# Error Handling with PHP (part II)

## By icarus

# Table of Contents

**The Road Ahead**.....1

**Raising Hell**.....2

**Rolling Back**.....8

**Turning Up The Heat**.....10

**Of Form And Function**.....13

**Buffer Zone**.....23

**Back To Class**.....26

**Endgame**.....28

# The Road Ahead

In the first part of this article, I introduced you to PHP's error−handling functions, demonstrating how they could be used to control the manner in which errors are handled. I showed you how to use the error reporting functions to filter out those error types you didn't want to see, and how to use a custom error handler to catch PHP's notices and warnings. Finally, I wrapped things up with a demonstration of how a custom error handler could be used to track errors on a Web site, and write those errors to a log file for later review.

There's a lot more you can do with PHP's error−handling API, though; what you've seen so far is just the tip of the iceberg. In this concluding section, I will demonstrate how you can piggyback your code on top of PHP's core engine by triggering your own errors and writing custom handlers to deal with them. I will also show you a simpler method of logging errors, and will (as usual) put the theory to the test with a couple of real−world examples. Much mayhem ahead...so don't go anywhere!

# Raising Hell

Thus far, we've been dealing with errors which are automatically generated by PHP when it encounters a problem. However, in addition to these built−in errors, PHP also allows you to raise errors of your own.

This is accomplished via a little function named trigger_error(), which allows you to raise any of the three errors in the E_USER family. These errors can then be caught and handled either by PHP's built−in error handler, or by a custom handler.

Here's a simple example which demonstrates how this works:

```
<?php

// function to validate password integrity
// generates an E_USER_WARNING if password fails a test
function
validatePassword($pass) {
// empty string
if(trim($pass) == "")
{
trigger_error("Empty password", E_USER_WARNING);
}

// too short
if(strlen($pass) < 5)
{
trigger_error("Password too short", E_USER_WARNING);
}

// only numeric
if(is_numeric($pass))
{
trigger_error("Password cannot contain only numbers",
E_USER_WARNING);
}
}

echo "<br>−− validating empty string −− <br>";

validatePassword("");

echo "<br>−− validating 12345 −− <br>";

validatePassword(12345);

echo "<br>−− validating Gh3 −−<br>";
```

```
validatePassword("Gh3");

?>
```

Here's what the output looks like:

```
-- validating empty string --

Warning: Empty password in /usr/local/apache/htdocs/x2.php on
line 10

Warning: Password too short in /usr/local/apache/htdocs/x2.php
on line
16

-- validating 12345 --

Warning: Password cannot contain only numbers in
/usr/local/apache/htdocs/x2.php on line 22

-- validating Gh3 --

Warning: Password too short in /usr/local/apache/htdocs/x2.php
on line
16
```

In this case, every time the argument to validatePassword() fails one of the tests within the function, an E_USER_WARNING error will be raised; this error will be caught by PHP's built−in handler and handled in the same way as "regular" warnings − it will be displayed to the user, but script processing will not be halted.

It's also possible to raise fatal errors in this fashion. Consider the next example, which updates the validatePassword() function to raise a fatal error only if the password string is empty.

```
<?php

// function to validate password integrity
// generates an E_USER_WARNING if password fails a test
function
validatePassword($pass) {
// empty string
// trigger a fatal error
if(trim($pass) == "")
{
trigger_error("Empty password", E_USER_ERROR);
}
```

```
// too short
if(strlen($pass) < 5)
{
trigger_error("Password too short", E_USER_WARNING);
}

// only numeric
if(is_numeric($pass))
{
trigger_error("Password cannot contain only numbers",
E_USER_WARNING);
}
}

echo "<br>-- validating 12345 -- <br>";

validatePassword(12345);

echo "<br>-- validating empty string -- <br>";

validatePassword(" ");

echo "<br>-- validating Gh3 --<br>";

validatePassword("Gh3");

?>
```

In this case, when the second password is evaluated, a fatal error will be raised, PHP's built−in handler will catch it, note that it is a fatal error and terminate script execution immediately. Here's what it looks like:

```
-- validating 12345 --

Warning: Password cannot contain only numbers in
/usr/local/apache/htdocs/x2.php on line 23

-- validating empty string --

Fatal error: Empty password in /usr/local/apache/htdocs/x2.php
on line
11
```

Note that the script never reaches the third call to validatePassword().

User−triggered errors can also be caught by a custom error handler, in much the same way as built−in errors.

