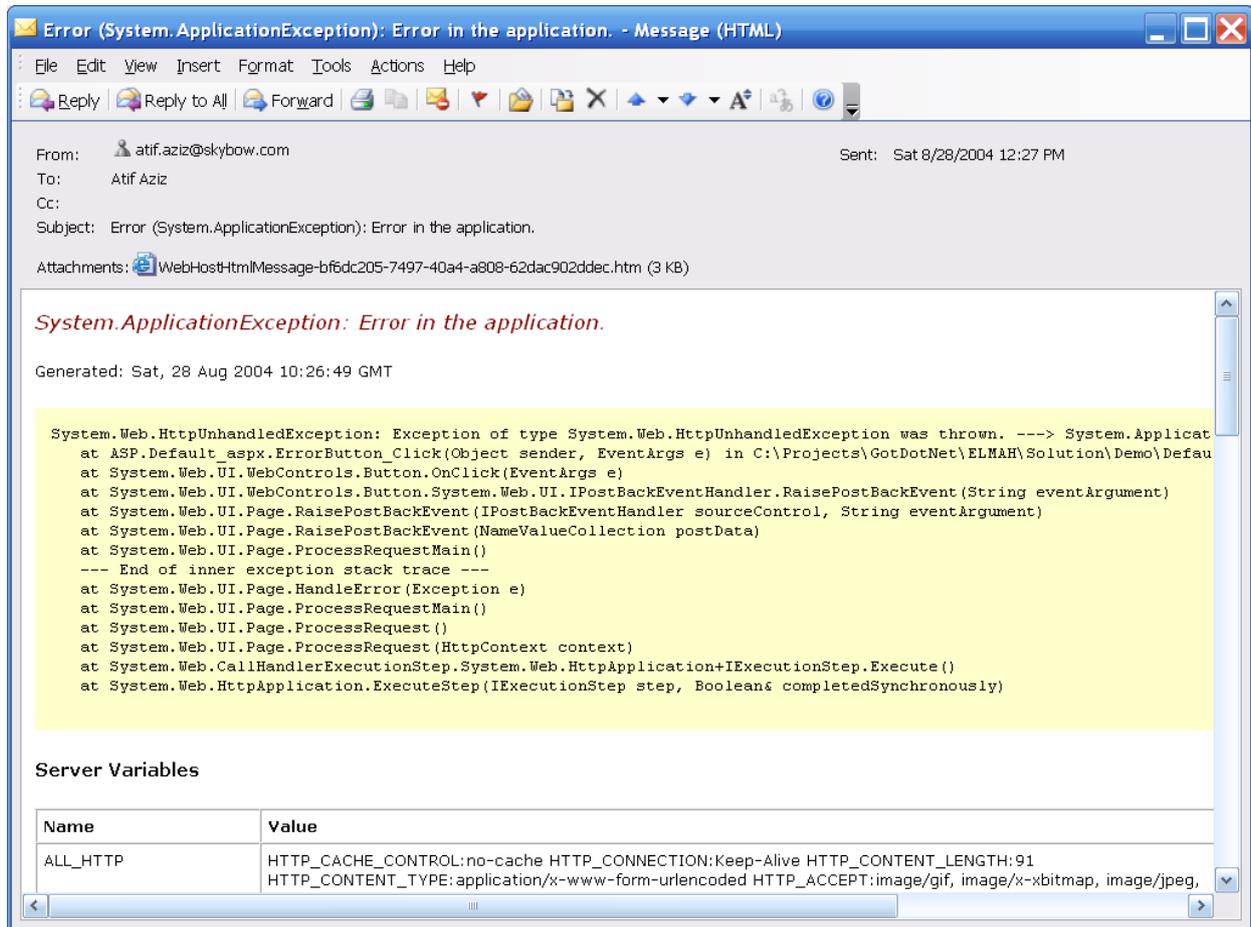


Figure 1

This basically takes care of the notification of errors that occur in a web application. There's a second form of notification, but I'll talk about that a little later.

When the **ErrorLogModule** receives the **Error** event, it goes ahead and logs it to a store. The store is defined by an implementation of the abstract **ErrorLog** class. The solution comes with a concrete implementation for Microsoft SQL Server 2000¹ that resides in the **SqlErrorLog** class and an in-memory implementation that resides in the **MemoryErrorLog** class. The majority of the remaining types in the solution are a set of handlers that provide the user interface for

¹ The reason for requiring Microsoft SQL Server 2000 is because I rely on some of the XML features as a shortcut, but nothing stops from back-porting the implementation to work with Microsoft SQL Server version 7.0. You may also be able to use MSDE 2000, but the solution hasn't been tested against it.

viewing the log. Like the module, the handlers also work off an **ErrorLog** implementation so if you go ahead roll your own implementations over, say the file system, Microsoft Access or an Oracle database, then the handlers will work against them too. Figure 2 shows the results from one of the handlers that displays a summary of the latest errors recorded for an application.

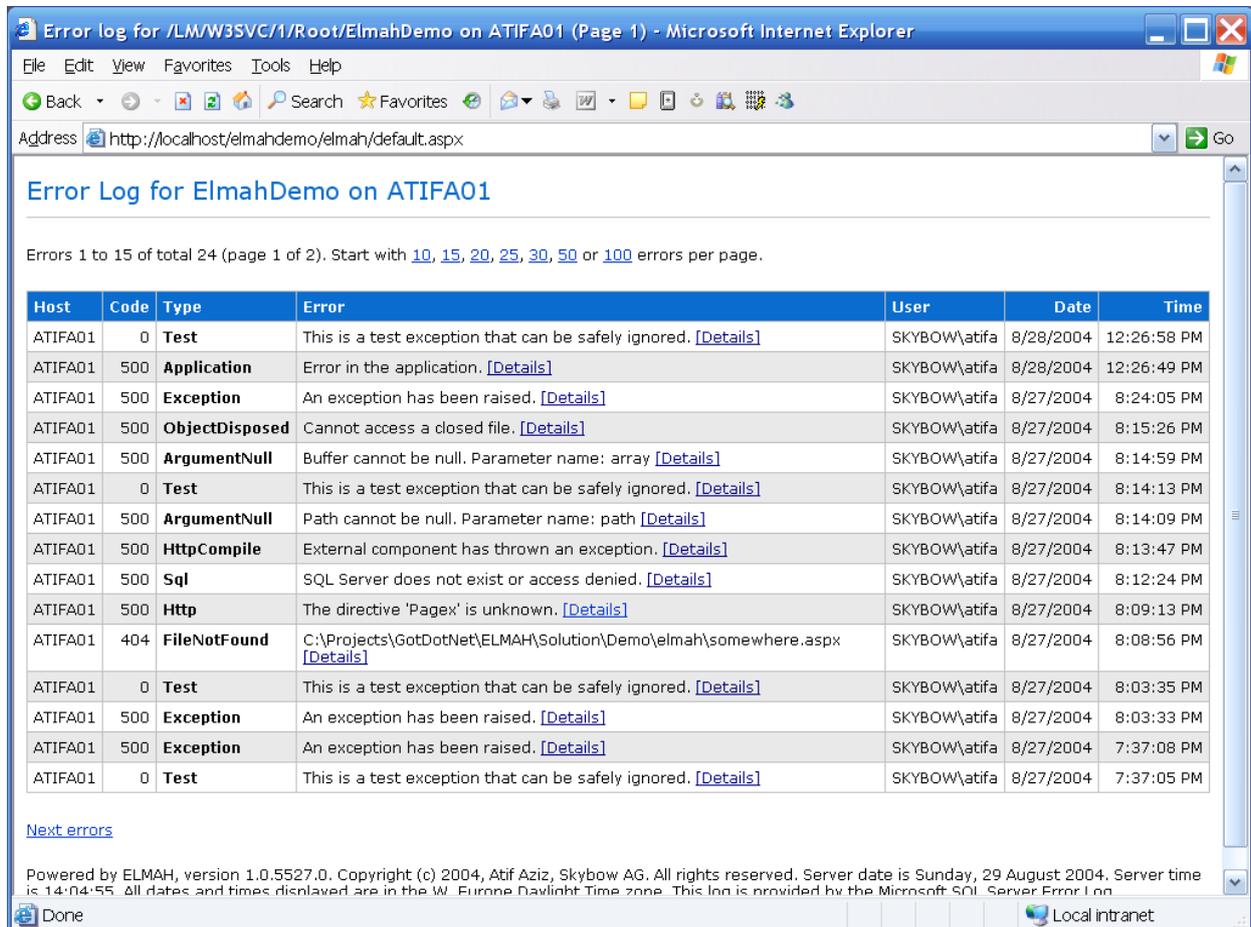


Figure 2

When one of the [Detail] links is clicked, another handler is invoked that renders the details of the selected error on a separate page. Figure 3 shows one such page in action.

