PHP Debugging

Draft: March 19, 2013

Introduction

Debugging is the art of locating errors in your code. There are three types of errors to deal with:

1. **Syntax errors:** When code violates the grammatical rules of the language because of a missing semicolon, misspelled keyword, improperly balanced quotes, parentheses, curly braces, etc., the code cannot even start to execute. Normally, syntax errors are the easiest to fix: the compiler or interpreter issues an error message telling you where the error is and some information about it.
2. **Runtime errors:** When a program is able to start running (no syntax errors), it may still fail because it violates some property of the language or the programming environment. Different languages can detect different types of runtime errors. For PHP, some examples are references to variables that have not been assigned a value yet, attempts to do arithmetic on non-numeric variables, and many more. The runtime system detects these errors and normally tells where they occurred in the code. For an interpreted language like PHP, the error message is as useful as a syntax error: the processor will issue a message telling what the error is and where it was detected.
3. **Logic errors:** These are errors that go undetected by the language processor: they are mistakes in the code that cause it to generate the wrong results. They are the most difficult to fix because there is no message that tells you where the error occurred. The programmer has to test all code carefully to make sure there are no logic errors, and when one does show up, the programmer has to figure out what the mistake was that caused it, all with no help from the language processor.

PHP runs in the context of a web server, which might be running on a different computer connected by the Internet to the one where you are writing your code. As a result, debugging PHP code is more difficult than other languages, where it is relatively easy to receive syntax and runtime error messages and even to interact directly with running code to track down logic errors.

This document covers some of the techniques available for debugging PHP code.

One important debugging tool is outside the scope of this document: *xdebug* is a module that can be added to a PHP processor so that you can trace and interact with PHP scripts in much the way debugging tools for other languages work. But (a) not all PHP installations have the *xdebug* module installed and (b) a separate client program is needed for interacting with the *xdebug* module, and there is no one client that "everyone" uses. As a result, there is a lot of setup to do before you can use *xdebug.*

Syntax Errors

Surprisingly, the simplest types of errors lead us directly into what are perhaps the most technically awkward techniques for debugging PHP code.

To set the background for those techniques, we need to review one model of how programs handle input and output:

In this model, *stdin, stdout,* and *stderr* are "streams" of characters. A program reads characters from *stdin,* writes its "normal" output to *stdout,* and writes error messages to *stderr.*

When this model was first adopted by the Unix operating system, the standard input to a program normally came from a user's keyboard, so the *stdin* stream was whatever sequence of keystrokes the user typed. The standard output and error streams were normally the sequence of characters displayed on the user's terminal. The Unix system's innovation was to provide a mechanism for connecting *stdin, stdout,* and *stderr* to other input/output devices (such as disk files or network connections) or even to other programs, in a *pipeline*. The '|' character on the command line to link programs this way, which is why it is usually called the "pipe character." By keeping *stdout* separate from *stderr* a pipeline could connect the normal output of one program to another one, while still sending messages about error conditions to the user's terminal or to a *log file* for later examination.

Here's an example Unix command line to illustrate the process:

```
cat < *.php | sort | uniq > distinct_lines.txt 2> error.log
```

The *cat* command (short for "concatenate") simply copies files from *stdin* to *stdout.* In this example, its *stdin* is redirected, using the '<' symbol, from the keyboard to come from the contents of all the files in the current directory that have the *.php* extension. The *stdout* stream of the *cat* command is piped to the *stdin* stream of the *sort* command, which arranges all of the lines it reads from *stdin* in alphabetical order; *stdout* of *sort* is piped to *stdin* of *uniq*, which reads lines from *stdin*, discards successive duplicates, and writes the remainder to *stdout*. (Unix developers didn't like typing, so they abbreviated "unique.") Finally, the *stdout* of *uniq* is redirected to a file named `distinct_lines.txt`, and any error messages are redirected to another file named `error.log`. The three streams are numbered 0-2, so '2>'

means to redirect stream #2: *stderr* instead of the default output stream, *stdout*, which would be #1.

Programs that adhere to this model and which copy *stdin* to *stdout* while making some changes along the way are commonly called "filters." (*cat, sort,* and *uniq* are all filters, but a command like *ls*, which lists the contents of a directory, is not.)

This is not the only execution model for programs: an *event-driven* model is used when there is a graphical user interface, like a Windows, OS X, or mobile application. But for PHP, this "standard I/O" model is the one that's used.

