Topeka
Capitol of Kansas
Error Handling with PHP (part I)
George washington
First President

By icarus

Table of Contents

Flame On

No developer, no matter how good he or she may be, writes error−free code all the time. Which is why most programming languages − including PHP − come with built−in capabilities to catch errors and take remedial action. This action could be something as simple as displaying an error message, or as complex as heating your computer's innards until they burst into flame (just kidding!)

Now, you might not know this, but PHP comes with a full−featured error handling API, which provides you with a number of options when it comes to trapping and resolving errors. Not only does PHP allow you to decide which types of errors get displayed to the user, but it also allows you to bypass its internal error handler in favour of your own custom functions, thereby opening up all sorts of possibilities for the creative developer.

My job over the next few pages is to educate you about these possibilities, first by offering some grounding in the fundamentals and then by showing you how PHP's error−handling functions can be used in real−world development. So keep reading.

Rogues Gallery

Before we get into the nitty−gritty of how to write an error handler, you need to know a little theory.

Normally, when a PHP script encounters an error, it displays a message indicating the cause of the error and (depending on how serious the error was) terminates script execution at that point. Now, while this behaviour is acceptable during the development phase, it cannot continue once a PHP application has been released to actual users. In these "live" situations, it is unprofessional to display cryptic error messages (which are usually incomprehensible to non−technical users); rather, it is more professional to intercept these errors and either resolve them (if resolution is possible), or notify the user with a clear error message (if not).

There are three basic types of errors in PHP:

1. Notices: These are trivial, non−critical errors that PHP encounters while executing a script − for example, accessing a previously−undefined variable. By default, such errors are not displayed to the user (although, as you will see, you can change this default behaviour).

2. Warnings: These are relatively more serious errors − for example, attempting to include() a file which does not exist. These errors are displayed to the user, but they do not result in script termination.

3. Fatal errors: These are critical errors − for example, instantiating an object of a non−existent class, or calling a function which does not exist. These errors cause the immediate termination of script execution, and are also displayed to the user when they take place.

It should be noted that a syntax error in a PHP script − for example, a missing brace or semi−colon − is treated as a fatal error and results in script termination.

Now, these three types of errors are generated by different entities within the PHP engine. They may be generated by the core, by PHP functions built on top of this core, or by the application. They may be generated at startup, at parse−time, at compile−time or at runtime. And so, PHP defines eleven different error types, each of which is identified by both a named constant and an integer value. Here they are:

```
Constant Value Description
----------------------------------------------
E_ERROR 1 Fatal runtime error

E_WARNING 2 Non−fatal runtime error

E_PARSE 4 Runtime parse error

E_NOTICE 8 Non−fatal runtime notice

E_CORE_ERROR 16 Fatal startup error

E_CORE_WARNING 32 Non−fatal startup error

E_COMPILE_ERROR 64 Fatal compile−time error
```

```
E_COMPILE_WARNING 128 Non−fatal compile−time error

E_USER_ERROR 256 User−triggered fatal error

E_USER_WARNING 512 User−triggered non−fatal error

E_USER_NOTICE 1024 User−triggered notice

E_ALL All of the above
```

Most of the time, you're not going to worry about all eleven types; your focus will usually be on the runtime errors (E_NOTICE, E_WARNING and E_ERROR) and the user−triggered errors (E_USER_NOTICE, E_USER_WARNING and E_USER_ERROR), and your error−handling code will need to gracefully resolve these error types.

The named constants and integer values provide a convenient way to reference the different error types. It's possible, for example, to combine the integer values together into a bitmask to represent a specific combination of error types. You'll see what I mean on the next page.

