How to handle those pesky errors in PHP

By Dennis Pallett

Introduction

Let's face it - none of us write perfect code, and your almost-perfect PHP scripts are bound to throw an error now and then, even if you've tested them hundreds of times. It's simply impossible to rule out everything, and that's no problem really, as long as you handle the errors properly.

PHP scripts should never publicly display any error on a live production website, as it could lead to security problems, and it just looks bad to your visitors. Can you imagine your bank's website throwing an error when trying to view your bank account? I'm sure that wouldn't inspire much faith in you.

That's why in this article I'm going to show you how to handle errors in PHP. In this article I will first take you through the basics of error handling, by catching all PHP errors with the set_error_handler() function. After that we'll have a look a new PHP 5 feature called Exceptions which can also be used to create and catch errors. At the end of the article I will show you how to trigger your own errors.

But before we begin with anything, let's have a look at how error reporting works in PHP.

Error reporting in PHP

Error reporting in PHP can be turned on and off in your php.ini file, and it's also possible to define what type of errors should be displayed. There are 11 error types in PHP (12 if you're using PHP5), and they can be found at http://nl2.php.net/manual/en/ref.errorfunc.php#errorlevels. Each level defines a different type of error, but it's usually recommended to use E_ALL on your developer environment, and E_ALL & ~E_NOTICE on your production environment.

In your php.ini several settings can be set related to error reporting. To completely turn off error reporting, you can set the display_errors setting to off, but this is only recommended on production servers, and your development server should always have errors on. It's also possible to turn on error logging, and have all the errors logged to a file.

For this article, make sure error reporting is turned on (display_errors = On) and that your error level is set to E_ALL (error_reporting = E_ALL).

It's also possible to set the error reporting level in your scripts, with the error_reporting() function, e.g.

```
<?php
 // Report all PHP errors
 error_reporting(E_ALL);
```

```
?>
```

That's pretty much the basics of errors in PHP. Let's move on to handling errors.

Handling errors

Let's create a simple script which will throw various errors:

```
<?php
 // Throws a notice
 $a = $b;

// Throws a warning:
 $f = fopen('bla');

// Throws a user notice:
 trigger_error ('Something went wrong!',  E_USER_NOTICE);

// Throws a user warning:
 trigger_error ('Something went REALLY wrong!', E_USER_WARNING);

// Throws a user error:
 trigger_error ('It\'s completely broken!', E_USER_ERROR);
 ?>
```

The above code throws various errors, and it also demonstrates use of the trigger_error() function which can be used to throw your own errors. More on this later in the article.

If you run the above code, you will get several standard ugly PHP errors, and this is something you definitely don't want to see on a live website. To handle these errors, you must set your own error handler, with the set_error_handler() function. The following code demonstrates a really simple error handler:

```
<?php

function handle_errors($errlevel, $errstr, $errfile='', $errline='', $errcontext='') {
        echo htmlentities($errstr) . "";
 }

set_error_handler('handle_errors');

// Throws a notice
 $a = $b;

// Throws a warning:
 $f = fopen('bla');

// Throws a user notice:
 trigger_error ('Something went wrong!',  E_USER_NOTICE);

// Throws a user warning:
 trigger_error ('Something went REALLY wrong!', E_USER_WARNING);
```

```
// Throws a user error:
 trigger_error ('It\'s completely broken!', E_USER_ERROR);

?>
```

(View Live Demo)

As you can see in the above code, the error handler function has five arguments, with the last three being optional. The first argument defines the error level, which can be used to differentiate between different types of errors. The second argument is a description of the error, usually something like "Undefined variable: b" or "fopen() expects at least 2 parameters, 1 given". The third argument gives the filename in which the argument happened, and the fourth argument is used to get the line on which the error happened. The fifth argument, errcontext, gives an array of every variable that existed when the error occurred. You probably won't use that last argument much, but it could still be useful.

In the above example the error handler is still pretty useless, because it treats all the errors the same, even though an E_NOTICE is far less serious as an E_WARNING, but it's very easy to handle errors differently. The below example demonstrates this:

```
<?php

function handle_errors($errlevel, $errstr, $errfile='', $errline='', $errcontext='') {
        $errstr = htmlentities($errstr);

 switch ($errlevel) {
   case E_USER_ERROR:
    echo "<b>My ERROR</b> [$errlevel] $errstr\n";
    echo "  Fatal error in line $errline of file $errfile";
    echo ", PHP " . PHP_VERSION . " (" . PHP_OS . ")\n";
    echo "Aborting...\n";
    exit(1);
    break;
   case E_USER_WARNING:
   case E_WARNING:
    echo "<b>My WARNING</b> [$errlevel] $errstr\n";
    break;
   case E_USER_NOTICE:
   case E_NOTICE:
    echo "<b>My NOTICE</b> [$errlevel] $errstr\n";
    break;
   default:
    echo "Unkown error type: [$errlevel] $errstr\n";
    break;
  }
}

set_error_handler('handle_errors');

// Throws a notice
 $a = $b;

// Throws a warning:
 $f = fopen('bla');
```

```
// Throws a user notice:
 trigger_error ('Something went wrong!',  E_USER_NOTICE);

// Throws a user warning:
 trigger_error ('Something went REALLY wrong!', E_USER_WARNING);

// Throws a user error:
 trigger_error ('It\'s completely broken!', E_USER_ERROR);

?>
```

(View Live Demo)

The above example is able to handle each error level differently, and actually stops when an E_USER_ERROR is thrown. One thing you might notice in the above example is that it doesn't catch the E_ERROR level, only the E_USER_ERROR level. This is because it isn't possible to handle E_ERROR errors, or E_PARSE errors. But these fatal errors are unlikely to slip by your testing phase anyway.