Let's see how, with a variant of the example on the previous page:

```
<?php

// function to validate password integrity
// generates an E_USER_WARNING if password fails a test
function
validatePassword($pass) {
// empty string
// trigger a fatal error
if(trim($pass) == "")
{
trigger_error("Empty password", E_USER_ERROR);
}

// too short
if(strlen($pass) < 5)
{
trigger_error("Password too short", E_USER_WARNING);
}

// only numeric
if(is_numeric($pass))
{
trigger_error("Password cannot contain only numbers",
E_USER_WARNING);
}
}

// custom error handler
function eh($type, $msg, $file, $line, $context)
{
switch($type)
{
// user-triggered fatal error
case E_USER_ERROR:
echo "A fatal error occurred at line $line of
file $file. The error message was <b>$msg</b> <br>";
echo "<font color=red><i>Script
terminated</i></font>";
die();
break;

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

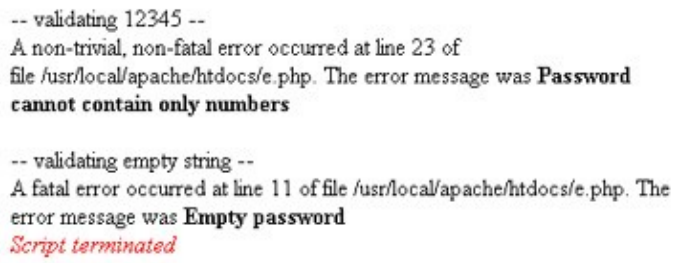
```
break;

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

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

echo "<br>-- validating 12345 -- <br>";

validatePassword(12345);

echo "<br>-- validating empty string -- <br>";

validatePassword(" ");

echo "<br>-- validating Gh3 --<br>";

validatePassword("Gh3");

?>
```

In this case, the user-generated errors are routed to the custom error handler, which prints a user-defined error message and - if the error was defined as fatal - terminates script execution.

Here's what it looks like:

```
-- validating 12345 --
A non-trivial, non-fatal error occurred at line 23 of file
/usr/local/apache/htdocs/x2.php. The error message was
Password cannot
contain only numbers

-- validating empty string --
A fatal error occurred at line 11 of file
/usr/local/apache/htdocs/x2.php. The error message was Empty
password
Script terminated
```

And here's a picture:




Note that it is the responsibility of the custom handler to die() in the event of user−generated fatal errors − PHP will not do this automatically.

# Rolling Back

PHP also comes with a restore_error_handler() function, which allows you to restore the previous error handler. This is useful if you need to switch between handlers in a single script. Consider the following simple example, which demonstrates:

```
<?php

// custom error handler
function eh($type, $msg, $file, $line, $context)
{
echo "This is the custom error handler speaking";
}

// trigger a warning
// this will be caught by the default handler
// since nothing else has been defined
trigger_error("Something bad happened", E_USER_WARNING);

// define a new handler
set_error_handler("eh");

// trigger another warning
// this will be caught by the custom handler
trigger_error("Something
bad happened", E_USER_WARNING);

// rollback to default handler
restore_error_handler();

// trigger another warning
// this will be caught by the default handler
trigger_error("Something
bad happened", E_USER_WARNING);

?>
```

Here's what the output looks like:

```
Warning: Something bad happened in
/usr/local/apache/htdocs/x2.php on
line 12 This is the custom error handler speaking
Warning: Something bad happened in
/usr/local/apache/htdocs/x2.php on
line 26
```

# Turning Up The Heat

In addition to catching and displaying errors, PHP's error−handling API also allows you to log errors, either to a default error log, to any other file or as an email message.

The error_log() function needs a minimum of two arguments: the error message to be logged, and an integer indicating where the message should be sent. There are three possible integer values in PHP 4.x:

0 − send the message to the system log file (note that you must have logging enabled and a log file specified in your PHP configuration for this to work);

1 − send the message to the specified email address;

3 − send the message to the specified file;

Here's a trivial example which demonstrates how this works:

```
<?php
// set a variable
$temp = 101.6;

// test it and log an error
// this will only work if
// you have "log_errors" and "error_log" set in your php.ini
file if
($temp > 98.6) {
error_log("Body temperature above normal.", 0);
}

?>
```

Now, if you look at the system log file after running the script, you'll see something like this:

```
[28−Feb−2002 15:50:49] Body temperature above normal.
```

You can also write the error message to a file,

```
<?php

// set a variable
$temp = 101.6;

// test it and log an error
```

```
if ($temp > 98.6)
{
error_log("Body temperature above normal.", 3, "a.out");
}

?>
```

or send it out as email.