When you include PHP code in a web page, the PHP processor acts as a filter. The web server (*Apache*) invokes the PHP processor after redirecting its *stdin* to come from your *.php* or *.xhtml* file, and its *stdout* to the network connection that goes back to the browser that requested the web page. The PHP processor writes text outside the `<?php ... ?>` blocks to *stdout* unchanged. Inside PHP blocks, *echo* statements and certain functions, like *exit(), var_dump(),* and *printf()* write to the *stdout* stream as well, while the rest of the PHP code gets executed without writing anything to *stdout*.

But *stderr*? Ah, there's the rub. The PHP processor writes both syntax errors and runtime error messages to *stderr.* During development, these error messages are invaluable for debugging, but in a production environment (a web site that is open to the world), these error messages can give a hacker valuable clues about the we site that would help in breaking into it. So during development, we want to be able to see these messages easily, but in production we still need to be able to see them (they tell us there is a bug we didn't know about), but we don't want to expose them to the world.

Both *Apache* and *PHP* use configuration files, which they read when the web server starts running, to control how to handle error messages. The *Apache* configuration file is called `httpd.conf`, typically located in a subdirectory where *Apache* was installed. For example, on a Windows computer, it might be found in the `C:\Program Files\Apache Software Foundation\Apache2.2\conf` directory. The PHP configuration file is called `php.ini`, and is typically in the directory where PHP was installed. For example, on a Windows computer, it might be found in the `C:\Program Files\PHP` directory. Anyone who can log into the server computer can view, but not change, both files. Doing so can be very instructive.

These two configuration files use different syntax rules from one another, but both generally take the form of lines that start with some sort of keyword, followed by a value.

Apache Configuration:

- `ErrorLog` The pathname of a file for the *stderr* streams for both *Apache* and *PHP*.

- `CustomLog` The pathname of a file where *Apache* writes a message for every web page it delivers to clients.

Unlike the PHP settings about to be described, these two *Apache* settings cannot be changed after the web server starts running. (Many other *Apache* settings can be customized for different directories, but not these two.)

PHP Configuration

PHP allows you to change several of the `php.ini` settings related to error messages from within your PHP code, as it executes. To make things concrete, here are the contents of a file that modifies some of these settings, with line-by-line explanations below.

Line No.	PHP Code
1	`<?php`
2	`  date_default_timezone_set('America/New_York');`
3	`  ini_set('error_reporting', E_ALL);`
4	`  ini_set('log_errors', 'On');`
5	`  ini_set('error_log', './error_log');`
6	`  ini_set('display_errors', 'On');`
7	`  assert_options(ASSERT_ACTIVE, 1);`
8	`  assert_options(ASSERT_WARNING, 1);`
9	`  include('index.php');`
10	`?>`

Lines 3-6 change the values of parameters that were set when *Apache* started running and the PHP module processed the `php.ini` file for the system. All PHP warnings and errors indicate problems in your code, so line 3 specifies that PHP should generate all possible messages. (There are situations where you might not want to have all these messages, but not when you are writing new code.) Line 4 says to write *stderr* to a log file, and line 5 says the log file is a file named `error_log` in the current directory instead of the *Apache* ErrorLog file. (See below for more information about setting up this log file.) Line 6 says to write error messages to *stdout* (i.e. to the client's browser) as well as to the log file. This setting in particular is one that should *never* be used in a production environment.

Lines 7 and 8 control the behavior of PHP *assert()* statements. Assertions are a way to make sure a program is working as expected, but are ignored by default because they make programs run a little slower. They are discussed further below.

Line 9 tells the PHP module to interject another file into its *stdin* stream at this point, in this case an index page that needs to be debugged.

By putting these 10 lines in a separate file (perhaps called `debug.php`), you can accomplish two things: (1) if you have to develop your code on a production site you can run your code in either development or production mode simply by using `debug.php` in the URL or not, and (2) if there are basic syntax errors in `index.php`, the PHP processor may "give up" before it can execute any calls to *ini_set()*; by putting them in this simple, separate, file, PHP executes the *ini_set()* calls before trying to parse `index.php`.

> *A previous version of this document suggested that the "giving up" issue had to do with whether HTTP headers had been sent or not, but that doesn't make sense.*

Setting Up To Log Errors

If you have direct access to *Apache's* ErrorLog file from the command line, a common debugging technique is to use the Unix *tail* command to view the contents of the file in real time: you can read the error messages as *Apache* and its PHP module write them. But by creating your own ErrorLog file from within your PHP code, as shown in lines 3-5 of the sample code above, provides several advantages:

- You can log error messages where you can see them even if, for some reason, *Apache* is configured to save them in a location not accessible to you, or even if *Apache* is configured not to log errors at all. (The latter is highly unlikely.)
- You can separate your own error messages out from all the other ones encountered on the system. The larger the number of web sites being hosted on the server there are, the more valuable this feature is.
- You can turn logging on and off dynamically rather than being constrained by the settings that were established in the `httpd.conf` and `php.ini` files when the server started running.
- You can add code to your site that writes messages to the log file to help you locate errors without cluttering up the system-wide log file. Two functions you can use to generate these debugging messages, *assert()* and *error_log ()*, are discussed further below.