Mary, Mary, Quite Contrary

With the theory out of the way, let's now apply it to some examples. Consider the following code snippet:

```
<?php
// string
$string = "Mary had a little lamb";

// attempt to join() this string
// this will generate an E_WARNING
// because the second argument to join() must be an array
join('', $string); ?>
```

Now, if you were to run this script, you'd see the following error:

```
Warning: Bad arguments to join() in
/usr/local/apache/htdocs/e.php on
line 8
```

Now, this is a non−fatal error, which means that if I had statements following the call to join(), they would still get executed. For example, the code snippet

```
<?php
// string
$string = "Mary had a little lamb";

// attempt to join() this string
// this will generate an E_WARNING
// because the second argument to join() must be an array
join('', $string);

// since this is a non−fatal error
// this statement should be executed
echo "-- end --";
?>
```

produces

```
Warning: Bad arguments to join() in
/usr/local/apache/htdocs/e.php on
line 8
```

```
-- end --
```

Want to see what a fatal error looks like? Try this:

```
<?php
// say something
echo "-- begin --";

// call a non-existent function
// this will generate an E_ERROR
someFunction();

// this statement will never be executed
echo "-- end --";
?>
```

Here's the output:

```
-- begin --
Fatal error: Call to undefined function: somefunction() in
/usr/local/apache/htdocs/e.php on line 7
```

PHP allows you to control error reporting via its − you guessed it − error_reporting() function. This function accepts either an integer (in the form of a bitmask) or named constant, and tells the script to only report errors of the specified type(s).

Consider the following example, which reworks one of the preceding code snippets to "hide" non−fatal errors:

```
<?php
// set error-reporting to only fatal errors
error_reporting(E_ERROR);

// string
$string = "Mary had a little lamb";

// attempt to join() this string
// this will generate an E_WARNING
// because the second argument to join() must be an array
join('',
$string);

// since this is a non-fatal error
// this statement should be executed
```

```
echo "-- end --";
?>
```

In this case, when the script executes, no warning will be generated even though the second argument to join() is invalid.

You can also turn off the display of fatal errors using this technique.

```
<?php
// no errors will be reported
error_reporting(0);

// say something
echo "-- begin --";

// call a non-existent function
// this will generate an E_ERROR
someFunction();

// this statement will never be executed
echo "-- end --";
?>
```

It should be noted, however, that this approach, although extremely simple, is *not* recommended for general use. It is poor programming practice to trap all errors, regardless of type, and ignore them; it is far better − and more professional − to anticipate the likely errors ahead of time, and write code to isolate and resolve them.

Bit By Bit

The argument provided to error_reporting() may be either an integer or a named constant. PHP's bitwise operators can be used to create different combinations of error types. Here are a few examples:

```
<?php
// no errors will be reported
error_reporting(0);

// all errors will be reported
error_reporting(2047);

// this is equivalent to
error_reporting(E_ALL);

// only report E_ERROR, E_WARNING and E_NOTICE errors
error_reporting(11);

// this is equivalent to
error_reporting(E_ERROR | E_WARNING | E_NOTICE);

// report all errors except E_USER_NOTICE
error_reporting(E_ALL & ~E_USER_NOTICE);
?>
```

By default, PHP 4.x is configured to report all errors except E_NOTICE errors; however, this can be altered by editing the PHP configuration file.

As an aside, remember that it is also possible to selectively turn off error display by prefixing function calls with the @ operator. For example, though the code snippet

```
<?php
// call a non-existent function
someFunction();
?>
```

would normally generate a fatal error (because someFunction() doesn't exist), this error could be suppressed by placing an @ symbol before the function call, like this:

```
<?php
// call a non-existent function
@someFunction();
?>
```

A call to error_reporting() without any arguments returns the current reporting level. So the code snippet

```
<?php
echo error_reporting();
?>
```

would output

```
2039
```

A Custom Job

By default, errors are handled by PHP's built−in error handler, which identifies the error type and then displays an appropriate message. This message indicates the error type, the error message, and the file name and line number where the error was generated. Here's an example of one such error message:

```
Warning: Failed opening 'common.php' for inclusion
(include_path='.') in
/usr/local/apache/htdocs/e.php on line 4
```

If the error generated is a fatal error, PHP will display an error message

```
Fatal error: Failed opening required 'common.php'
(include_path='.;') in
/usr/local/apache/htdocs/e.php on line 4
```

and terminate script execution at that point itself.

Now, this default behaviour is all well and good for small, uncomplicated scripts. However, if you're building a complex Web application, this behaviour is probably too primitive for your needs. What you would probably like is a way to bypass PHP's error−handling mechanism and handle errors directly, in a way that is best suited to the application you are developing.

