PHP Error Handling

When creating scripts and web applications, error handling is an important part. If your code lacks error checking code, your program may look very unprofessional and you may be open to security risks.

This tutorial contains some of the most common error checking methods in PHP.

We will show different error handling methods:

- Simple "die()" statements
- Custom errors and error triggers
- Error reporting

Basic Error Handling: Using the die() function

The first example shows a simple script that opens a text file:

```
<?php
$file=fopen("welcome.txt","r");
?>
```

If the file does not exist you might get an error like this:

```
Warning: fopen(welcome.txt) [function.fopen]: failed to open stream:
No such file or directory in C:\webfolder\test.php on line 2
```

To avoid that the user gets an error message like the one above, we test if the file exist before we try to access it:

```
<?php
if(!file_exists("welcome.txt"))
 {
 die("File not found");
 }
else
 {
 $file=fopen("welcome.txt","r");
 }
?>
```

Now if the file does not exist you get an error like this:

```
File not found
```

The code above is more efficient than the earlier code, because it uses a simple error handling mechanism to stop the script after the error.

However, simply stopping the script is not always the right way to go. Let's take a look at alternative PHP functions for handling errors.

Creating a Custom Error Handler

Creating a custom error handler is quite simple. We simply create a special function that can be called when an error occurs in PHP.

This function must be able to handle a minimum of two parameters (error level and error message) but can accept up to five parameters (optionally: file, line-number, and the error context):

Syntax

```
error_function(error_level,error_message,
error_file,error_line,error_context)
```

Parameter	Description
Error_level	Required. Specifies the error report level for the user-defined error. Must be a value number. See table below for possible error report levels
Error_message	Required. Specifies the error message for the user-defined error
Error_file	Optional. Specifies the filename in which the error occurred
Error_line	Optional. Specifies the line number in which the error occurred
Error_context	Optional. Specifies an array containing every variable, and their values, in use when the error occurred

Error Report levels

These error report levels are the different types of error the user-defined error handler can be used for:

Value	Constant	Description
2	E_WARNING	Non-fatal run-time errors. Execution of the script is not halted
8	E_NOTICE	Run-time notices. The script found something that might be an error, but could also happen when running a script normally
256	E_USER_ERROR	Fatal user-generated error. This is like an E_ERROR set by the programmer using the PHP function trigger_error()
512	E_USER_WARNING	Non-fatal user-generated warning. This is like an E_WARNING set by the programmer using the PHP function trigger_error()
1024	E_USER_NOTICE	User-generated notice. This is like an E_NOTICE set by the programmer using the PHP function trigger_error()
4096	E_RECOVERABLE_ERROR	Catchable fatal error. This is like an E_ERROR but can be caught by a user defined handle (see also set_error_handler())
8191	E_ALL	All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0)

Now lets create a function to handle errors:

```
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }
```

The code above is a simple error handling function. When it is triggered, it gets the error level and an error message. It then outputs the error level and message and terminates the script.

Now that we have created an error handling function we need to decide when it should be triggered.

Set Error Handler

The default error handler for PHP is the built in error handler. We are going to make the function above the default error handler for the duration of the script.

It is possible to change the error handler to apply for only some errors, that way the script can handle different errors in different ways. However, in this example we are going to use our custom error handler for all errors:

```
set_error_handler("customError");
```

Since we want our custom function to handle all errors, the set_error_handler() only needed one parameter, a second parameter could be added to specify an error level.

Example

Testing the error handler by trying to output variable that does not exist:

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr";
 }
//set error handler
set_error_handler("customError");
//trigger error
echo($test);
?>
```

The output of the code above should be something like this:

```
Custom error: [8] Undefined variable: test
```

Trigger an Error

In a script where users can input data it is useful to trigger errors when an illegal input occurs. In PHP, this is done by the trigger_error() function.

Example

In this example an error occurs if the "test" variable is bigger than "1":

```
<?php
$test=2;
if ($test>1)
{
trigger_error("Value must be 1 or below");
}
```

```
?>
```

The output of the code above should be something like this:

```
Notice: Value must be 1 or below
in C:\webfolder\test.php on line 6
```

An error can be triggered anywhere you wish in a script, and by adding a second parameter, you can specify what error level is triggered.

Possible error types:

- E_USER_ERROR - Fatal user-generated run-time error. Errors that can not be recovered from. Execution of the script is halted
- E_USER_WARNING - Non-fatal user-generated run-time warning. Execution of the script is not halted
- E_USER_NOTICE - Default. User-generated run-time notice. The script found something that might be an error, but could also happen when running a script normally

Example

In this example an E_USER_WARNING occurs if the "test" variable is bigger than "1". If an E_USER_WARNING occurs we will use our custom error handler and end the script:

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }
//set error handler
set_error_handler("customError",E_USER_WARNING);
//trigger error
$test=2;
if ($test>1)
 {
 trigger_error("Value must be 1 or below",E_USER_WARNING);
 }
?>
```

The output of the code above should be something like this:

```
Error: [512] Value must be 1 or below
Ending Script
```

Now that we have learned to create our own errors and how to trigger them, lets take a look at error logging.

Error Logging

By default, PHP sends an error log to the servers logging system or a file, depending on how the error_log configuration is set in the php.ini file. By using the error_log() function you can send error logs to a specified file or a remote destination.