```
<?php

// set a variable
$temp = 101.6;

// test it and log an error
if ($temp > 98.6)
{
error_log("Body temperature above normal.", 1,
"administrator@this.body.com"); }

?>
```

It's possible to combine this error logging facility with a custom error handler to ensure that all script errors get logged to a file. Here's an example which demonstrates this:

```
<?php

// custom handler
function eh($type, $msg, $file, $line)
{
// log all errors
error_log("$msg (error type $type)", 0);

// if fatal error, die()
if ($type == E_USER_ERROR)
{
die($msg);
}
}


// report all errors
error_reporting(E_ALL);

// define custom handler
```

```
set_error_handler("eh");

// let's go through the rogues gallery

// this will trigger E_NOTICE (undefined variable)
echo $someVar;

// this will trigger E_WARNING (missing file)
include("common.php");

// this will trigger E_USER_NOTICE
trigger_error("Time for lunch");

// this will trigger E_USER_WARNING
trigger_error("Body temperature above normal",
E_USER_WARNING);

// this will trigger E_USER_ERROR
trigger_error("No brain activity", E_USER_ERROR);
?>
```

And here's the output that gets logged to the system log file:

```
[28-Feb-2002 16:15:06] Undefined variable: someVar (error type
8)
[28-Feb-2002 16:15:06] Failed opening 'common.php' for
inclusion
(include_path='.;') (error type 2)
[28-Feb-2002 16:15:06] Time for lunch (error type 1024)
[28-Feb-2002
16:15:06] Body temperature above normal (error type 512)
[28-Feb-2002
16:15:06] No brain activity (error type 256)
```

# Of Form And Function

Finally, how about a real−life example to put all this in context? The following script sets up a simple HTML form, and then uses the error−handling functions just explained to catch and resolve form input errors once the form is submitted. Take a look:

```
<html>
<head><basefont face="Arial"></head>
<body>

<?php
if (!$submit)
{
?>
<h2>User Profile</h2>
<form action="<?=$PHP_SELF?>" method="POST">
<b>Name</b><br>
<input type="text" name="name" size="15"><br>

<b>Age</b><br>
<input type="text" name="age" size="2" maxlength="2"><p>

<b>Email address</b><br>
<input type="text" name="email" size="20"><p>

<b>Favourite pizza topping</b><br>
<select name="topping">
<option value="">-- select one --</option>
<option value="1">Pepperoni</option>
<option value="2">Pineapple</option>
<option value="3">Red peppers</option>
<option value="4">Raw fish</option>
</select><p>

<input type="submit" name="submit" value="Save">
</form>
<?php
}
else
{
// initialize an array to hold warnings
$warningList = array();

// array mapping error codes to messages
// add new elements as required
// this should ideally be stored in a separate file
// available to all scripts in an application
```

```
$errorCodes = array(
41 => "Invalid or incomplete data",
43 => "Invalid selection",
49 => "Incomplete form input",
55 => "No database connection available",
56 => "Selected database unavailable",
58 => "Error in SQL query"
);

// ideally, the next three functions should all
// be global functions, which can be read in where required

// function which accepts an error code
// and translates it into a human readable message
// it then raises an E_USER_WARNING
function raiseWarning($code, $info="")
{
global $errorCodes;

// use code to get corresponding error message
// from $errorCodes array
$msg = $errorCodes[$code];

// if additional info available
// append it to message
if ($info != "")
{
$msg .= " -> $info";
}

// raise an error
trigger_error($msg, E_USER_WARNING);
}

// function which accepts an error code
// and translates it into a human readable message
// it then raises an E_USER_ERROR
function raiseError($code, $info="")
{
global $errorCodes;
$msg = $errorCodes[$code];

if ($info != "")
{
$msg .= " -> $info";
}

trigger_error($msg, E_USER_ERROR);
}
```

```
// function to iterate through $warningsList
// and display warnings as bulleted list
function displayWarnings()
{
global $warningList;
if (sizeof($warningList) > 0)
{
echo "The following non-fatal errors occurred:";
echo "<ul>";
foreach ($warningList as $w)
{
echo "<li>$w";
}
echo "</ul>";
return true;
}
else
{
return false;
}
}