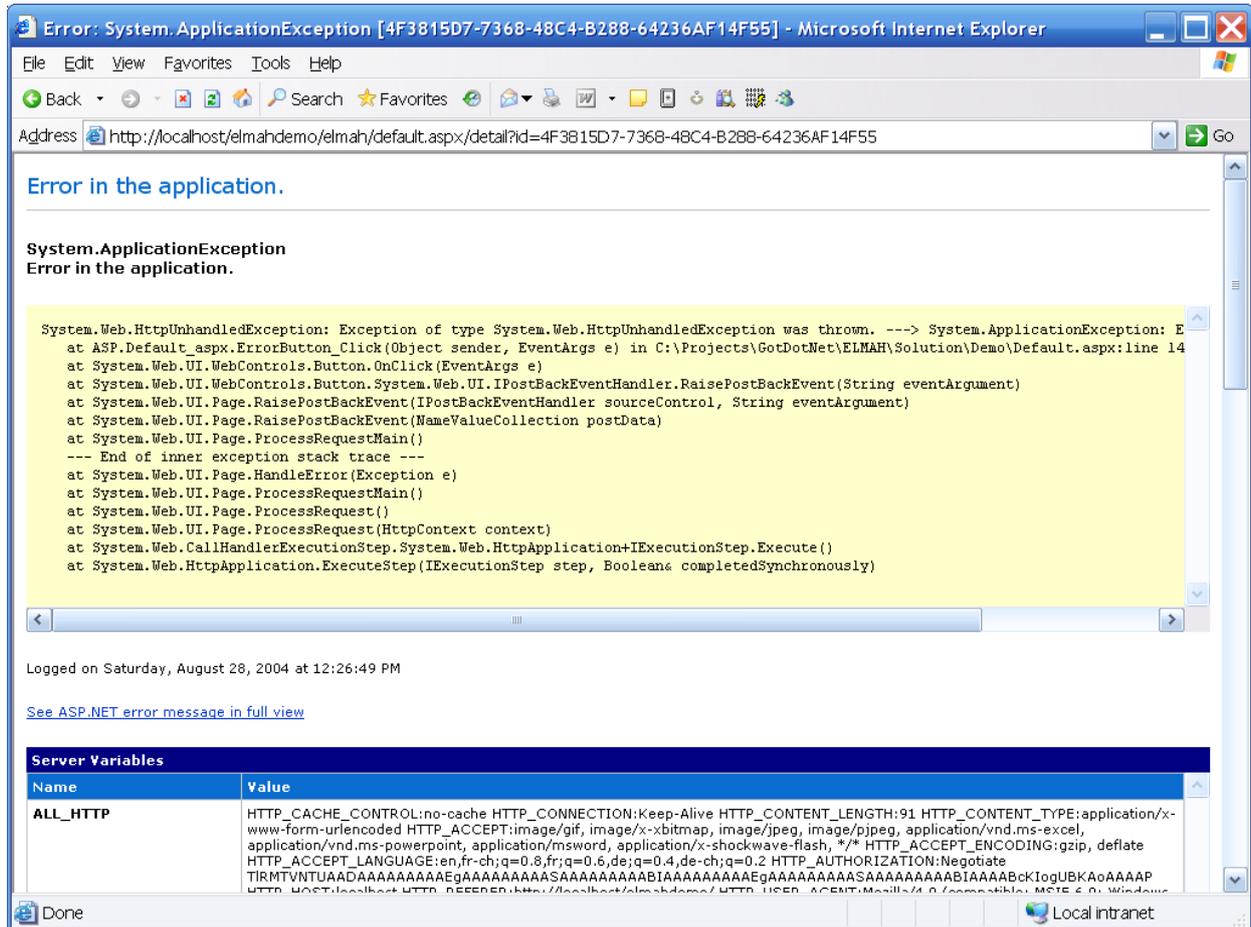


Figure 3

These days, you can't write a web-based solution that's considered fashionable without involving [RSS](#) in one way or another. So to be totally *en vogue*, one of the handlers that I've provided renders the last 15 errors recorded in the error log as a RSS feed. This is the second form of notification that I mentioned earlier. A sample output of the RSS feed can be seen in Figure 4.

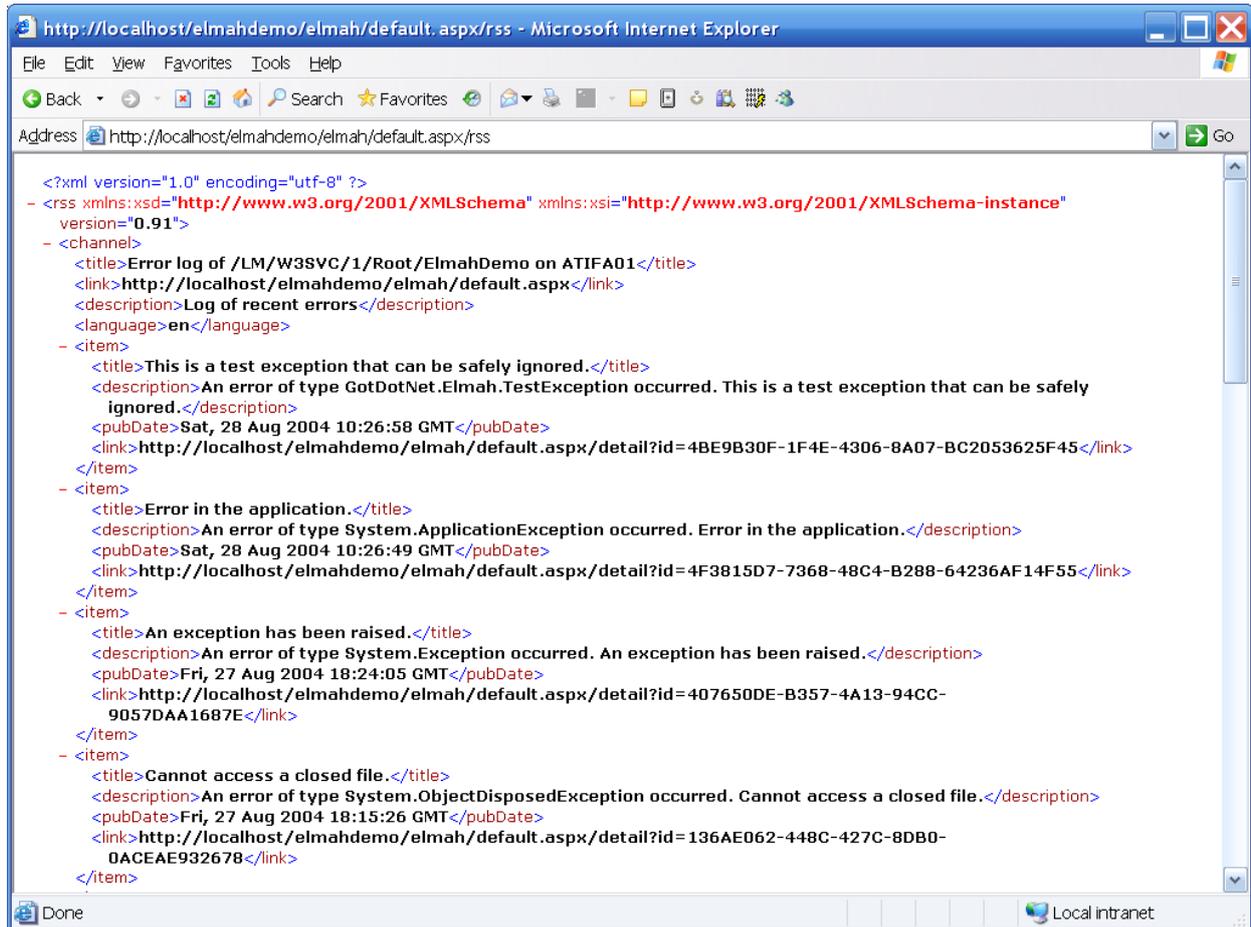


Figure 4

It's not as detailed as the mail-based counterpart, of course, but it nevertheless allows developers, administrators and operators alike to use their favorite RSS aggregator to receive most recently logged errors as news items. If your RSS aggregator also pops a toaster sort of window à la instant messengers like MSN Messenger, then the errors will really get your attention. Sometimes, e-mails can take time to make their way to your inbox as they are pushed through the various systems. With an RSS news aggregator, however, you are in pull mode and consequently more in control of how often a feed is polled. Either way, you'll get a notification through a pull or push method.

Quick Start

So with all these modules and handlers in hand, the next question is how to use and enable them in a web application. Let's start with a tour for enabling logging for a single application. Open the **web.config** for any desired web application and add the following line to the **<httpHandlers>** section:

```
<add verb="POST,GET,HEAD" path="elmah/default.aspx"
type="GotDotNet.Elmah.ErrorLogPageFactory, GotDotNet.Elmah,
Version=1.0.5527.0, Culture=neutral, PublicKeyToken=978d5e1bd64b33e5" />
```

Copy the **GotDotNet.Elmah.dll** assembly to the **bin** folder and that's it. At this point, you can navigate to the specified path² and see a page similar to Figure 5:

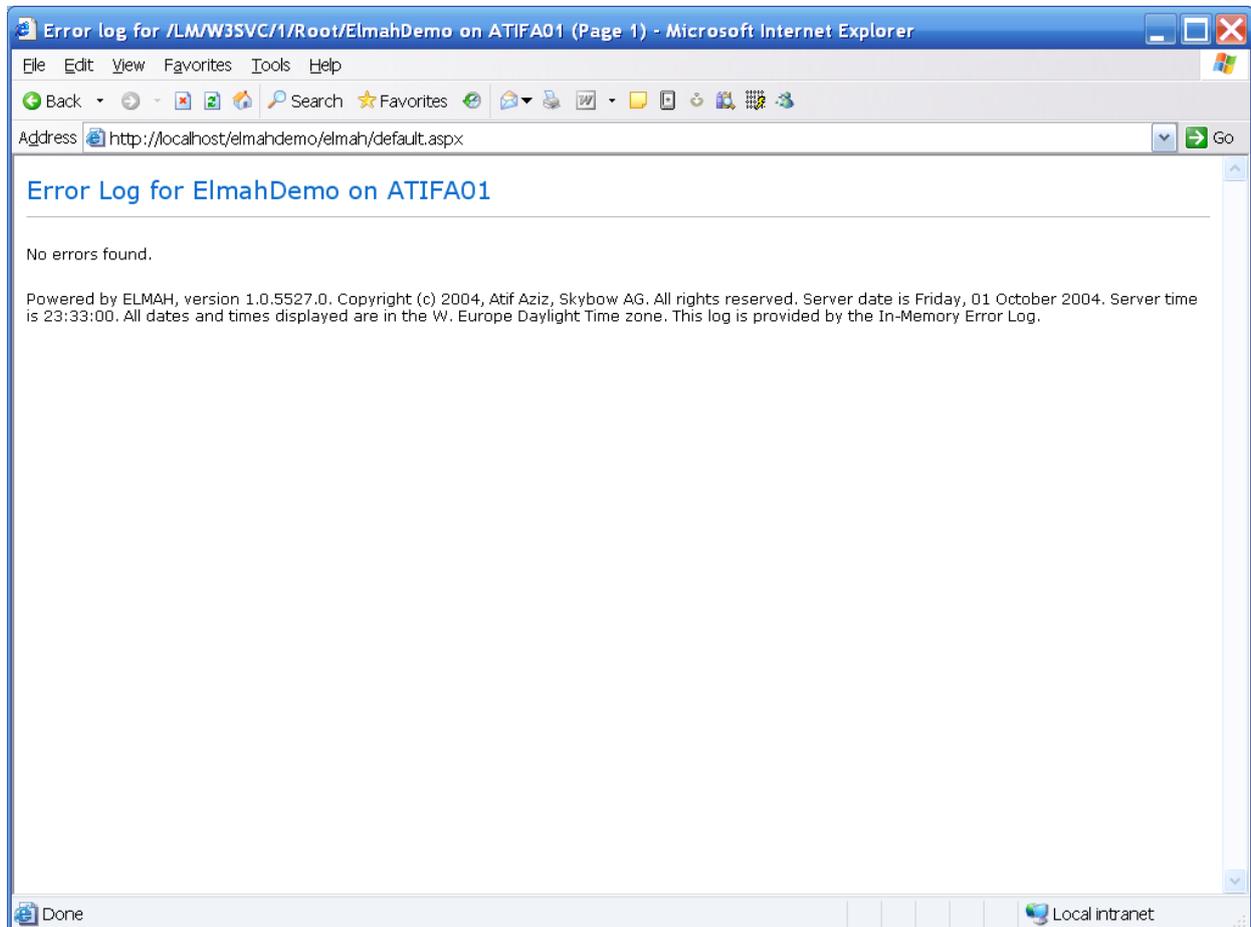


Figure 5

Since there are no errors yet, the page is pretty much empty. If you generate an exception in your application, you still won't see any errors here. This is because we haven't added the logging module, so let's do that now. Go to the **<httpModules>** section and add the following line:

```
<add name="ErrorLog" type="GotDotNet.Elmah.ErrorLogModule, GotDotNet.Elmah,
Version=1.0.5527.0, Culture=neutral, PublicKeyToken=978d5e1bd64b33e5" />
```

² Of course, you can configure any path you like and it will serve as the root of all handlers in the solution.