Enter the set_error_handler() function, which allows you to define your own custom error handler for a script.

The set_error_handler() function accepts a single string argument, the name of the function to be called whenever an error occurs. This user−defined function must be capable of accepting a minimum of two arguments − the error type and corresponding descriptive message − and up to three additional arguments − the file name and line number where the error occurred, and a snapshot of the variable set at the time of error.

The following example might make this clearer:

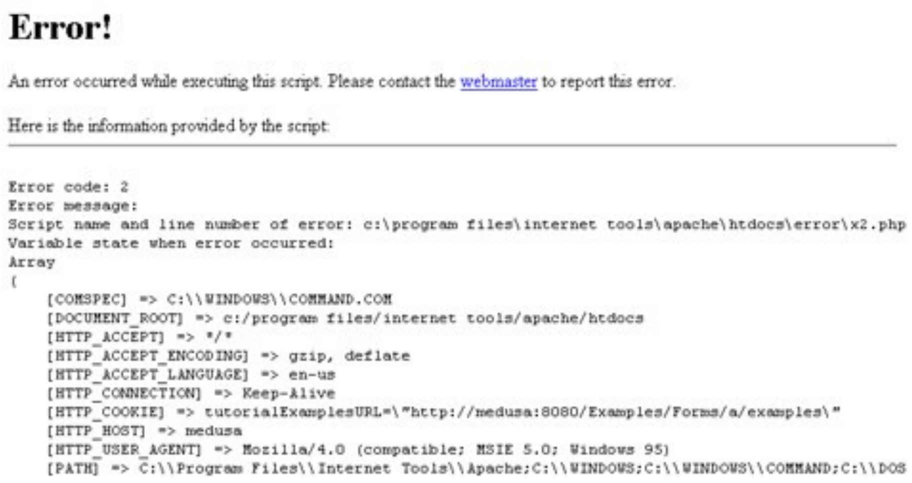
```
<?php
// custom error handler
function eh($type, $msg, $file, $line, $context)
{
echo "<h1>Error!</h1>";
echo "An error occurred while executing this script. Please
contact the <a
href=mailto:webmaster@somedomain.com>webmaster</a> to
report this error.";
echo "<p>";
echo "Here is the information provided by the script:";
echo "<hr><pre>";
echo "Error code: $type<br>";
echo "Error message: $message<br>";
echo "Script name and line number of error: $file:$line<br>";
echo "Variable state when error occurred: <br>";
print_r($context);
echo "</pre><hr>";
}

// define a custom error handler
set_error_handler("eh");

// string
$string = "Mary had a little lamb";

// this will generate an E_WARNING
join('', $string);
?>
```

Here's what it looks like:



Error!

An error occurred while executing this script. Please contact the webmaster to report this error.

Here is the information provided by the script:

```
Error code: 2
Error message:
Script name and line number of error: c:\program files\internet tools\apache\htdocs\error\x2.php
Variable state when error occurred:
Array
(
    [COMSPEC] => C:\\WINDOWS\\COMMAND.COM
    [DOCUMENT_ROOT] => c:/program files/internet tools/apache/htdocs
    [HTTP_ACCEPT] => */*
    [HTTP_ACCEPT_ENCODING] => gzip, deflate
    [HTTP_ACCEPT_LANGUAGE] => en-us
    [HTTP_CONNECTION] => Keep-Alive
    [HTTP_COOKIE] => tutorialExamplesURL=\"http://medusa:8080/Examples/Forms/a/examples\"
    [HTTP_HOST] => medusa
    [HTTP_USER_AGENT] => Mozilla/4.0 (compatible; MSIE 5.0; Windows 95)
    [PATH] => C:\\Program Files\\Internet Tools\\Apache;C:\\WINDOWS;C:\\WINDOWS\\COMMAND;C:\\DOS
```

In this case, my first step is to override PHP's default error handler via a call to set_error_handler(). This function tells the script that all errors are to be routed to my user−defined eh() function.

If you take a close look at this function, you'll see that it's set up to accept five arguments: the error type, message, file name, line number and an array containing the current set of variables and values. These arguments are then used internally to create an error page that is friendlier and more informative than PHP's one−line error messages.