// function to validate email addresses
function is_valid_email_address($val)
{
if(trim($val) != "")
{
$pattern =
"/^([a-zA-Z0-9])+([\.a-zA-Z0-9_-])*@([a-zA-Z0-9_-])+(\.[a-zA-Z0-9_-]+)+/
";
if(preg_match($pattern, $val))
{
return true;
}
else
{
return false;
}
}
else
{
return false;
}
}

// custom error handler
function e($type, $msg)
{
```

```
global $warningList;

// log error
error_log($msg, 0);

switch($type)
{
// if warning, add message to $warningList array
case E_WARNING:
case E_USER_WARNING:
$warningList[] = $msg;
break;

// if error, die()
case E_USER_ERROR:
die("<b>The following fatal error
occurred: $msg</b>");
break;

// everything else
default:
break;
}
}

// functions end

// adjust this as per your needs
error_reporting(E_ERROR | E_WARNING | E_USER_ERROR |
E_USER_WARNING);

// define handler
set_error_handler("e");

// check form data
// raise warnings appropriately
if(!$name || trim($name) == "") { raiseWarning(41, "NAME"); }

if(!$age || trim($age) == "") { raiseWarning(41, "AGE"); }

if(!$email || trim($email) == "") { raiseWarning(41, "EMAIL
ADDRESS"); }

if (!is_numeric($age)) { raiseWarning(41, "AGE"); }

if (!is_valid_email_address($email)) { raiseWarning(41, "EMAIL
ADDRESS"); }

if (!$topping) { raiseWarning(43, "PIZZA TOPPING"); }
```

```
// show all warnings
// if warnings exist, it means that the form data
// is invalid/incomplete
// so raise a fatal error and don't go any further
if (displayWarnings()) { raiseError(49); }

// attempt a MySQL connection
$connection = @mysql_connect("localhost", "john", "doe") or
raiseError(55);

mysql_select_db("data") or raiseError(56);

$query = "INSERT INTO users(name, age, email, topping)
VALUES('$name', '$age', '$email', '$topping')";

// execute query to input form data
$result = mysql_query($query, $connection) or raiseError(58,
$query . " -> " . mysql_error());

// success!
echo "<h2>Thank you. Your profile was successfully
added.</h2>";
} ?> </body> </html>
```

This probably seems very complicated, so let me break it down for you. The first part of the script merely checks for the presence of the $submit variable and displays an HTML form if it is not present. Here's what the form looks like:


Now, once this form is submitted, calls to error_reporting() and set_error_handler() are used to define the error reporting level and the custom handler for the script.

```
<?php

// adjust this as per your needs
error_reporting(E_ERROR | E_WARNING | E_USER_ERROR |
E_USER_WARNING);

// define handler
set_error_handler("e");

?>
```

Next, the form data is validated. In the event that it fails any of the validation tests, my custom raiseWarning() function is called, and passed a cryptic error code as argument.

```
<?php

// check form data
// raise warnings appropriately
if(!$name || trim($name) == "") { raiseWarning(41, "NAME"); }

if(!$age || trim($age) == "") { raiseWarning(41, "AGE"); }

if(!$email || trim($email) == "") { raiseWarning(41, "EMAIL
ADDRESS"); }


if (!is_numeric($age)) { raiseWarning(41, "AGE"); }

if (!is_valid_email_address($email)) { raiseWarning(41, "EMAIL
ADDRESS"); }

if (!$topping) { raiseWarning(43, "PIZZA TOPPING"); }

?>
```

If you look at the innards of the raiseWarning() function,

```
<?php

function raiseWarning($code, $info="")
{
global $errorCodes;

// use code to get corresponding error message
// from $errorCodes array
```

```
$msg = $errorCodes[$code];

// if additional info available
// append it to message
if ($info != "")
{
$msg .= " -> $info";
}

// raise an error
trigger_error($msg, E_USER_WARNING);
}

?>
```

---

you'll see that it does two very simple things. First, it uses the numeric code passed to it (and any additional information that may be available, like the form field name) to construct a human−readable error message (by mapping the error code to the global $errorCodes array). Next, it uses this error message to raise an error of type E_USER_WARNING via trigger_error().

This error will ultimately be routed to the custom error handler e(), which handles it by adding to an array named $warningList.

---

```
<?php

function e($type, $msg)
{
global $warningList;

// log error
error_log($msg, 0);

switch($type)
{
// if warning, add message to $warningList array
case E_WARNING:
case E_USER_WARNING:
$warningList[] = $msg;
break;

// if error, die()
case E_USER_ERROR:
die("<b>The following fatal error occurred:
$msg</b>");
break;

// everything else
```

```
default:
break;
}
}
?>
```

Once all the form validation has been completed, a call to the displayWarnings() function takes care of displaying the errors in the form (if any).