If you generate an error now then the error log will display it. If you can't think of a way to generate an exception and test the functionality then don't worry. The solution comes with a **TestException** that can be generated at any time by appending **/test** to the path for the handlers. So in the address bar of the browser, type:

<http://www.example.com/myapp/elmah/default.aspx/test>

If custom errors are disabled then you should see the standard ASP.NET error page that looks like this (assuming that [custom errors](#) are off):

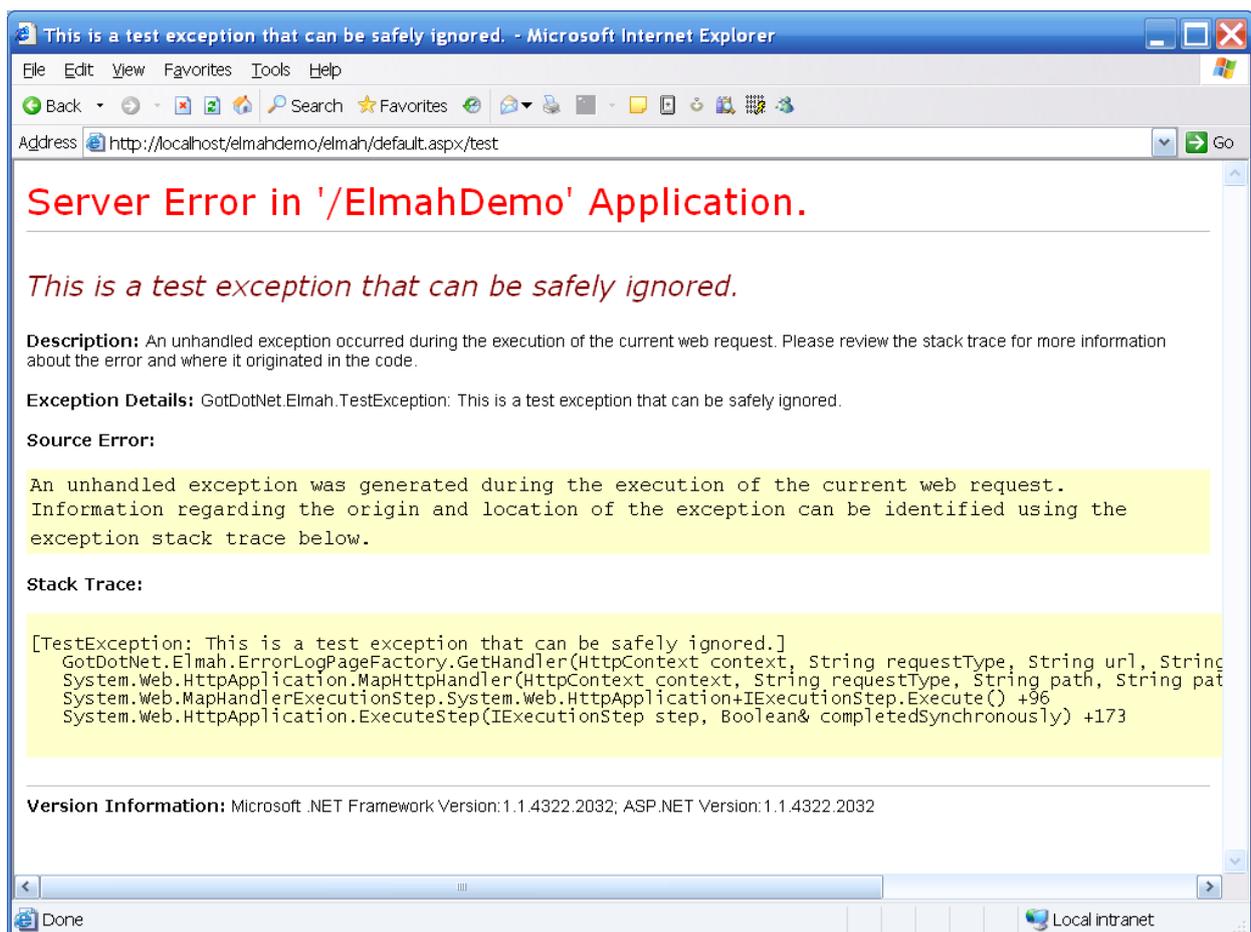


Figure 6

The difference this time is that the log also contains a record of this error so let's check that out. Go back to error log root page³ and this time you should see an entry in there:

³ The error log root page is considered the path configured in the entry added to `<httpHandlers>`.

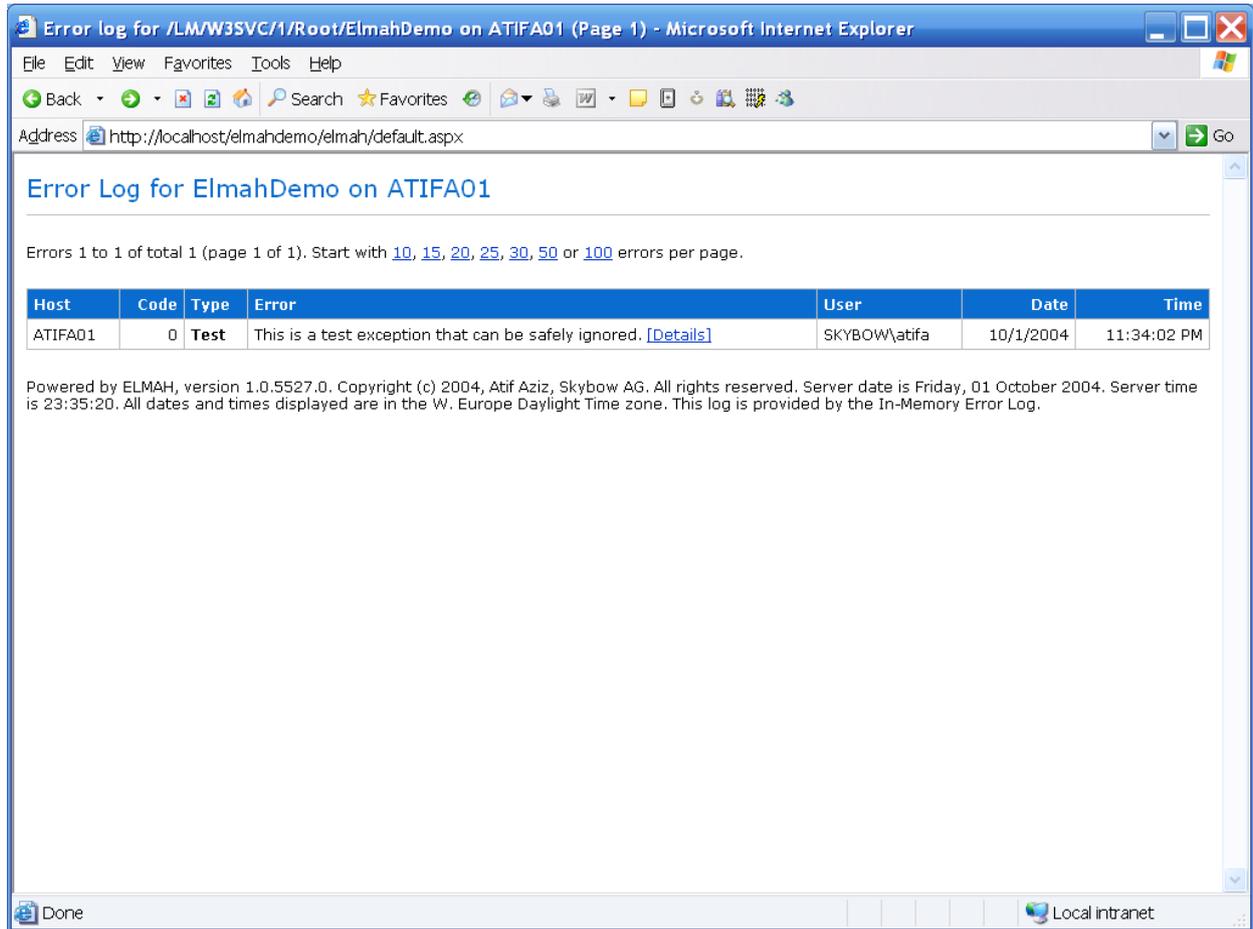


Figure 7

Now click on the link next to the error message to see the details of the error entry. The next page should look similar to this:

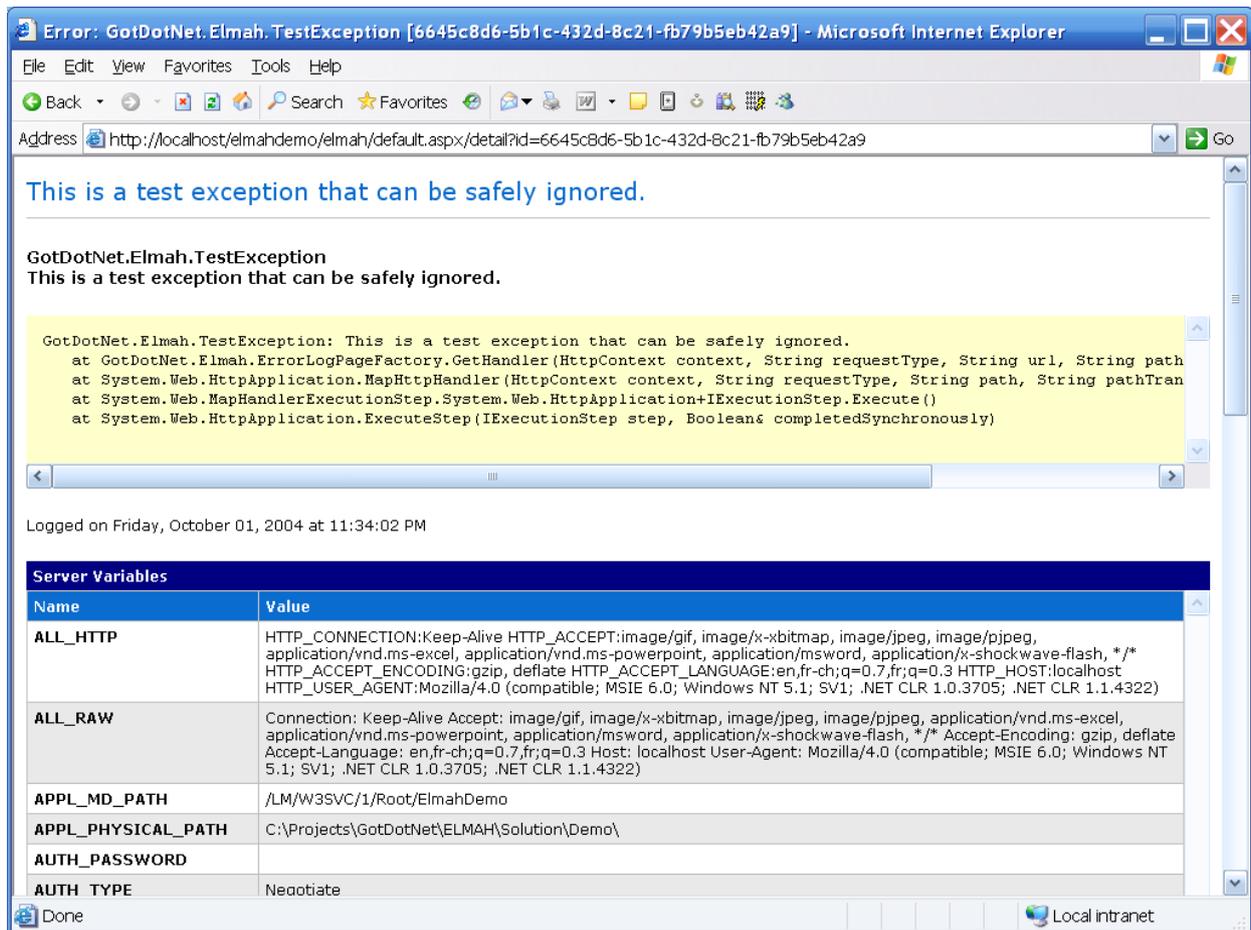


Figure 8

Note that in addition to the basic error information, web collections such as the server variables are also captured and displayed. Now let's generate another exception. In this case, we're going to make a fault in the error page handler itself. In the query string, you should see a parameter named "id" whose value is a GUID. Change it so that it no longer is a valid GUID. For example, replace the last character with an 'x'. Not surprisingly, you should now see a [FormatException](#):

