Chapter 12. Error Handling

The best laid plans of mice and men often go awry.

adapted from Robert Burns

If you work in programming or software development, you’re probably very familiar with bugs and the process of debugging. You might have even spent as much time, if not more, tracking down bugs as you do writing code in the first place. It’s an unfortunate maxim of software—no matter how hard a team works to build correct software, there will inevitably be a failure that needs to be identified and corrected.

Luckily, PHP makes finding bugs relatively straightforward. The forgiving nature of the language often also renders a bug a nuisance rather than a fatal flaw.

The following recipes introduce the quickest and easiest way to identify and handle bugs in your code. They also detail how to both code and handle custom exceptions thrown by your code in the event of invalid data output by a third-party API or other incorrect system behavior.

12.1 Finding and Fixing Parse Errors

Problem

The PHP compiler has failed to parse a script within your application; you want to find and correct the problem quickly.

Solution

Open the offending file in a text editor and review the line called out by the parser for syntax errors. If the problem isn’t immediately apparent, walk backwards through the code one line at a time until you find the problem and make corrections in the file.

Discussion

PHP is a relatively forgiving language and will often attempt to let even an incorrect or problematic script run to completion. In many situations, though, the parser cannot properly interpret a line of code to identify what should be done and will instead return an error.

As a contrived example, loop through Western states in the US:

```
$states = ['Washington', 'Oregon', 'California'];
foreach $states as $state {
    print("{$state} is on the West coast.") . PHP_EOL;
}
```

This code, when run in a PHP interpreter, will throw a `Parse error` on the second line:

```
PHP Parse error:  syntax error, unexpected variable "$s
in php shell code on line 2
```

Based on this error message alone, you can zero in on the offending line. Remember that, though `foreach` is a language construct, it is still written similar to a function call with parentheses. The correct way to iterate through the array of states would be as follows:

```
$states = ['Washington', 'Oregon', 'California'];
foreach ($states as $state) {
    print("{$state} is on the West coast.") . PHP_EOL;
}
```

This particular error—omitting parentheses while leveraging language constructs—is common among developers frequently moving between languages. The same mechanism in Python, for example, looks nearly the same but is syntactically correct when omitting the parentheses on the `foreach` call. For example:

```python
states = ['Washington', 'Oregon', 'California']
for state in states:
    print(f"{state} is on the West coast.")
```

The syntax of the two languages is confusingly similar. Thankfully, they are different enough that each language's parser will catch these differences and alert you, should you make such a mistake when moving back and forth between projects.

Conveniently, IDEs like Visual Studio Code automatically parse your script and highlight any syntax errors for you. Figure 12-1 illustrates how this highlighting makes it relatively easy to track down and correct issues before your application ever runs.

Figure 12-1. Visual Studio Code identifies and highlights syntax errors before your application runs

See Also

The list of tokens, various parts of the source code, used by the PHP parser.

12.2 Creating and Handling Custom Exceptions

Problem

You want your application to throw (and catch) a custom exception when things go wrong.

Solution

Extend the base `Exception` class to introduce custom behavior, then leverage `try` / `catch` blocks to capture and handle exceptions.

Discussion

PHP defines a basic `Throwable` interface implemented by any kind of error or exception in the language. Internal issues are then represented by the `Error` class and its descendants, while problems in userland are represented by the `Exception` class and its descendants.

Generally, you will only ever be extending the `Exception` class within your application, but you can catch any `Throwable` implementation within a standard `try` / `catch` block.

For example, assume you are implementing a division function with very precise, custom functionality:

1. Division by 0 is not allowed.
2. All decimal values will be rounded down.
3. The integer 42 is never valid as a numerator.
4. The numerator must be an integer, but the denominator can also be a float.

Such a function might leverage built-in errors like `ArithmeticError` or `DivisionByZeroError`. But in the preceding list of rules, the third stands out as requiring a custom exception. Before defining your function, you would define a custom exception as in Example 12-1.

Example 12-1. Simple custom exception definition

```
class HitchhikerException extends Exception
{
    public function __construct(int $code = 0, Throwabl
    {
        parent::__construct('42 is indivisible.', $code
    }

    public function __toString()
    {
        return __CLASS__ . ": [{$this->code}]: {$this->
    }
}
```

Once the custom exception exists, you can `throw` it within your custom division function as follows:

```
function divide(int $numerator, float|int $denominator)
{
    if ($denominator === 0) {
        throw new DivisionByZeroError;
    } elseif ($numerator === 42) {
        throw new HitchhikerException;
    }

    return floor($numerator / $denominator);
}
```

Once you've defined your custom functionality, it's a matter of leveraging that code in an application. You know the function *could* throw an error, so it's important to wrap any invocation in a `try` statement and handle that error appropriately. Example 12-2 will iterate through four pairs of numbers, attempting division on each, and handling any subsequent errors/exceptions thrown.