```
<?php
function displayWarnings()
{
global $warningList;
if (sizeof($warningList) > 0)
{
echo "The following non-fatal errors occurred:";
echo "<ul>";
foreach ($warningList as $w)
{
echo "<li>$w";
}
echo "</ul>";
return true;
}
else
{
return false;
}
}
?>
```

Obviously, if displayWarnings() returns true, it implies that the form data is corrupt and so should not be used as is. And so, a fatal error is raised, via my raiseError() function.

```
<?php

if (displayWarnings()) { raiseError(49); }

?>
```

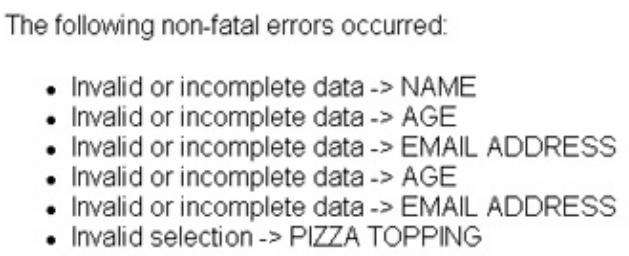
This raiseError() function is identical to the raiseWarnings() function, except that it raises an error of type E_USER_ERROR rather than of type E_USER_WARNING. This error also gets routed to the custom handler e(), which − since this is a fatal error − kills the script with an appropriate message.

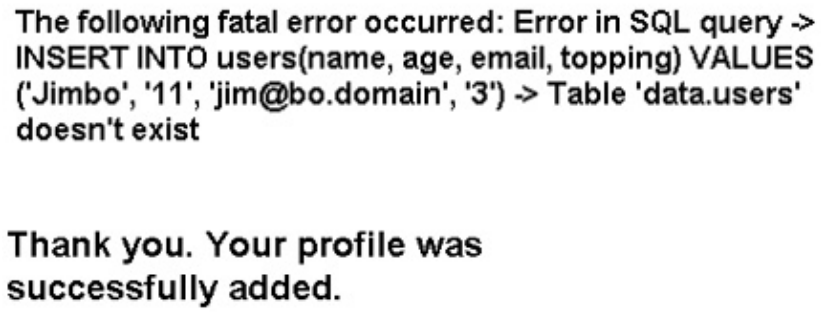
```
<?php

function raiseError($code, $info="")
{
global $errorCodes;
$msg = $errorCodes[$code];

if ($info != "")
{
$msg .= " -> $info";
}

trigger_error($msg, E_USER_ERROR);
}

?>
```

Assuming no errors occurred while validating the form data, the next step is to do something with it − in this case, insert it into a database. Since any difficulty in connecting to the database and executing the query should be considered fatal, I've used the raiseError() function again to catch and handle errors that may occur during this process.

```
<?php

// attempt a MySQL connection
$connection = @mysql_connect("localhost", "john", "doe") or
raiseError(55);

mysql_select_db("data") or raiseError(56);

$query = "INSERT INTO users(name, age, email, topping)
VALUES('$name',
'$age', '$email', '$topping')";

// execute query to input form data
$result = mysql_query($query, $connection) or raiseError(58,
$query . "
-> " . mysql_error());

// success!
echo "<h2>Thank you. Your profile was successfully
added.</h2>";

?>
```

Here are a few images demonstrating how this works in practice:



# Buffer Zone

You might remember, from the first part of this article, an example which demonstrated how a custom error handler could be used to write error messages to a file on a PHP−driven Web site. You might also remember that one of the drawbacks of that script was the fact that warnings would get printed while the page was being constructed.

It's possible to bypass this problem − and also simplify the error logging mechanism used in that example − via a series of judicious calls to PHP's output−buffering functions. Take a look at this revised script, which sets things up just right:

```
<?php
// use an output buffer to store page contents
ob_start();
?>

<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";

// log the error string to the specified log file
error_log($errorString, 3, $errorLog);
```

```
// discard current buffer contents
// and turn off output buffering
ob_end_clean();

// display error page
echo "<html><head><basefont face=Arial></head><body>";

echo "<h1>Error!</h1>";

echo "We're sorry, but this page could not be displayed
because
of an internal error. The error has ben 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>";

echo "</body></html>";