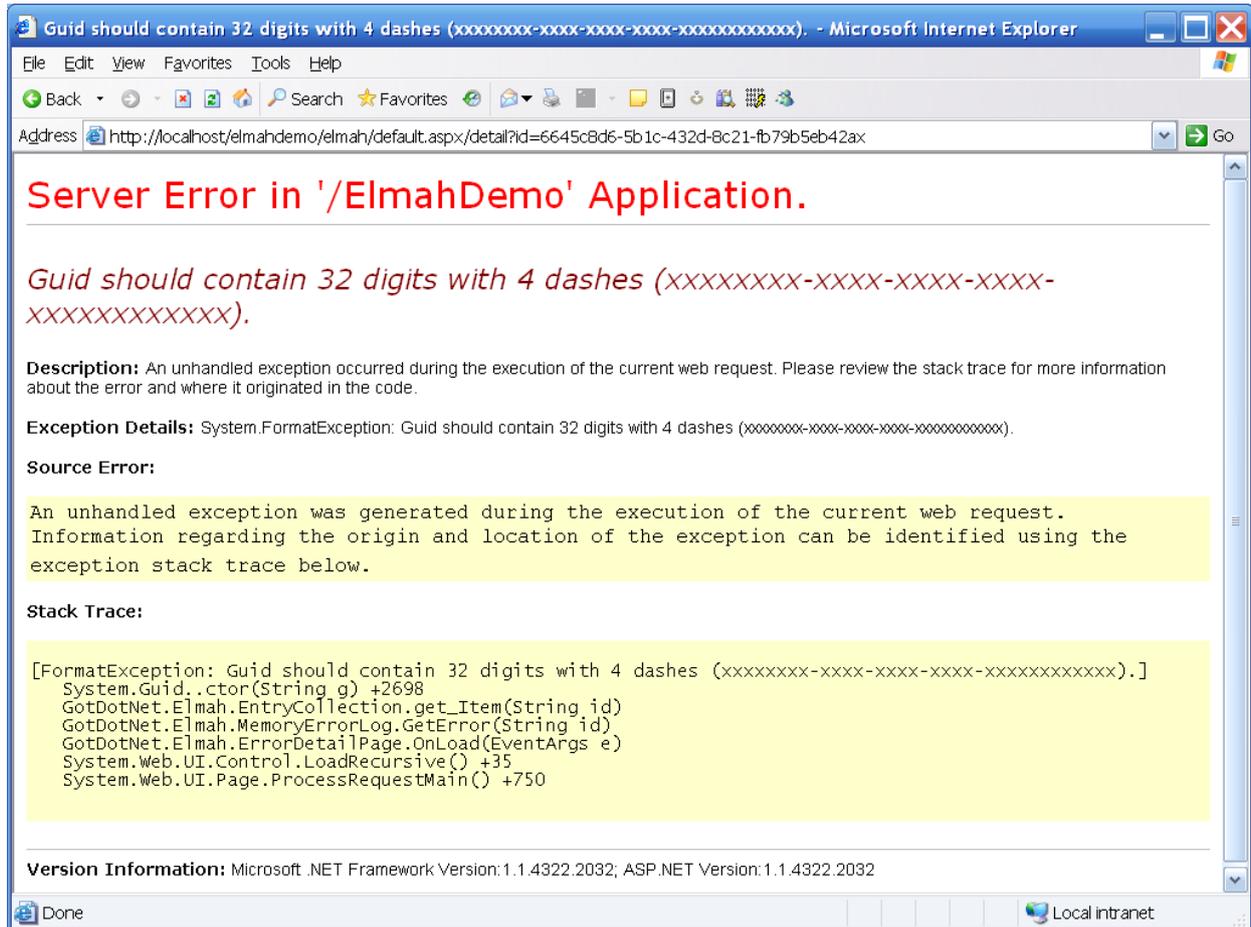


Figure 9

Now let's go back and see what's in the log⁴. You should see two entries:

⁴ If you've been following the tutorial strictly then you can just press the Back button twice in the browser. You may also have to click Refresh button to get the latest updates.

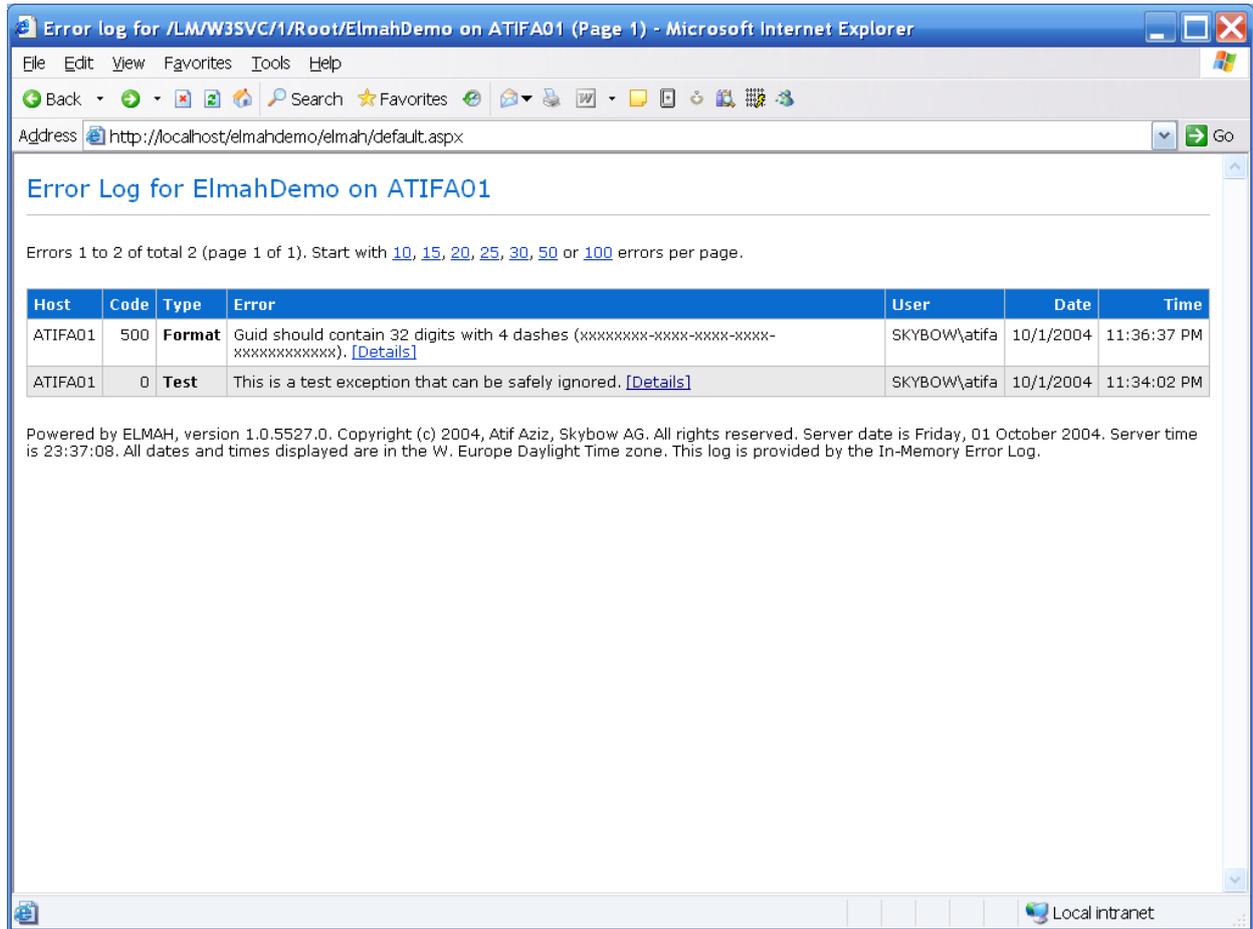


Figure 10

If you dig into the details of this new entry, you should notice something different this time:

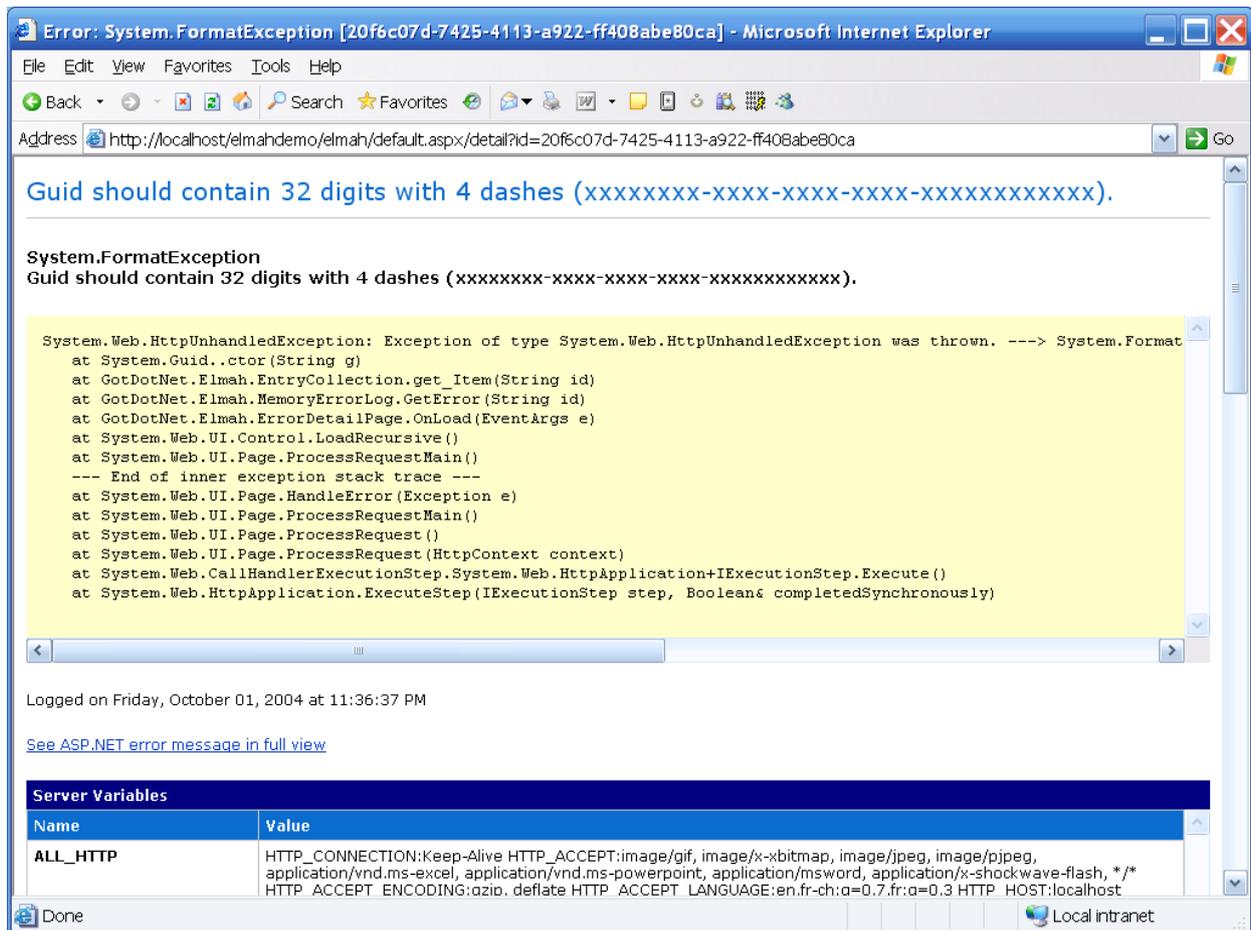


Figure 11

There's an extra link that says, "See ASP.NET error message in full view." When you click this, you'll see a full blown HTML page that is the exact message that ASP.NET generated at the time of the exception!⁵ This means that even if you enable custom errors in your web application (typical of when you go to production, for example), the log can capture the original message that we've all come to love and rely upon. Haven't we all been there? You write a web application, deploy it to production and then you get this dreadful page:

⁵ This can be a little deceiving sometimes. You might think that clicking the link has generated an exception in the application, but that's not the case here. It's really the entire HTML document that was generated by ASP.NET for the exception. It may have been better to use frames to put a small reminder at the top, but the idea was dropped to keep the code size down for the article.

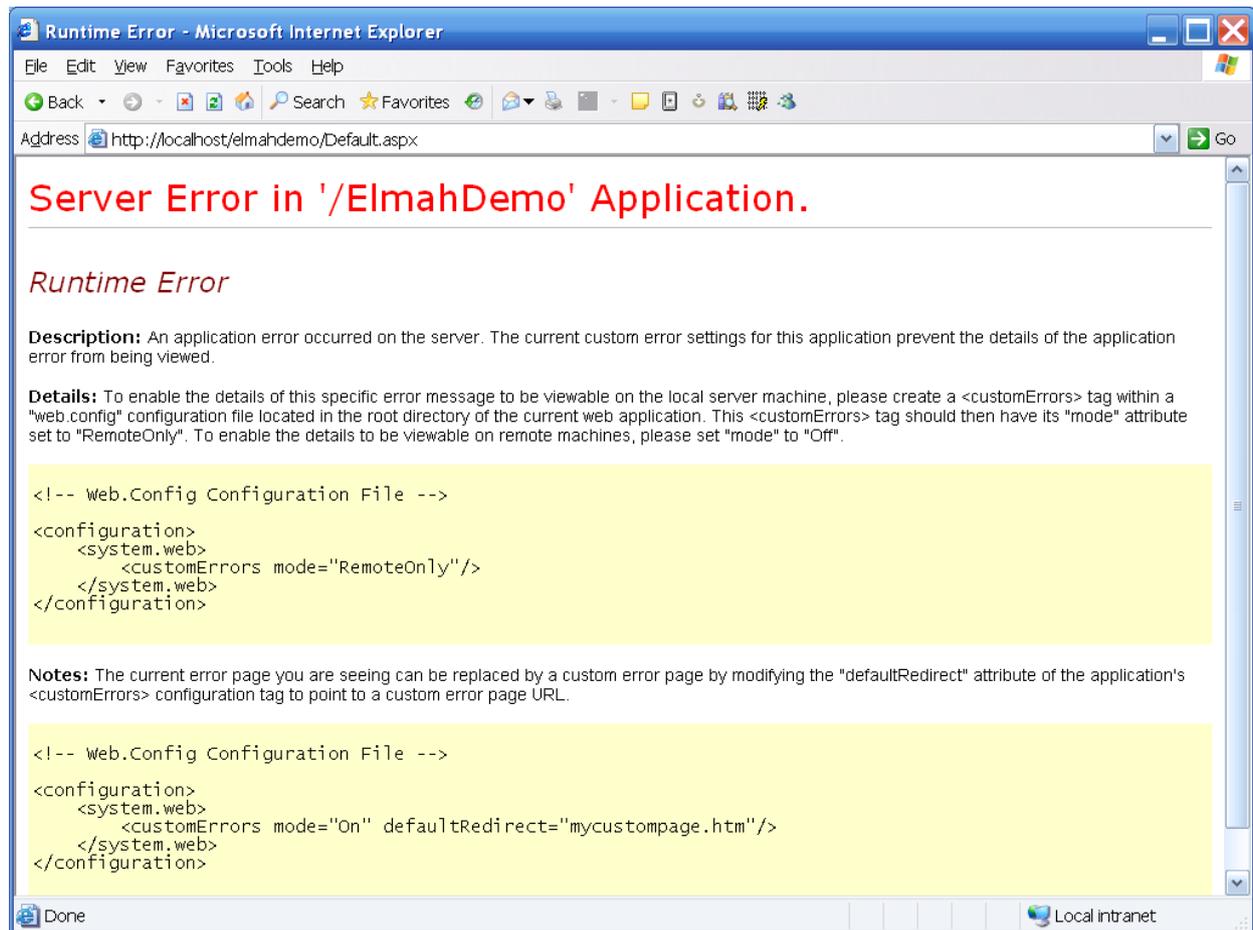


Figure 12

This is the point where you wish you could see what's going on behind. So what's your first instinct? Go back turn off custom errors completely so you can view the message while hoping that no customer hits the web site in the same time frame in which you are troubleshooting. Now with the error log, you don't have to worry about that anymore. If you see this page, go to the error log and get the full-blown page⁶.

So far, all the logging has been taking place in an in-memory log⁷ that does not survive application restarts or failures. In fact, if you just *touch* your **web.config** by saving it then the application will restart and you'll notice that the log is empty again. Let's try something more

⁶ This is implemented using a hidden gem in the .NET Framework. See the [GetHtmlMessage](#) method of the [HttpException](#) class. So why did this link not show up the first time? Unfortunately, ASP.NET does not always provide the contents of the error message HTML. It depends on when the exception occurs in the execution of the HTTP pipeline.

⁷ This is the default when no log is configured. The current log implementation in effect is displayed in the footer of the error log pages.

robust and reliable like using SQL Server as the store for the log. Create a new database called **ELMAH** and run the supplied SQL script⁸ to create the required table and related stored procedures.

At this point we have to add a new section to the configuration file, right at the top:

```
<configSections>
  <sectionGroup name="gotdotnet.elmah">
    <section name="errorLog"
type="System.Configuration.SingleTagSectionHandler, System,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
/>
  </sectionGroup>
</configSections>
```

These entries basically introduce new configuration elements to the runtime's configuration system, telling it that if it sees a certain element in the configuration file, then it should use a certain **[IConfigurationSectionHandler](#)** implementation to process its contents. Here, **<section>** is adding a new section called **errorLog**, whose handler is in fact **[a factory-supplied section handler](#)** from the **[BCL \(Base Class Library\)](#)**. The **<sectionGroup>** does exactly what its name suggests. It just introduces a grouping element. There is no handler class associated with it. If you've got several related sections then it's usually a good idea to group them together. We'll be introducing yet another section later on so that's why the group is being defined at this time. It often helps to think of a section group as a namespace.

Now we can go ahead and configure the error log as follows:

```
<gotdotnet.elmah>
  <errorLog type="GotDotNet.Elmah.SqlErrorLog, GotDotNet.Elmah,
Version=1.0.5527.0, Culture=neutral, PublicKeyToken=978d5e1bd64b33e5"
connectionString="Server=.;Database=ELMAH;Trusted_Connection=True" />
</gotdotnet.elmah>
```

The **type** attribute on **errorLog** specifies the class that now serves as the error log implementation (**SqlErrorLog**). Change the **connectionString** attribute as needed. With this in place, you can try to generate once more some exceptions (like the test one earlier) and see that errors are indeed logged into the database and survive the application's lifetime.

⁸ Assuming that you chose that proposed installation location during the setup of ELMAH, the script can be found using the path **|Program Files|GotDotNet|ELMAH|1.0|src|Database.sql** on the selected drive. The SQL script does not create the database in case you want to integrate it into an existing database.

You can also get to the RSS feed by simply appending `/rss` to the root of the error log's path. Go ahead and try it against your favorite RSS news aggregator application.

Now let's add the final piece, which is getting an e-mail upon an exception. For this, add the following module to the configuration file:

```
<add name="ErrorMail" type="GotDotNet.Elmah.ErrorMailModule,
GotDotNet.Elmah, Version=1.0.5527.0, Culture=neutral,
PublicKeyToken=978d5e1bd64b33e5" />
```

And the following section handler under the `gotdotnet.elmah` section group registered earlier:

```
<configSections>
  <sectionGroup name="gotdotnet.elmah">
    <section name="errorLog"
type="System.Configuration.SingleTagSectionHandler, System,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
/>
    <section name="errorMail"
type="System.Configuration.SingleTagSectionHandler, System,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
/>
  </sectionGroup>
</configSections>
```

The new section can now be configured as shown error:

```
<gotdotnet.elmah>
  <errorLog type="GotDotNet.Elmah.SqlErrorLog, GotDotNet.Elmah,
Version=1.0.5527.0, Culture=neutral, PublicKeyToken=978d5e1bd64b33e5"
connectionString="Server=.;Database=ELMAH;Trusted_Connection=True" />
  <errorMail to="john.doe@example.com" />
</gotdotnet.elmah>
```

The minimal setting required by the error mailing module is the recipient's e-mail address, which is specified here using the `to` attribute⁹. Now if the application generates any exceptions then you should receive an e-mail in addition to it being logged in the database.

Note that all of the modules and handlers function fairly independent of each other so you can enable and disable them individually. For example, if you remove entry that registers the HTTP

⁹ You can specify more than one recipient by delimiting addresses with semi-colon (;).

handler with ASP.NET then you will not be able to view the error log although logging will continue working. If you remove the entry for the logging module instead, then the log can still be viewed via the handlers but no new errors will be logged. The same is applicable for the mailing module.