Since the custom error handler is completely defined user−defined, a common technique involves altering the error message on the basis of the error type. Take a look at the next example, which demonstrates this technique:

```
<?php

// custom error handler
function eh($type, $msg, $file, $line, $context)
{
switch($type)
{
// notice
case E_NOTICE:
// do nothing
break;

// warning
case E_WARNING:
// report error
echo "A non-fatal error occurred at line $line
of file $file. The error message was $msg";
break;

// fatal
case E_ERROR:
// report error and die()
die("A fatal error occurred at line $line of
file $file. The error message was $msg");
break;
}
}

// define a custom error handler
set_error_handler("eh");

// string
$string = "Mary had a little lamb";

// this will generate an E_WARNING
join('', $string);

?>
```

Under The Microscope

When a custom error handler is defined with set_error_handler(), the relationship between error_reporting() and set_error_handling() bears examination. When both these functions coexist within the same script, PHP will assume that all error types defined in the error_reporting() function call, with the exception of the E_ERROR and E_PARSE types and the E_COMPILE and E_CORE families, will be handled by the custom error handler.

That might not make too much sense, but the next few examples should help to make it clearer.

```
<?php

// custom error handler
function eh($type, $msg, $file, $line, $context)
{
switch($type)
{
case E_ERROR:
die("A fatal error occurred at line $line of
file $file. The error message was <b>$msg</b> <br>");
break;

case E_WARNING:
echo "A non-trivial, non-fatal error occurred at
line $line of file $file. The error message was <b>$msg</b>
<br>";
break;

case E_NOTICE:
echo "A trivial, non-fatal error occurred at
line $line of file $file. The error message was <b>$msg</b>
<br>";
break;

}
}

// turn on all error reporting
error_reporting(E_ALL);

// define a custom error handler
set_error_handler("eh");

// this will generate two errors:
// E_NOTICE because $string is undefined
// E_WARNING because the second argument to join() is wrong
join('',
$string);

?>
```

In this case, even notices get reported, and handled by the custom handler. Here's what it looks like:

```
A trivial, non-fatal error occurred at line 32 of file
/usr/local/apache/htdocs/error/e.php. The error message was
Undefined
variable: string
A non-trivial, non-fatal error occurred at line 32 of file
/usr/local/apache/htdocs/error/e.php. The error message was
Bad
arguments to join()
```

Now, suppose I introduce a fatal error into the script above:

```
<?php

// custom error handler
function eh($type, $msg, $file, $line, $context)
{
switch($type)
{
case E_ERROR:
// this will actually never be used!
die("A fatal error occurred at line $line of
file $file. The error message was <b>$msg</b> <br>");
break;

case E_WARNING:
echo "A non-trivial, non-fatal error occurred at
line $line of file $file. The error message was <b>$msg</b>
<br>";
break;

case E_NOTICE:
echo "A trivial, non-fatal error occurred at
line $line of file $file. The error message was <b>$msg</b>
<br>";
break;

}

}
```

```
// turn on all error reporting
error_reporting(E_ALL);

// define a custom error handler
set_error_handler("eh");

// this will generate two errors:
// E_NOTICE because $string is undefined
// E_WARNING because the second argument to join() is wrong
join('',
$string);

// this will generate an E_ERROR
someFunction();
?>
```

Here's what happens when I run it:

```
A trivial, non−fatal error occurred at line 34 of file
/usr/local/apache/htdocs/e.php. The error message was
Undefined
variable: string
A non−trivial, non−fatal error occurred at line 34 of file
/usr/local/apache/htdocs/e.php. The error message was Bad
arguments to
join()

Fatal error: Call to undefined function: somefunction() in
/usr/local/apache/htdocs/e.php on line 37
```

In this case, even though I have an error handler defined for the fatal error type E_ERROR, it will not be used; instead, PHP's built−in handler will be called to handle the error. Which is why I said, right at the beginning, that E_ERROR and E_PARSE error types will be routed to the built−in handler regardless of what you set error_reporting() to.