Even though we're now using our own error handler, it's still pretty pointless, since we're still printing out the errors. The next step is to silently log the error, and only return something to the visitor when it's really serious. The below example does just that:

```
<?php

// Location of the error log file
 define ('logfile', dirname(__FILE__) . '/logfile.txt');

function handle_errors($errlevel, $errstr, $errfile='', $errline='', $errcontext='') {
   $errstr = htmlentities($errstr);
   $error = '[' . date('d/m/Y H:i:s') . '] ';

 switch ($errlevel) {
   case E_USER_ERROR:
    $error .= "ERROR: ";

   echo 'Sorry, something unexpected happen, and it has been logged for further investigation. Please go back where you came from. Thank you for your understanding.';
    $die = true;
    break;
   case E_USER_WARNING:
   case E_WARNING:
    $error .= "WARNING: ";
    break;
   case E_USER_NOTICE:
   case E_NOTICE:
    $error .= "NOTICE: ";
    break;
   default:
    $error .= "UNKNOWN: ";
    break;
   }

  $error .= " $errstr in line $errline of file $errfile\n";
```

```
  // Log error
   $f = fopen(logfile, 'a');
   fwrite($f, $error);
   fclose($f);

  // Error -> stop script execution?
   if (isset($die) AND $die == true) {
         die();
   }
 }

echo

set_error_handler('handle_errors');

// Throws a notice
 $a = $b;

// Throws a warning:
 $f = fopen('bla');

// Throws a user notice:
 trigger_error ('Something went wrong!',  E_USER_NOTICE);

// Throws a user warning:
 trigger_error ('Something went REALLY wrong!', E_USER_WARNING);

// Throws a user error:
 trigger_error ('It\'s completely broken!', E_USER_ERROR);

?>
```

(View Live Demo)

If you run the above code, the only thing that will be printed on the screen is the apology message and nothing more. Meanwhile, all the errors are being logged to the log file (defined at the top of the script), which you will have to check every 2 days or so.

That's it for error handling; let's move on to a PHP5 feature, called exceptions. If you don't have PHP5, or simply aren't interested, you can skip the next section.

PHP5 and Exceptions

PHP5 has got a new type of errors called exceptions, something you might already be familiar with if you've used another programming language (like Java). Exceptions are mainly useful if you want to trigger your own errors, and most standard functions in PHP5 still throw regular errors (which is a shame really). The below code demonstrates a simple exception:

```
<?php

try {
    $error = 'Always throw this error';
```

```
    throw new Exception($error);

   // Code following an exception is not executed.
    echo 'Never executed';

} catch (Exception $e) {
    echo 'Caught exception: ',  $e->getMessage(), "\n";
 }

echo 'And the code execution goes on...';

?>
```

The try block is used to do a certain action whereby an exception might be thrown, which can then be handled by the catch block, as seen in the above example. The example is pretty pointless though, as there will always be an exception in the try block since an exception is actually thrown there on purpose. But imagine something like this:

```
<?php

try {
        // Try loading file
        $data = load_file('C:\does\not\exist.txt');

   // Code following an exception is not executed.
    echo 'Never executed';
 } catch (Exception $e) {
        // Can't load the file, handle it
        echo 'Unable the load file because: ' . $e->getMessage() . "\n";
 }

echo 'And the code execution goes on...';

function load_file ($file) {
        if (file_exists($file) == false) {
                throw new Exception('File doesn\'t exist');
                return false;
        }

       // load the file here

}

?>
```

The above example really demonstrates why exceptions can be so useful. By throwing an exception in the load_file function, you can handle the problem in the catch block, making it much easier to handle errors in PHP5 than in PHP4.

The only downside of using exceptions is that you must always catch them, or otherwise they are printed to the visitor, and they look even worse then regular errors. But it's also possible to set an exception handler to handle all the uncaught exceptions, like so:

```
<?php

function handle_exceptions($e) {
        echo "Uncaught exception: " , $e->getMessage(), "\n";
 }

set_exception_handler('handle_exceptions');

// This exception won't be caught
 // since it's not in a try-catch block
 $error = 'Always throw this error';
 throw new Exception($error);

echo 'Never executed';

?>
```

In the example above the exception isn't in a try-catch block, so it won't get caught, but the exception handler will take care of it. Be aware though that once the exception handler is called, execution of the script stops, so it's always better to use a try-catch block.