Note also the error log handlers can be secured using the normal role-based security of ASP.NET. You'll need to just [use the location element on the configured path](#).

Up to this point, we've only considered a single web application. However, if you just take all the settings and drop them into the **machine.config** then the same facility becomes available machine-wide and to all web applications. You don't even have to copy the assembly to the **bin** directory of each application since it will go in the GAC¹⁰. You can append **elmah/default.aspx** to the virtual root of any application and get its private error log¹¹. The error mail facility will also be automatically available to all applications. Of course, each individual application is free to remove entries from the sections related to handlers and modules and disable the feature. In fact, a single application can even choose its own error log store by changing the connection string in its local **web.config**. All other settings will be just inherited from the machine configuration file. This really illustrates the awesome power of componentizing web application aspects this way and is the heart of the approach demonstrated by ELMAH. Even an Application Server Provider (ASP) could offer error logging, viewing and mailing to all hosted applications as a built-in facility without requiring any change in the hosted application code.

So start thinking about how you can refactor some aspects of your web application into handlers and modules. You should aim to reach a level of componentization that allows applications to acquire some functionality by simply deploying the assembly and adding some configuration. In the next part of the article

¹⁰ The solution assembly is strong-named so it can be added to the GAC.

¹¹ ELMAH automatically isolates logs of each application. It is not possible for one application to view the log of another.

A Note on the Implementation

At this stage we're ready to dig into the implementation details, but before doing that, I'd like to make a note about the design philosophy of ELMAH as sample solution. Bear in the mind that the overall goal of ELMAH (as a sample accompanying the original article) was to demonstrate an approach to componentizing an entire aspect of a web application. It is sample code after all and such its purpose is not to provide a comprehensive solution in terms of customization. The sample tries to do a few things and do them well, even though I've gone to some lengths to encapsulate implementation details and provide extensibility wherever possible¹². The accompanying code is a vastly reduced version of a full-blown solution that is being used in production applications. When trimming down the code, I had to make a lot of hard decisions about what to keep and what to cut out. In the end, the point was to provide a reference implementation of the overall idea being illustrated by the article. One size barely fits all, so you are more than welcome to take the sample as the foundation for your own version. One of the major guiding factors was that I did not want the number of classes to sprawl out of control by factoring out as many ideas into their leanest and meanest set of interfaces and protocols.

Architectural Overview

Figure 13 depicts an architectural overview of the main classes in the solution. The diagram is not based on any particular modeling system or theory. It roughly shows how the classes relate to each other and what they do using a simple, convenient and self-explanatory notation.

¹² There are many places where changing the behavior of ELMAH does not require you to change the base source code. A lot of the classes are designed for inheritance (see those that are unsealed and bear virtual members). So it's better to inherit, override functionality and configure your class to run instead of changing the base source.

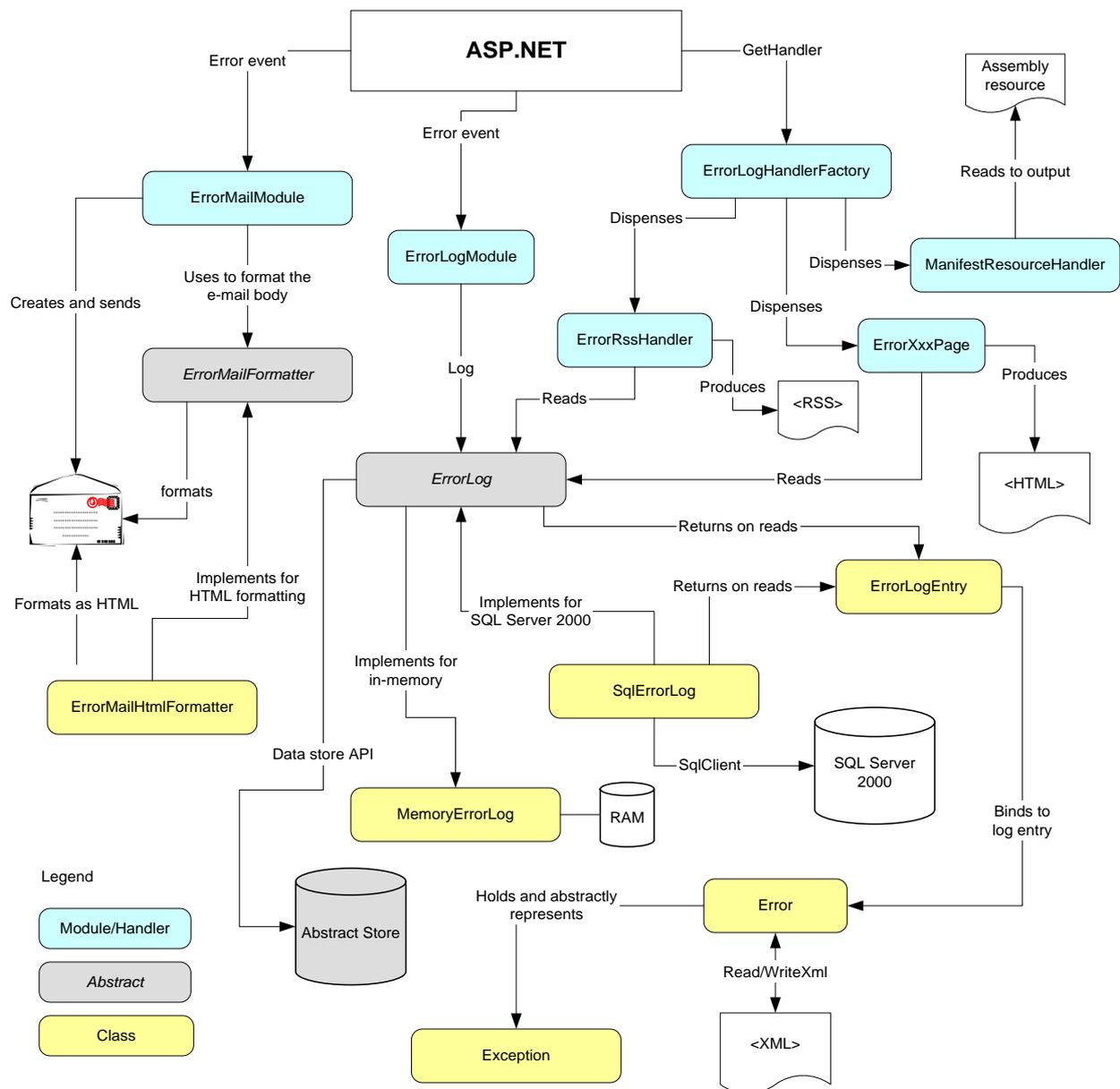


Figure 13

Error: The Phantom Exception

The **Error** class is a central piece of the entire error logging and mailing solution, so it's important to understand it first. One of the problems with exceptions is that they're good for communicating errors along the code stack *and* especially while the code is running, but they don't make great candidates for logging or long-term storage purposes. Sure, they support serialization and you could just blast an exception to some storage as a binary blob, but de-serialization adds a whole number complications if you don't have the right assemblies and types available. This makes them less portable with the containing storage. Moreover, during development, types come and go and details change, but you want the logging architecture to be

rather stable and independent. You also want to be able to ship the log to another machine and view it from there. Actually, as far as the log is concerned, it should be an “informational” trace of the errors that occurred rather than trying to maintain 100% fidelity to the entire state of the exceptions.

This is where the **Error** class comes in. It’s a loose and abstract digest of an exception for informational purposes. Put another way, it’s the lingering “ghost” or remains of a once living and kicking exception. The **Error** is just a holder of properties¹³ deemed fundamental along with some state that may eventually help in diagnosing the error. An example of the latter is web collections like **ServerVariables**, **QueryString**, **Form** and **Cookie**. Some of the properties of the **Error** object like **Source** and **Message** correspond directly to those of an **Exception**. However, almost all properties are “loosely” defined (see following table). That is, the **Error** object does not mandate, for example, that the **Type** strictly reflect the type name of the **Exception** class (although that’s what’s done in the code). It’s really just a string that is meant to be representative of the type of error that occurred. If you want to just put the word “Script” to describe the type of the error then that’s fine. All that said, the **Error** object has constructors that allow you to convert an **Exception** into an **Error**. It will use the properties of the exception to most reasonably fill up the properties of the **Error** object. If you also provide an **HttpContext** instance, then it will make a copy of the web collections associated with the request for diagnostic purposes¹⁴.

Property	Description
Exception	The Exception instance represented by this error. This is a run-time property only that is not persisted during XML serialization.
ApplicationName	The name of application in which this error occurred.
HostName	The name of host machine where this error occurred. A good default is Environment.MachineName .
Type	The type, class or category of the error. Usually this would be the full type name (sans the assembly qualification) of the exception.

¹³ Informally and often known as a *data-centric class*.

¹⁴ Originally, I had the idea of distinguishing basic errors from web-based errors. For example, the web collections would go into a separate class called **WebError** that had **Error** as its base class. However, I rolled this into one class for the sake of simplicity and to reduce the number of classes in the sample. Again, ELMAH is not an academic exercise into abstracting the notion of errors.

Property	Description
Source	The source of the error, usually the same as the Message property of an Exception object.
Message	A brief text describing the error, usually the same as the Message property of an Exception object.
Detail	Detailed text of the error, such as the complete stack trace.
User	The User logged into the application at the time of the error, such as that returned by Thread.CurrentPrincipal.Identity.Name .
Time	The date and time at which the error occurred. This is always in local time.
StatusCode	The status code being returned in the response header as a result of the error ¹⁵ . For example, this is 404 for a FileNotFoundException .
WebHostHtmlMessage	The default HTML message that the web host (ASP.NET) would have generated in absence of custom error pages.
ServerVariables	A NameValueCollection of web server variables, such as those contained in HttpRequest.ServerVariables .
QueryString	A NameValueCollection of HTTP query string variables, such as those contained in HttpRequest.QueryString .
Form	A NameValueCollection of form variables, such as those contained in HttpRequest.Form .
Cookies	A NameValueCollection of cookies sent by the client, such as those contained in HttpRequest.Cookies .

The **Error** class also sports XML serialization via its **ToXml** and **FromXml** methods. Here's an example of how it looks like¹⁶:

```
<error
  application="/LM/W3SVC/1/Root/ElmahDemo"
  host="ATIFA01"
  type="System.IndexOutOfRangeException"
  message="Index was outside the bounds of the array."
  detail="..."
  user="skybow\atifa"
  time="2004-06-11T16:25:07.7570000+02:00"
```

¹⁵ The **StatusCode** property on the **Response** object is unreliable and unfortunately the exact status code that is returned to the client can never be fully determined.