Sending errors messages to yourself by e-mail can be a good way of getting notified of specific errors.

Send an Error Message by E-Mail

In the example below we will send an e-mail with an error message and end the script, if a specific error occurs:

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Webmaster has been notified";
 error_log("Error: [$errno] $errstr",1,
 "someone@example.com","From: webmaster@example.com");
}
//set error handler
set_error_handler("customError",E_USER_WARNING);
//trigger error
$test=2;
if ($test>1)
 {
 trigger_error("Value must be 1 or below",E_USER_WARNING);
 }
?>
```

The output of the code above should be something like this:

```
Error: [512] Value must be 1 or below
Webmaster has been notified
```

And the mail received from the code above looks like this:

```
Error: [512] Value must be 1 or below
```

This should not be used with all errors. Regular errors should be logged on the server using the default PHP logging system.

Validating Data

Some of the validation strategies discussed in this section use regular expressions, which are powerful text-matching patterns, written in a language all their own. If you're not familiar with regular expressions, provides a quick introduction.

Example 6-9. Displaying error messages with the form

```
// Logic to do the right thing based on

// the hidden _submit_check parameter

if ($_POST['_submit_check']) {

    // If validate_form( ) returns errors, pass them to show_form( )

    if ($form_errors = validate_form( )) {

        show_form($form_errors);

    } else {

        process_form( );

    }

} else {

    show_form( );

}


// Do something when the form is submitted

function process_form( ) {

    print "Hello, ". $_POST['my_name'];

}


// Display the form

function show_form($errors = '') {

    // If some errors were passed in, print them out

    if ($errors) {

        print 'Please correct these errors: <ul><li>';

        print implode('</li><li>', $errors);

        print '</li></ul>';

    }
```

```
    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
<input type="hidden" name="_submit_check" value="1">
</form>
_HTML_;
}

// Check the form data
function validate_form( ) {
    // Start with an empty array of error messages
    $errors = array( );

    // Add an error message if the name is too short
    if (strlen($_POST['my_name']) < 3) {
        $errors[  ] = 'Your name must be at least 3 letters long.';
    }

    // Return the (possibly empty) array of error messages
    return $errors;
}
```

Required Elements

To make sure something has been entered into a required element, check the element's length with strlen( ), as in Example 6-10.

Verifying a required element

```
if (strlen($_POST['email']) =  = 0) {
   $errors [  ] = "You must enter an email address.";
}
```

It is important to use strlen( ) when checking a required element instead of testing the value itself in an if( ) statement. A test such as if (! $_POST['quantity']) treats a value that evaluates to false as an error. Using strlen( ) lets users enter a value such as 0 into a required element.

Numeric or String Elements

To ensure that a submitted value is an integer or floating-point number, use the conversion functions intval( ) and floatval( ). They give you the number (integer or floating point) inside a string, discarding any extraneous text or alternative number formats.

To use these functions for form validation, compare a submitted form value with what you get when you pass the submitted form value through intval( ) or floatval( ) and then through strval( ). The strval( ) function converts the cleaned-up number back into a string so that the comparison with the element of $_POST works properly. If the submitted string and the cleaned-up string don't match, then there is some funny business in the submitted value and you should reject it.

Checking for an integer

```
if ($_POST['age'] != strval(intval($_POST['age'])) {

   $errors[  ] = 'Please enter a valid age.';

}
```

Use of floatval( ) and strval( ) to check that a submitted value is a floating-point or decimal number.

Checking for a floating-point number

```
if ($_POST['price'] != strval(floatval($_POST['price']))) {

   $errors[  ] = 'Please enter a valid price.';

}
```

Combining trim( ) and strlen( )

```
if (strlen(trim($_POST['name'])) =  = 0) {

   $errors[  ] = "Your name is required.";

}
```

Number Ranges

To check whether a number falls within a certain range, first make sure the input is a number. Then, use an if( ) statement to test the value of the input

Checking for a number range

```
if ($_POST['age'] != strval(intval($_POST['age']))) {

   $errors[  ] = "Your age must be a number.";

} elseif (($_POST['age'] < 18) || ($_POST['age'] > 65)) {
```

```
    $errors[  ] = "Your age must be at least 18 and no more than 65.";

}
```

Email Addresses

If you don't want to go to all the trouble of verifying the email address with a separate message, there are still some syntax checks you can do in your form validation code to weed out mistyped addresses. The regular expression ^[^@\s]+@([-a-z0-9]+\.)+[a-z]{2,}$ matches most common email addresses and fails to match common mistypings of addresses. Use it with preg_match( )

Checking the syntax of an email address

```
if (! preg_match('/^[^@\s]+@([-a-z0-9]+\.)+[a-z]{2,}$/i',

                $_POST['email'])) {

    $errors[  ] = 'Please enter a valid e-mail address';

}
```

The one danger with this regular expression is that it doesn't allow any whitespace in the username part of the email address (before the @). An address such as "Marles Pickens"@sludge.example.com is valid according to the standard that defines Internet email addresses, but it won't pass this test because of the space character in it. Fortunately, addresses with embedded whitespace are rare enough that you shouldn't run into any problems with it.