// exit
exit();
}

// 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";
}

// no errors occured
// print buffer contents
ob_end_flush();
?>

</body>
</html>
```

In this case, the first thing I've done is initialized the output buffer via a call to ob_start() − this ensures that all script output is placed in a buffer, rather than being displayed to the user. This output may be dumped to the standard output device at any time via a call to ob_end_flush().

Now, whenever an error occurs, my custom error handler, cleverly named e(), will first flush the output buffer, then send a custom error template to the browser and terminate script execution. So, even if there was a Web page being constructed on the fly when the error occurred, it will never see the light of day, as it will be discarded in favour of the custom error template. If, on the other hand, the script executes without any errors, the final call to ob_end_flush will output the fully−generated HTML page to the browser.

Note that, as before, fatal errors cannot be handled by the custom handler. The only way to avoid the output of fatal error messages is by telling PHP not to display them (take a look at the "display_errors" configuration directive in the PHP configuration file).

# Back To Class

If you're not a big fan of rolling your own code, you might find it instructive and useful to download the free, open−source ErrorHandler class from http://www.phpclasses.org/browse.html/package/345. Created by Gyozo Papp, this PHP class is a robust, full−featured error handler that can easily be integrated into your application code.

The ErrorHandler class comes with some very interesting features: the ability to dump errors to a separate console window so that your primary interface is not disrupted, to include in that error report the source code which caused the error, and to append customized error messages or variable context to the error report. Obviously, it also supports error logging (to a file, an email message, the system logger, or all three) and can catch and replace PHP's error messages with a user−defined error template.

Here's a small example of how it works − take a look at the manual included with the class for more examples and information.

```
// include the class
include('ErrorHandler.inc');

// instantiate an object
$error =&new ErrorHandler();

// configure the error handler

// log errors to a file
$error->report_layout('LOGGING', FILE_LOG, 'error.log');

// don't display symbol table
$error->report_layout('CONTEXT', FALSE);

// uncomment this to hide errors and display a customized
error template
// $error->set_silent(TRUE);
// $error->set_silent("error.html");

// generate some errors
include('non.existent.file');

mysql_connect("localhost", "baduser", "badpassword");
?>
```

And here's what it looks like:

```
[2002-03-01 11:39:46] Malfunction: [WARNING=0x02] Failed opening 'non.existent.file' for inclusion
(include_path='.:/usr/local/lib/php')
SOURCE REPORT from /usr/local/apache/htdocs/e.php around line 21 (20-22)
<?php
// generate some errors
include('non.existent.file');

#*/?>
[2002-03-01 11:39:47] Malfunction: [WARNING=0x02] Can't connect to local MySQL server through socket
'/tmp/mysql.sock' (111)
SOURCE REPORT from /usr/local/apache/htdocs/e.php around line 23 (22-24)
<?php

mysql_connect("localhost", "baduser", "badpassword");
?> [2002-03-01 11:39:47] Malfunction: [WARNING=0x02] MySQL Connection Failed: Can't connect to local MySQL
server through socket '/tmp/mysql.sock' (111)

SOURCE REPORT from /usr/local/apache/htdocs/e.php around line 23 (22-24)
<?php

mysql_connect("localhost", "baduser", "badpassword");
?>
```

# Endgame

And that's about it from me. In this two−part article, you learned how to use PHP's error−handling API to exert fine−grained control over the way the language handles errors. You learned how to control the display of specific error types, how to customize the manner in which they're handled, and how to raise errors of your own. Next, you learnt how to log errors, write them to a file and email them out to all and sundry. And, as if all that wasn't enough, the final section of this article demonstrated the process of creating a robust, scalable error−handling system for a Web application.

In case you'd like to know more about the material discussed in this article, consider checking out the following links:

PHP's error handling and logging features, at http://www.php.net/manual/en/features.error−handling.php

A description of PHP error types, at http://www.php.net/manual/en/phpdevel−errors.php

Gyozo Papp's ErrorHandler class, at http://www.phpclasses.org/browse.html/package/345

PHP output buffering, at http://www.php.net/manual/en/ref.outcontrol.php

PHP's MySQL functions, at http://www.php.net/manual/en/ref.mysql.php

PHP's file handling functions, at http://www.php.net/manual/en/ref.filesystem.php

I hope you enjoyed this article, and that you found it interesting and informative. Till next time...be good!

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!