¹⁶ Values that were simply too long to display have been replaced with ellipsis.

```

statusCode="500"
webHostHtmlMessage="...">
<serverVariables>
  <item name="ALL_HTTP">
    <value string="..." />
  </item>
  <item name="ALL_RAW">
    <value string="..." />
  </item>
  <item name="APPL_MD_PATH">
    <value string="/LM/W3SVC/1/Root/ElmahDemo" />
  </item>
  <item name="APPL_PHYSICAL_PATH">
    <value string="C:\Demo" />
  </item>
  <item name="AUTH_TYPE">
    <value string="Negotiate" />
  </item>
  <item name="AUTH_USER">
    <value string="skybow\atifa" />
  </item>
  <item name="AUTH_PASSWORD">
    <value string="" />
  </item>
  <item name="LOGON_USER">
    <value string="SKYBOW\atifa" />
  </item>
  <item name="REMOTE_USER">
    <value string="skybow\atifa" />
  </item>
  <item name="CONTENT_LENGTH">
    <value string="0" />
  </item>
  <item name="CONTENT_TYPE">
    <value string="" />
  </item>
  <!-- rest removed for brevity -->
</serverVariables>
</error>

```

Note that the **ToXml** and **FromXml** methods come from my own interface, namely **IXmlExportable**. The reason for doing this is threefold. First, **IXmlSerializable** is officially undocumented in .NET Framework 1.x. It becomes documented with Whidbey so this argument does not hold so strong. Second, I didn't want to implement **GetSchema**. Third, I read and write the XML slightly differently. While **IXmlSerializable** places an extra container element around the object's XML, I didn't want it because it felt unnecessary. In any case, **IXmlSerializable** can always be implemented later in terms of **IXmlExportable** so this is not a big deal.

Note that when an **Error** object is constructed from an **Exception** instance, it also maintains a reference to it. This is done only for providing some fidelity at runtime. It has no consequence on logging or the XML serialization. Right now, the sample does not do any filtering based on exception types. However, if you ever wanted to add such a facility, then that's where the

Exception property would come in handy. For example, given an **Error** object, you could find out if it is of the class **ArgumentException** or not by inspecting its **Exception** property. You'll notice that when the **Error** object is re-created from the log for displaying purposes, the **Exception** property is always **null**.

ErrorLog and ErrorLogEntry

The **ErrorLog** class is the abstract contract for a logging store. The log is responsible for recording instances of **Error** and allowing them to be read back. To create a concrete implementation, one has to provide only 3 methods: **Log**, **GetErrors** and **GetError**. The **Log** method takes an **Error** object and is supposed to serialize it, however it wishes, to its backing store. The **GetErrors** method retrieves logged errors in paged result-sets. That is, you specify the page index and the count of the errors to retrieve with each call. It is supposed to read the log and return errors in their most natural order, which is defined to be from latest to earliest. Put another way, sorted on time in descending order. **GetErrors** is only required to retrieve a digest version of the full error that is well-suited for quick on-screen and summary listing as shown in Figure 2. The digest (light-weight) properties are defined as **ApplicationName**, **HostName**, **Type**, **Source**, **Message**, **User**, **StatusCode** and **Time**. However, a caller should be prepared to handle empty values for these properties. Finally, the **GetError** method retrieves an error in its entirety given its ID.

Both “get” methods actually return **ErrorLogEntry** instances rather than the **Error** object directly. The purpose of the **ErrorLogEntry** is to bind together the log-supplied data such as the ID with the actual **Error** object. Note that the **Id** property is typed as a string but the log can internally use an integer, GUID or what have you. The only requirement is that an **ErrorLog** implementation can round-trip its ID through a string.

An **ErrorLog** implementation is expected to store and retrieve errors bound to an application name¹⁷ and accounts for the **ApplicationName** property on **Error**. The reasons for this will become clearer as we move along.

¹⁷ Note that there is no method for retrieving the application names registered in a log. This is somewhat intentional for basic security reasons. Imagine a hosting provider who offers the error log facility to all the web applications. If there were a method like **GetApplications** on **ErrorLog**, then anyone could call

SqlErrorLog

SqlErrorLog is an implementation of **LogError** for Microsoft SQL Server 2000. The implementation is fairly straight forward since most of the code has to do with calling the stored procedures in the database. No direct table access is ever done.

The database has a single table called **Error**:

Column	Type	Purpose
ErrorId	UNIQUEIDENTIFIER	The unique ID of the logged error
Application	NVARCHAR(60)	Stores the ApplicationName property of the Error object.
Host	NVARCHAR(50)	Stores the HostName property of the Error object.
Type	NVARCHAR(100)	Stores the Type property of the Error object.
Source	NVARCHAR(60)	Stores the Source property of the Error object.
Message	NVARCHAR(500)	Stores the Message property of the Error object.
User	NVARCHAR(50)	Stores the User property of the Error object.
StatusCode	INT	Stores the StatusCode property of the Error object.
TimeUtc	DATETIME	Stores the Time property of the Error object in UTC.
Sequence	INT	This is an identity column that is solely used for the purpose of recording the sequence in which the errors were inserted into the log. This helps for sorting.
AllXml	NTEXT	The entire serialized XML of the Error object, acquired by calling the WriteXml method.

ErrorLog.Default.GetApplications and discover information private to others. This does not only apply to a hosting provider scenario. You can extend the same issue to an enterprise operations environment where web servers host more than one departmental business application. I imagine that this sort of issue can be solved with Code Access Security, but again, that's been omitted for sake brevity and keeping the solution focused.

Apart from **ErrorId** and **Sequence**, an **Error** object is split and stored in **AllXml** and the remaining columns. The **AllXml** column contains the full XML of **Error** object as it is serialized by its **ToXml** method. Because the **Error** object contains complex and hierarchical state like collections of named values, it seemed simplest to just store it as one blob item of type **NTEXT**. But why have the other columns? Well, remember that the **Log** needs to return a digest version of the error when the **GetErrors** method is called. It would be too expensive to de-serialize the entire object from XML only to return a piece of it. Consequently, the lightweight properties are cached away in separate columns for quicker retrieval. What's more, having properties like **Type** and **User** readily available allows you to quickly run statistics on the table. This is not done in the solution, but you could imagine where this would be useful in a reporting extension.

The downside of this approach, of course, is that if you update values in the cached columns, then they'll be out of sync with the corresponding values stored in the **AllXml** column (and vice versa). The differences will show up in the **Error** objects returned by the **GetErrors** and **GetError** method. Again, you are more than welcome to change the implementation details of **SqlErrorLog** to your liking. The rest of the solution will not be affected.

There's actually a much more subtle reason for having this kind of approach. Notice that the **Error** object is not sealed. Theoretically this means that it is extensible and you can go ahead and add your own properties to be stored in the log. However, for the log implementation to be oblivious to the final type it stores and serves, it simply records the XML of the object. The **FromXml** and **ToXml** methods on the **Error** class are extensible for precisely this reason. My implementation of **SqlErrorLog**, however, does not store the type of the **Error** object anywhere, so if you create your own **MyError** class, it does not know about it. It will be able to log it with all the state, but upon retrieval, you will always get back **Error** instances. One of my design goals was to keep type identity information out of the logs by as much as possible. Therefore **SqlErrorLog** offers a protected member called **NewError** that serves as an overridable factory. If you subclass **Error**, then you also have to provide a subclass for **SqlErrorLog** that can serve instances of your class.

The **LogError** method of **SqlErrorLog** is the one that splits up the **Error** object's properties into the cached set and the all encompassing XML column. The values are then passed as parameters to the **ELMAH_LogError** stored procedure that does the actual **INSERT**. There's

nothing else that is specially going on over here. It is worth noting that errors are never deleted in the default implementation. The exact policy of when and how to delete the errors can be enforced in the **ELMAH_LogError** stored procedure and right after the **INSERT** statement.

The **GetError** method of **SqlErrorLog** calls the related **ELMAH_GetErrorXml** stored procedure to get a single error identified by a GUID¹⁸. The stored procedure does nothing more than return the value of **AllXml** column from the corresponding row. On the way back, **GetError** simply re-creates an **Error** object by sending the XML to its **FromXml** method.

The **GetError** method of **SqlErrorLog** calls the related **ELMAH_GetErrorsXml** stored procedure, which returns a page of errors as XML. However, this time the XML only contains the digest properties mentioned earlier. **ExecuteXmlReader** is used to dig through the data and again **ReadXml** is used to populate **Error** objects that are eventually returned in a list. The use of XML in this case is just to have a lazy implementation that leverages the de-serialization infrastructure already present in the **Error** class. One could have easily used a data reader to read the data in a standard fashion and manually populate **Error** instances. Use of **ExecuteXmlReader** and the **FOR AUTO XML** in the stored procedure is really the reason for requiring SQL Server 2000, but this can be changed easily.

MemoryErrorLog

MemoryErrorLog is an implementation of **ErrorLog** that purely uses memory as its backing store. Needless to say, this log does not survive application restarts or failures, but it provides a very simple implementation that can come in handy in some hard cases where even **SqlErrorLog** would fail. For example, **SqlErrorLog** requires a complex store like a database and this assumes that an entire chain of infrastructure (network, server, database, etc.) will be fully operation in order to log exceptions in the first place. But say that even your error logging database becomes unavailable in the face of some catastrophic failure. This is where you could change your configuration file and temporarily switch to the **MemoryErrorLog** implementation to be able to diagnose problems albeit a volatile backing store.

¹⁸ The reason for choosing a GUID is rather important. If you want to ship error logs from several machines to a single server then using a GUID as the primary key and identification of an error provides the most conflict-free solution. Also, each error gets its own unique ID “forever.”

The implementation of the **MemoryErrorLog** is fairly straightforward. It uses a static instance of a nested collection implementation to store the error entries. The static instance is, of course, bound to the application domain so it will only store and serve errors private to the application. You can initialize the log with a size parameter that specifies the maximum number of entries it will log in its store. Once the log is filled up, the older entry is dropped to make place for a new one. The default size of the log is 15, with a maximum of 500. The default should be fine for most cases, but don't make it too big since errors will accumulate and consume memory unnecessarily. Remember, the point of this log implementation is for troubleshooting or even otherwise testing purposes.

There are only two items to mention with this implementation. First, it uses a **ReaderWriterLock** instance to synchronize access to the log. A writer-lock is acquired during the **Log** method and reader-lock during the **GetError** and **GetErrors** methods. The lock itself is a static member of the **MemoryErrorLog** and is initialized together with the type. The entries collection is itself initialized when the first error is logged. Until then, it has minimum footprint.

The second item to note is that the **MemoryErrorLog** makes defensive copies of the **Error** objects on the way in and out of its store (the internal collection). Since **Error** objects are mutable, the log clones an **Error** object in the **Log** method to maintain its own private copy and then returns yet another clone in **GetError** and **GetErrors** for the private use of the caller. I could have avoided all this extra copying by making the **Error** object immutable in the first place, but I decide to keep things short and simple for the article. From a more commercial-grade solution, it would be very favorable to support immutability on **Error** and its collections especially because it's really not all that hard to do.