Example 12-2. Handling errors in custom division

```
$pairs = [
    [10, 2],
    [2, 5],
    [10, 0],
    [42, 2]
];

foreach ($pairs as $pair) {
    try {
        echo divide($pair[0], $pair[1]) . PHP_EOL;
    } catch (HitchhikerException $he) {
                                      ❶

        echo 'Invalid division of 42!' . PHP_EOL;
    } catch (Throwable $t) {
                                      ❷
        echo 'Look, a rabid marmot!' . PHP_EOL;
    }
}
```

If 42 is ever passed as a numerator, the `divide()` function will throw a `HitchhikerException` and fail to recover. Capturing this exception allows you to provide feedback either to the application or to the user and move on.

Any other `Error` or `Exception` thrown by the function will be caught as an implementation of `Throwable`. In this case, you're throwing that error away and moving on.

See Also

Documentation on the following:

- The base `Exception` class
- The list of predefined exceptions
- Additional exceptions as defined by the Standard PHP Library (SPL)
- Creating custom exceptions through extension
- The error hierarchy as of PHP 7

12.3 Hiding Error Messages from End Users

Problem

You've fixed all of the bugs you know of and are ready to launch your application in production. But you also want to prevent any new errors from being displayed to end users.

Solution

To completely suppress errors in production, set both the `error_reporting` and `display_errors` directives in *php.ini* to `Off` as follows:

```
; Disable error reporting
error_reporting = Off
display_errors  = Off
```

Discussion

The configuration change presented in the Solution example will impact your entire application. Errors will be entirely suppressed and, even if they were to be thrown, will never be displayed to an end user. It's considered bad practice to present errors or unhandled exceptions directly to users. It might also lead to security issues if stack traces are presented directly to end users of the application.

However, if your application is misbehaving, nothing will be logged for the development team to diagnose and address.

For a production instance, leaving `display_errors` set to `Off` will still hide errors from end users, but reverting to the default `error_reporting` level will reliably send any errors to the logs.

There might be specific pages with known errors (due to legacy code, poorly written dependencies, or known technical debt) that you wish to omit, though. In those situations you can *programmatically* set the error reporting level by using the `error_reporting()` function in PHP. This function accept a new error reporting level and returns whatever level was previously set (the default if not previously configured).

As a result, you can use calls to `error_reporting()` to wrap problematic blocks of code and prevent too-chatty errors from being presented in the logs. For example:

```
$error_level = error_reporting(E_ERROR);
❶

// ... Call your other application code here.

error_reporting($error_level);
❷
```

❶

Set the error level to the absolute minimum, including only fatal runtime errors that halt script execution.

Return the error level to its previous state. ❷

The default error level is `E_ALL`, which presents all errors, warnings, and notices.[1] You can use integer reporting levels to override this, but PHP presents several named constants that represent each potential setting. These constants are enumerated in Table 12-1.

NOTE

Prior to PHP 8.0, the default error reporting level started with `E_ALL` and then explicitly removed diagnostic notices (`E_NOTICE`), strict type warnings (`E_STRICT`), and deprecation notices (`E_DEPRECATED`).

Table 12-1. Error reporting level constants

Integer value	Constant	Description
1	`E_ERROR`	Fatal runtime errors that result in script execution halting.
2	`E_WARNING`	Runtime warnings (nonfatal errors) that do not halt script execution.
4	`E_PARSE`	Compile-time errors generated by the parser.
8	`E_NOTICE`	Runtime notices that indicate that the script encountered something that could indicate an error but could also happen in the normal course of running a script.
16	`E_CORE_ERROR`	Fatal errors that occur during PHP's initial startup. This is like

Integer value	Constant	Description
		`E_ERROR` , except it is generated by the core of PHP.
32	`E_CORE_WARNING`	Warnings (nonfatal errors) that occur during PHP's initial startup. This is like `E_WARNING` , except it is generated by the core of PHP.
64	`E_COMPILE_ERROR`	Fatal compile-time errors. This is like `E_ERROR` , except it is generated by the Zend Scripting Engine.
128	`E_COMPILE_WARNING`	Compile-time warnings (nonfatal errors). This is like `E_WARNING` , except it is generated by the Zend Scripting Engine.
256	`E_USER_ERROR`	User-generated error messages. This is like `E_ERROR` , except it is generated in PHP code by using the PHP function `trigger_error()` .
512	`E_USER_WARNING`	User-generated warning messages. This is like `E_WARNING` , except it is generated in PHP code by using the PHP function `trigger_error()` .
1024	`E_USER_NOTICE`	User-generated notice messages. This is like `E_NOTICE` , except it is generated in PHP code by using the PHP function `trigger_error()` .

Integer value	Constant	Description
2048	`E_STRICT`	Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code.
4096	`E_RECOVERABLE_ERROR`	Catchable fatal errors. A dangerous error occurred, but PHP is not unstable and can recover. If the error is not caught by a user-defined handle (see also Recipe 12.4), the application aborts as if it was an `E_ERROR` .
8192	`E_DEPRECATED`	Runtime notices. Enable this to receive warnings about code that will not work in future versions.
16384	`E_USER_DEPRECATED`	User-generated warning messages. This is like `E_DEPRECATED` , except it is generated in PHP code by using the PHP function `trigger_error()` .
32767	`E_ALL`	All errors, warnings, and notices.