No News Is Good News

This ability to catch and internally handle non−fatal script errors is particularly useful when building real−world PHP−driven Web sites. By defining a custom error handler for each PHP script, it becomes possible to trap unsightly error messages (well, at least the less serious ones) and handle them in a graceful manner.

In order to illustrate this, consider the following script, which builds a Web page dynamically from the data in a MySQL database. Note how a custom error handler has been used to catch warnings before they are displayed to the user, and write them to a log file instead.

```
<html>
<head><basefont face="Arial"></head>
<body>
<h2>News</h2>
<?php

// custom error handler
function e($type, $msg, $file, $line)
{
// read some environment variables
// these can be used to provide some additional debug
information
global $HTTP_HOST, $HTTP_USER_AGENT, $REMOTE_ADDR,
$REQUEST_URI;

// define the log file
$errorLog = "error.log";

// construct the error string
$errorString = "Date: " . date("d−m−Y H:i:s", mktime()) .
"\n";
$errorString .= "Error type: $type\n";
$errorString .= "Error message: $msg\n";
$errorString .= "Script: $file($line)\n";
$errorString .= "Host: $HTTP_HOST\n";
$errorString .= "Client: $HTTP_USER_AGENT\n";
$errorString .= "Client IP: $REMOTE_ADDR\n";
$errorString .= "Request URI: $REQUEST_URI\n\n";

// write the error string to the specified log file
$fp = fopen($errorLog, "a+");
fwrite($fp, $errorString);
fclose($fp);

// if you wanted to, you could do something else here
// − log errors to a database
// − mail() them
// − die()

// display error message
echo "<h1>Error!</h1>";
echo "We're sorry, but this page could not be displayed
because
of an internal error. The error has been recorded and will be
rectified
as soon as possible. Our apologies for the inconvenience. <p>
<a
href=/>Click here to go back to the main menu.</a>";

}

// report warnings and fatal errors
error_reporting(E_ERROR | E_WARNING);

// define a custom handler
set_error_handler("e");

// attempt a MySQL connection
$connection = @mysql_connect("localhost", "john", "doe");
mysql_select_db("content");

// generate and execute query
$query = "SELECT * FROM news ORDER BY timestamp DESC";
$result = mysql_query($query, $connection);

// if resultset exists
if (mysql_num_rows($result) > 0)
{
?>

<ul>

<?php
// iterate through query results
// print data
while($row = mysql_fetch_object($result))
{
?>
<li><b><?=$row−>slug?></b>
<br>
<font size=−1><i><?=$row−>timestamp?></i></font>
<p>
<font size=−1><?php echo substr($row−>content, 0, 150); ?>...
<a
href=story.php?id=<?=$row−>id?>>Read more</a></font>
<p>
<?php
}
?>
</ul>
<?php
}
else
{
echo "No stories available at this time";
}
?>

</body>
</html>
```

In this case, all errors of type E_WARNING will be intercepted by the custom handler, and will be written to the specified log file via the fopen() function. The user will never see these errors; however, a webmaster could periodically review the log file to evaluate (and hopefully correct) the errors.

The example above writes the error message to a file, together with a bunch of other useful debugging information. This is, however, by no means your only option; since the custom error handler is completely under your control, you could just as easily write code to INSERT the error into a database, or email it to the webmaster via PHP's mail() function. I personally prefer the write−to−file option because it is simple, does not require the overhead of a database connection, and is less likely to cause an avalanche of email to the site administrator (you know how touchy those guys are!)

Note also that the script above is silent on the topic of fatal errors. If a fatal error occurs, it will still be displayed to the user, together with any other information PHP chooses to append to the error string.

Finally, in the event that a warning is generated while the script above is executing, the custom error handler will write the error message to the screen, regardless of what output has already been printed. Consequently, you might end up with a half−constructed Web page sporting an error message at the end. Needless to say, this is not a Good Thing.

Luckily for you, there is an alternative, more robust solution to the problem. That piece of code, together with a simpler way of logging errors and information on how to use the PHP error−handling API to generate your own custom errors, is all available in the second part of this article. Keep an eye out for that one − and, until then, go practice! Note: All examples in this article have been tested on Linux/i586 with Apache 1.3.20 and PHP 4.1.1. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!