Although it would be inappropriate to do this in a production environment, you can place a personal ErrorLog file in a place where you can view it from a web browser. That's what we did on line 5 of the sample code above: the file is in the same directory as the one that holds the index file being debugged. The log file is plain text, so telling a browser to use a URL that points to the log file will cause it to be displayed as-is with no HTML formatting.

No matter where the log file is located, there is a permissions issue to deal with: *Apache/PHP* must be able to write to it. As a developer, you have permission to write files to the server. And when *Apache* is running it has permission to read those files so it can deliver them over the Internet. But just as you don't have permission to write files just anywhere on the server, *Apache* can't write files just anywhere either. The ErrorLog file has to be set

up explicitly so that *Apache* can write to it. Without going into the full details of the way users and permissions are managed under different operating systems, this means that a *superuser* must grant write permission on the log file to the *Apache* user. For example, on an OS X system where *Apache* runs as a user named *_www*, this command could be used:

```
sudo chown _www error_log
```

This command changes the owner of the file named `error_log` to the *_www* user, which works because the owner of a file normally has write access to it.

An advantage of having a personal `ErrorLog` file not mentioned above is that during development, a bug can put your code in an endless loop: if that loop includes something that generates an error, the log file can get very big, very fast. (A recent case: 1.7GB in 5 seconds!) Even if your log file isn't too large, you are generally not interested in old parts of the file, and may want to clear the file so you don't have to scroll down to the bottom each time you look at it. The following web page will do that for you. It doesn't need any special permissions, just that the *Apache* "user" can write to `error_log`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Clear Error Log</title>
  </head>
  <body>
<?php
  echo "<h1>" . `du -sh error_log` . "</h1>\n";
  fopen('error_log', 'w+') or die("<p>Unable to clear error_log</p>\n");
  echo "<p>error_log cleared</p>";
  ?>
  </body>
</html>
```

The call to *fopen()* is what clears the file; the remainder of the code makes this into a web page that provides some feedback to you when you look at it.

Naming Your Error Log File

The conventional name for an `ErrorLog` file on *Apache* is `error_log`, but you can name it anything you want: `error.log`, `my_problems`, whatever. The point is not to confuse the name of the file with the name of the PHP setting for naming it, as on line 5 in the sample code.

Syntax Errors: Summary

The process of recognizing if a program is syntactically correct is called *parsing,* and the messages from PHP that deal with syntax errors generally contain the word "parse" in them. By setting up a custom error log and using the sample code at the beginning of this section, you can get to see any syntax error messages generated by the PHP processor, even in a production environment.

Run-Time Errors

it

The PHP error_log() Function

There is a function you can call from within your PHP code to write whatever messages you like to your `ErrorLog` file and, as might be able to guess, it's name is *error_log()*. There are several optional parameters you can pass to this function (including a way to get the message emailed to you), but the simplest version is just to pass it a string that gives you the information you want to see. Note that string interpolation and concatenation can be your friends here:

```
error_log("The value of a_variable is $a_variable on line " .
 __LINE__ . " of " . __FILE__);
```

Assertions

The *assert()* function takes a string as its argument. The PHP processor executes the string as PHP code, and either "succeeds" or "fails" depending on whether the result of executing the code yields a value of *true* or *false* respectively. If the assertion succeeds, nothing happens, but if the assertion fails, the processor takes the action established by the *assert_options()* function: if `ASSERT_WARN` is set, a message gets written to the `ErrorLog`, but if `ASSERT_BAIL` is set instead, the PHP processor "bails out" at that point: it stops processing the file. The nice feature of assertions compared to the use of *echo, var_dump(), or error_log()* function calls is that they can be left in the code without causing any overhead (because they will be ignored) by setting the `ASSERT_ACTIVE` option off.

A common use for assert processing is to test for program correctness by establishing *preconditions* (before this code gets executed, certain conditions must be true) and *postconditions* (after this code executes, certain conditions must be true. For example, if a function receives an argument that is supposed to be the name of a directory, the function definition could start with:

```
function my_function($pathname_of_a_directory)
{
  assert('is_dir($pathname_of_a_directory)');
  // remainder of the function definition …
}
```