Binding to the Log Implementation

So how does the **ErrorLogModule**, or anyone for that matter, know which concrete implementation of the log to use? The **ErrorLog** class has a **Default** property that provides, well, the default error log implementation configured for the application. The implementation of the **Default** property internally calls the **CreateFromConfigSection** method of the **SimpleServiceProviderFactory** to obtain the **ErrorLog** instance. The job of **SimpleServiceProviderFactory** is to create and configure a type from the configuration at a

specified configuration section. **SimpleServiceProviderFactory** makes three basic assumptions to succeed¹⁹:

- The configuration returned by [ConfigurationSettings.GetConfig](#) will be a dictionary, allowing you to use a number of the [configuration section handlers supplied with the base class library](#).
- The dictionary must have a “type” entry whose value supplies the [standard type-specification](#) of the object to instantiate.
- The type has a constructor that takes an [IDictionary](#) as the parameter.

The type is created using [Activator.CreateInstance](#) and via a constructor that is expected to take a single parameter typed as **IDictionary**. The dictionary can then be used by the type of initialize itself. For example, **SqlErrorLog** expects to find its connection string in there whereas the **MemoryErrorLog** looks for a “size” override.

Note that **SimpleServiceProviderFactory** removes the “type” key from the dictionary before passing it on to the type its constructing. However, it cannot do this simply on the dictionary returned by [GetConfig](#) because this is cached away by the framework and we don’t want to be modifying that version²⁰. So, instead, the dictionary is cloned and then the “type” key is removed. This defensive copy turns out to be good idea anyhow because the **SimpleServiceProviderFactory** cannot assume what the type is going to do with it anyhow.

So getting back to where it all started, the **Default** property on **ErrorLog** grabs the implementation from the configuration and hangs on to it in a static member. This means that subsequent calls to return the default instance will be instantaneous. One item of interest to note here is the static member is bound to the thread and not the application domain. In other words, an independent error log instance is maintained per thread. This yields two major benefits without writing any code and yet at the expense of very little memory overhead. We don’t need to synchronize access to the static field and, even more importantly, the downstream error log

¹⁹ These assumptions were one of those design decisions that were made to reduce the number of classes and abstractions needed in the solution. Some people may not agree with them, but they amply serve the purpose for the sample.

²⁰ Ideally, the objects returned from [GetConfig](#) would be read-only. Alas, this is not the case with the dictionaries returned by factory implementations such as [SingleTagSectionHandler](#).

implementation can be free of multi-threaded details. This is a lot like how ASP.NET also makes life easy for implementers of modules.

The Handlers to the ErrorLog

There's really nothing special going on in the handlers. Most of the code is basic practice for writing handlers and it's just rendering à la [HtmlTextWriter](#). The only interesting point to mention is that, except for the RSS handler, all handlers are in fact "pages." That is, they eventually inherit from the same [Page](#) class (via the [ErrorPageBase](#)) that you're used to. This provides all the benefits of a regular Web Form (ASPX page) like view state, validators and web controls so you don't necessarily have to resort to the [HtmlTextWrite](#)-style of rendering. The only thing you won't get is the dynamic compilation and the convenience of an HTML designer. I haven't employed any web controls in the solution because I didn't need anything as rich as the [DataGrid](#) or [Calendar](#). However, inheriting from [Page](#) still makes some things just more accessible as you don't have to pass around the [HttpContext](#) object supplied to [ProcessRequest](#).

The [ErrorPageBase](#) class serves as the base class for all the HTML content handlers. It basically provides some base convenience properties and a frame for the page layout as illustrated in Figure 14:

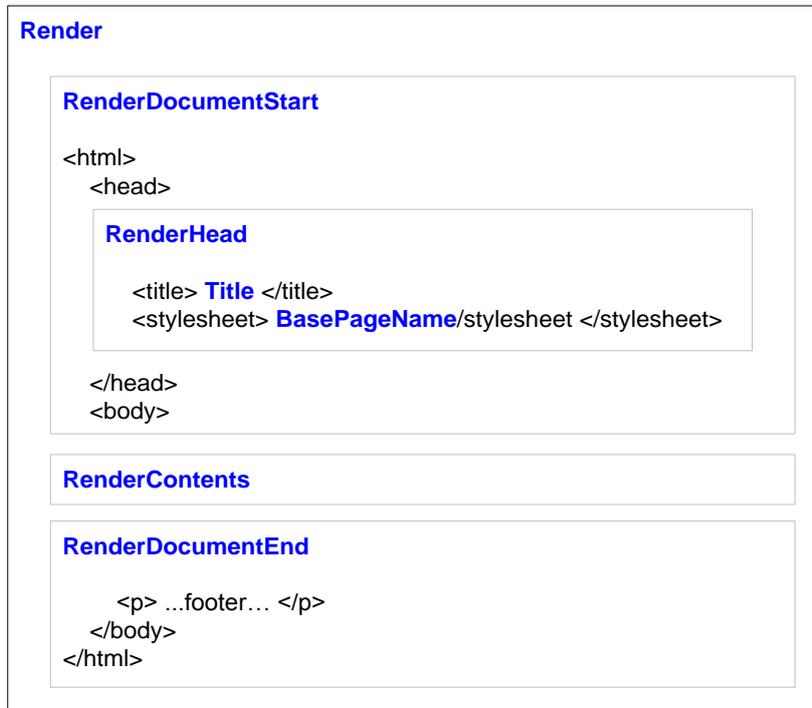


Figure 14

Note that style sheet for all pages comes from a CSS file that is as a resource in the assembly manifest.

The **ErrorLogPageFactory** is just a plain “switch” factory that is responsible for cracking the URL and returning the corresponding handler class. The factory uses the [PathInfo](#)²¹ property of [HttpRequest](#) to determine which resource is being requested.

Note how the style sheet for all pages is embedded as a resource and served using the **MainfestResourceHandler** class.

A lot of handlers provide no form of customization. So what do you do if you don’t like how the pages are laid out? Perhaps you want to put in there some extra detail or make them more compatible for some browser. Write your own handler to view the **ErrorLog**. It’s so easy that creating fully extensible and configurable handlers, plus the design decisions that entail them, is simply not worth the effort.

²¹ I used the [PathInfo](#) because it really keeps it leave query string clean for use by the downstream handler.

ErrorMailModule

The **ErrorMailModule** has a very simple implementation. During the **Init** method, it reads the configuration section to get some basic settings like the sender address, the recipient(s)²², the subject line and whether to send the mail synchronously or asynchronously. Like **ErrorLogModule**, it subscribes to the **Error** event of the application.

When an unhandled error propagates to the top, the **OnError** handler of the module gets called by ASP.NET. There an **Error** object is constructed from the **Exception** and then either **ReportErrorAsync** or **ReportError** is called depending on the **async** setting from the configuration. The idea of reporting the error asynchronously is simply to prevent an operational issue from delaying the response to the user unnecessarily²³. Asynchronous reporting is achieved [by posting a worker item to the thread pool](#). It may have been better not to borrow a thread from the same pool used by ASP.NET to server requests but this detail can be changed by a subclass easily²⁴. For the article, it illustrates the point adequately.

The workhorse of the implementation is in the **ReportError** method. This is where basically an e-mail is created, formatted and dispatched. A **MailMessage** object is created to specify the sender, recipients, format and body of the e-mail. The module is designed to focus on its job of preparing and sending a message, but the details how the body of the message is formatted with details from the **Error** object are isolated into a separate type, the **ErrorTextFormatter**.

The **ErrorTextFormatter** is actually an abstract base class that defines the contract that the error formatters must implement. It has a single property and method, namely **MimeType** and **Format**. The **MimeType** property of a mail formatter is used to set the **MailFormat** property of the **MailMessage**. **ErrorMailModule** only understands “text/plain” and “text/html”, both of which directly translate to **MailFormat.Text** and **MailFormat.Html**. The **Format** method is where the actual writing of the mail body takes place. It receives a **TextWriter** and an **Error** object as parameters. How the formatter then writes out the body is completely oblivious to the

²² This can be a semicolon-delimited list of e-mail addresses to reach several recipients.

²³ The same idea could have been used for the logging subsystem but I decided to demonstrate the idea in this module instead since it is less overloaded with implementation details.

²⁴ It's actually more important to check whether asynchronous reporting in the case of mailing buys you a lot before committing to a complicated implementation.

ErrorMailModule. To obtain the concrete error formatter implementation, the **ErrorMailModule** calls a protected virtual method named **CreateErrorFormatter**. In the supplied solution, there's only the **ErrorMailHtmlFormatter** implementation provided and which is returned from this method. It is the one responsible for the formatting the HTML mail as shown in Figure 1.

Just before sending out the mail, the **ErrorMailModule** calls **PreSendMail** to give subclasses a last crack at the mail to be sent and the error object²⁵. The default implementation checks if the **Error** object's **WebHostHtmlMessage** has a value or not. If it does, it creates an attachment and blasts the HTML contents into it. Finally, the **SendMail** method gets called and whose implementation forwards the call to **SmtpMail.Send** from the base class library.

The main thing to note about the implementation of the **ErrorMailModule** class is that it uses approach of the [Template Method design pattern](#). Rather than adding lots of configuration options, I've provided the workhorse of the implementation. For all customization, you can override various protected virtual members to change things that you don't like. Here's a summary:

Method	Why Override?
Init	Override this method to principally change how the class is initialized.
MailSender	If you override the Init method completely ²⁶ then this property still allows the remaining of the class to get access to the sender address.
MailRecipient	If you override the Init method completely then this property still allows the remaining of the class to get access to the recipient address(es).

²⁵ This is a little bit like how [PreRender](#) event is raised before the rendering phase begins for ASP.NET controls and pages.

²⁶ That is, without calling the base class implementation. The **ErrorMailModule** itself never relies on its private fields, but rather uses protected virtual properties that can be overridden. The only exception is the [_reportAsynchronously](#) field. This one has no corresponding property because that behavior can be overridden via **OnError** or **ReportErrorAsync**. For example, if you want to completely disable the feature, then just override **ReportErrorAsyc** to call **ReportError**.

MailSubjectFormat	If you override the Init method completely then this property still allows the remaining of the class to get access to the subject format.
OnError	Override this method to principally change how the Error object is obtained or how ReportError is invoked. This could also be a good place to add filtering based on the type of exception, like don't report HttpCompileException type of exceptions.
ReportErrorAsync	Override this method to principally change how ReportError is called asynchronously. Default implementation uses a worker from the system-supplied thread pool.
ReportError	Override this method to principally change the implementation of how an error is reported.
PreSendMail	Override this method if all you want to do is get a last crack at the mail message and error object before the e-mail is dispatched.
DisposeMail	Override this method to do any clean-up associated with the mail. The default implementation deletes attached files, assuming that they were created only for the temporary purpose of sending the mail.
CreateErrorFormatter	Override this method to principally return your own implementation of the ErrorTextFormatter .
SendMail	Override this method to principally change how the mail is really sent. If you want to use your own SMTP mailing library, for example, then this would be the right point to do the conversion.
GetConfig	Override this method to principally change how the configuration is obtained. For example, you could change the name of the configuration section used over here.
GetLastError	Override this method to principally change how the Exception is obtained and converted to an Error object.