You can also modify the exception handler to write to the log file as well, e.g.

```
<?php

// Location of the error log file
 define ('logfile', dirname(__FILE__) . '/logfile.txt');

function handle_exceptions($e) {
        $error = '[' . date('d/m/Y H:i:s') . '] EXCEPTION: ';
        $error .= $e->getMessage() . " in line " . $e->getLine() . " of file " . $e->getFile() . "\n";

       // Log error
        $f = fopen(logfile, 'a');
        fwrite($f, $error);
        fclose($f);

       echo 'Sorry, something unexpected happen, and it has been logged for further investigation. Please go back where you came from. Thank you for your understanding.';
 }

set_exception_handler('handle_exceptions');

// This exception won't be caught
 // since it's not in a try-catch block
 $error = 'Always throw this error';
 throw new Exception($error);

echo 'Never executed';

?>
```

That's about it for exceptions. Let's move on to the final part of the article: how to trigger your own errors.

Creating your own errors

Triggering your own errors sounds pretty simple, especially with functions like trigger_error which I've already talked about a bit, but it's actually harder than you think. The whole point of triggering your own errors is so that the rest of your PHP application knows what's going on. This sounds a bit cryptic, so let me demonstrate it with an example:

```
<?php

// I want to load a file, so I use a custom function:
 $data = load_file ('C:\myile.txt');

// What happened? Did I successfully load the file or not?
 // How can I found out?

function load_file ($file) {
        // Check if file exists
        if (file_exists($file) == false) {
                // Need to return an error, but how?
        }

       // Load the file...
 }

?>
```

The above example demonstrates exactly what I mean. I use a custom function to load a file, and I expect it to return file data, but it's possible that an error occurs, which needs to be handled by my script. The most common solution is to simply have the function return false, like so:

```
<?php

// I want to load a file, so I use a custom function:
 $data = load_file ('C:\myile.txt');

if ($data == false) {
        echo 'Unable to load file';
 }

function load_file ($file) {
        // Check if file exists
        if (file_exists($file) == false) {
                return false;
        }

       // Load the file...
 }

?>
```

This does the job, and works, but it's far from perfect because you don't really know what the problem is, and therefore can't return an informative error message either.

Another possibility is to use exceptions, like so:

```
<?php

// I want to load a file, so I use a custom function:
 try {
        $data = load_file ('C:\myile.txt');
 } catch (Exception $e) {
        echo 'Error ' . $e->getCode() . ': ' . $e->getMessage();
 }

function load_file ($file) {
        // Check if file exists
        if (file_exists($file) == false) {
                throw new Exception('File doesn\'t exist', 10);
                return false;
        }

       // Load the file...
 }

?>
```

This is already much better, because you can define your own error numbers and messages, which means you can create informative messages, and handle each error differently. But if you're still stuck on PHP4, this is no solution, because you can't use exceptions, and another disadvantage of using exceptions is that you must catch them, otherwise they will halt execution of your script.

A better solution is to use a global static class which emulates exceptions in a different way, and this is a method I really like myself. See the (simplified) example below to understand what I mean:

```
<?php

// I want to load a file, so I use a custom function:
 $data = load_file ('C:\myile.txt');

// Anything went wrong? Check error class for code and message
 if ($data == false) {
        echo 'Error ' . Error::getCode() . ': ' . Error::getMessage();
 }

function load_file ($file) {
        // Check if file exists
        if (file_exists($file) == false) {
                Error::throwError ('File doesn\'t exist', 10);
                return false;
        }

       // Load the file...
 }

Class Error {
        function throwError ($msg, $num=) {
```

```
            $errors =& Error::errors();
            $errors[] = array('errmsg' => $msg, 'errnum' => $num);
        }

        function getCode() {
            // Get code of latest error
            $errors =& Error::errors();
            $latest = end($errors);

            return $latest['errnum'];
        }

        function getMessage() {
            // Get message of latest error
            $errors =& Error::errors();
            $latest = end($errors);

            return $latest['errmsg'];
        }

        function &errors() {
            static $errors;
            return $errors;
        }
}

?>
```

In the example above you get the best of both worlds: exception-like functionality, meaning it's possible to define your own error codes and messages, it's PHP4 compatible, and there's no need to use a try-catch block either, because it doesn't matter if you don't catch any of these errors. Another advantage is that it's possible to get a list of all the errors that happened in your script.

Of course the above is just a simplified example, and there are different ways of doing it, but I've had much success with the above method.

Conclusion

In this article I've first taken you through the basics of error handling in PHP, and after that shown you some more advanced PHP5 features, ending with a custom solution for triggering your own errors. Errors can be a really useful tool, especially during development but also on your live production website. Make sure you log every error that happens, including notices, and check your error log every 2-3 days. It's likely you'll spot something you haven't seen before, and it could save you a lot of trouble in the long run.

If you have any comments or questions on this article, feel free to leave them in the comments below or join us at the PHPit Forums.

About this PDF

For more information, contact us at support@pallettgroup.com.