Note that you can combine error levels via binary operations, creating a bitmask. A simple error reporting level might include errors, warnings, and parser errors alone (omitting core, user errors, and notices). This level would be adequately set with the following:

```
error_reporting(E_ERROR | E_WARNING | E_PARSE);
```

See Also

Documentation on `error_reporting()`, the `error_reporting` directive, and the `display_errors` directive.

12.4 Using a Custom Error Handler

Problem

You want to customize the way PHP handles and reports errors.

Solution

Define your custom handler as a callable function in PHP, then pass that function into `set_error_handler()` as follows:

```
function my_error_handler(int $num, string $str, string
{
    echo "Encountered error $num in $file on line $line
}

set_error_handler('my_error_handler');
```

Discussion

PHP will leverage your custom handler in most situations where an error is recoverable. Fatal errors, core errors, and compile-time issues (like parser errors) either halt or entirely prevent program execution and cannot be handled with a user function. Specifically, `E_ERROR`, `E_PARSE`, `E_CORE_ERROR`, `E_CORE_WARNING`, `E_COMPILE_ERROR`, and `E_COMPILE_WARNING` errors can never be caught. In addition, most `E_STRICT` errors in the file that invoked `set_error_handler()` cannot be caught, as the errors will be thrown before the custom handler is properly registered.

If you define a custom error handler consistent with the one presented in the Solution example, any catchable errors will then invoke this function and print data to the screen. As illustrated in Example 12-3, attempting to `echo` an undefined variable will cause an `E_WARNING` error.

Example 12-3. Catching recoverable runtime errors

```
echo $foo;
```

With `my_error_handler()` from the Solution example defined and registered, the erroneous code in Example 12-3 will print the following text to the screen, referencing the integer value of the `E_WARNING` error type:

```
Encountered error 2 in php shell code on line 1: Undefi
```

Once you've caught an error to handle it, if the error is something that will lead to instability in the application, it is your responsibility to invoke `die()` to halt execution. PHP won't do this for you outside the handler and will instead continue processing the application as if no error had been thrown.

If, once you've handled errors in part of your application, you wish to restore the original (default) error handler, you should do so by calling `restore_error_handler()`. This merely reverts your earlier error handler registration and restores whatever error handler was previously registered.

Similarly, PHP empowers you to register (and restore) custom exception handlers. These operate the same as custom error handlers but instead capture any exception thrown outside a `try` / `catch` block. Unlike error handlers, program execution will halt after the custom exception handler has been called.

For more on exceptions, review Recipe 12.2 and the documentation for both `set_exception_handler()` and `restore_exception_&#x200b;han&#x2060;dler()`.

See Also

Documentation on `set_error_handler()` and `restore_error_handler()`.

12.5 Logging Errors to an External Stream

Problem

You want to log application errors to a file or to an external source of some sort for future debugging.

Solution

Use `error_log()` to write errors to the default log file as follows:

```
$user_input = json_decode($raw_input);
if (json_last_error() !== JSON_ERROR_NONE) {
    error_log('JSON Error #' . json_last_error() . ': '
}
```

Discussion

By default, `error_log()` will log errors to whatever location is specified by the `error_log` directive in *php.ini*. Often on Unix-style systems, this will be a file within */var/log* but can be customized to be anywhere within your system.

The optional second parameter of `error_log()` allows you to route error messages where necessary. If the server is set up to send emails, you can specify a message type of `1` and provide an email address for the optional third parameter to send errors by email. For example:

```
error_log('Some error message', 1, 'developer@somedomai
```

NOTE

Under the hood, `error_log()` will use the same functionality as `mail()` to send errors by email. In many cases, this might be disabled for security purposes. Be sure to verify the functionality of any mail systems before relying on this functionality, particularly in a production environment.

Alternatively, you can specify a file *other* than the default log location as the destination and pass the integer `3` as the message type. Rather than writing to the default logs, PHP will append the message to that file directly. For example:

```
error_log('Some error message', 3, 'error_log.txt');
```

WARNING

When logging directly to a file with `error_log()`, the system will *not* append a newline character automatically. You will be responsible for either appending `PHP_EOL` to any string or encoding the `\r\n` newline literals.

Chapter 11 covers the file protocol at length as well as the other streams exposed by PHP. Remember that directly referencing a filepath is transparently leveraging the `file://` protocol so, in reality, you are logging errors to a file *stream* with the preceding code block. You can just as easily reference any other kind of stream so long as you properly reference the stream protocol. The following example logs errors directly to the console's standard error stream:

```
error_log('Some error message', 3, 'php://stderr');
```

See Also

Documentation on `error_log()` and Recipe 13.5's coverage of Monolog, a more comprehensive logging library for PHP applications.

1

The default error level can be directly set in *php.ini* and, in many environments, might already be set to something other than `E_ALL`. Confirm your own environment's configuration to be sure.