PHP on the Command Line – Part 1 Article

(http://www.sitepoint.com/author/harry-fuecks/)

Harry Fuecks (http://www.sitepoint.com/author/harry-fuecks/)

Published June 16, 2004

We all know PHP is a great language for building Web applications with but we all *also* know there's a lot more to running a serious Website than just the user interface.

There are back ups to be performed, log files to be analysed, data to be pruned and all those essential but mundane tasks that happen in the background. Typically, such routine administration tasks are prime targets for automation and, if you've spent any time with UNIX-based systems, you've probably run into the notion of shell scripts and cron.

What may or may not surprise you is that PHP does a pretty good job *off* the Web as well. If, after reading Getting Started with PEAR (http://www.sitepoint.com/article/getting-started-with-pear), you installed the PEAR package manager, you've already had a taste of what PHP can do on the command line.

In this two-part series, I'll be looking at PHP's Command Line Interface (CLI) in detail. This first part will concentrate on the fundamentals; the input and output to a command line script. The second will look further at what PEAR has to offer for the command line, as well as some of PHP's less charted functionality.

Today’s Command Line Arguments

- Introducing the CLI: what it’s all about
- Setting Up: checking your CLI
- Hello World: your first CLI script
- Stream In, Stream Out: dealing with input and output
- Multiple Choice: interactive user input
- Errors and Pipes: good manners
- Coping with Arguments: more input
- PEAR::Console_Getopt: managing arguments
- Compatibility: keeping everyone happy

Also, you might like to download the code archive (http://www.sitepoint.com/examples/phpcli/phpcli1_code.zip) that contains all the examples we’ll see in this tutorial.

Introducing the CLI

PHP’s Command Line Interface first showed up in PHP 4.2.0, as experimental functionality. With PHP 4.3.0 the CLI became an official SAPI (Server API). What that means in practical terms is that the CLI version of PHP now comes as a separate PHP binary (executable), which you can use to run scripts on the command line, as well as providing “access points” to the rest of the world, much like the `$_GET` and `$_POST` variables that give you access to incoming data over the Web.

Before I go any further I should mention that, as you’re probably aware, PHP is not the only choice for writing command line scripts. Both Perl and Python, to name just two, are widely used for writing command line applications and, in many cases, make better a choice than PHP. They provide a mature set of tools for common problems and, typically, better performance.

So, why use PHP? An obvious — and very good — reason may simply be that you know PHP better than the alternatives. Less obvious is that, if you’re developing a Web application in PHP, writing supporting scripts in another language can lead to extra headaches — even if you’re confident in both. There’s both the human aspect of having to switch programming “mind sets”, and the overhead of having to support two platforms and the potential missed opportunities for re-use; data access logic may need to be implemented twice, for example.

What’s more, if you’re building a PHP application that will be used by “unknown” third parties, requiring them to set up other platforms in addition to PHP is likely to cause frustration, particularly if they’re using Windows. Although PHP may not be the best choice overall for writing command line utilities, it’s not a bad choice, either, and, in building tools to support an application that’s already written in PHP, it makes sense to stick to a single platform.

What types of problems are suited to being solved with a command line script? Obvious choices are anything that relates to the administration of the server on which your Web application runs, such as backups, purging and archiving old data, and analysing log files — perhaps sending an email alert should critical errors occur. These are all tasks that are suited to automation with cron (http://www.unixgeeks.org/security/newbie/unix/cron-1.html), the “intended user” of the scripts being another script or application.

It needn’t necessarily stop there, though. You might consider writing a command line “installer” for your Web application, to take care of copying PHP scripts to the right locations, setting up the correct file system permissions, loading the database schema and, for the intrepid, managing upgrades to the application. When executing PHP scripts over the Web via Apache, you typically don’t have rights to complete tasks such as changing file system permissions â€“ plus, you don’t necessarily want to expose to the Web code that’s capable of making radical changes. However, run from the command line, using a normal user account, the script can be kept offline while providing you the necessary rights to copy files, change permissions and so on.

The advantage of command line scripts is that they’re usually easier to develop and, more importantly, quicker to use then Web-based of desktop GUIs. As such, they can also play a big part in your development process, helping you perform “build” tasks quickly and easily, such as generating API documentation with phpDocumentor (http://www.phpdoc.org) or running your Simple Test (http://www.lastcraft.com/simple_test.php) suite. Phing (http://phing.info/wiki/index.php) (a build tool based on Apache’s ANT) and rephlux (http://sourceforge.net/projects/rephlux) (a continuous integration tool based on CruiseControl (http://cruisecontrol.sourceforge.net/)) provide frameworks to help you automate your development process.

Setting Up

I’ve already covered the basics of setting up PHP so you can run it from the command line in Getting Started with PEAR, Page 2 (http://www.sitepoint.com/article/getting-started-with-pear/2). It’s also worth being aware of what the manual has to say about Using PHP from the command line (http://www.php.net/manual/en/features.commandline.php).

When writing your own command line scripts (as opposed to using the PEAR Package Manager), you need to be conscious of which PHP binary you're using (your host, for example, may have set you up with the CGI binary).

A quick way to test this is type the following from the command line.

```
 $ php -v
```

This should result in something like the following message:

```
 PHP 4.3.6 (cli) (built: Apr 19 2004 10:02:14)
Copyright (c) 1997-2004 The PHP Group
Zend Engine v1.3.0, Copyright (c) 1998-2004 Zend Technologies
```

Note the `(cli)` above — if instead you see " `(cgi)` ", don't worry. The CGI binary can be told to behave like the CLI binary when it's used, but you need to be aware there's a difference. I'll look at solving compatibility issues between the two, as well as older PHP versions, at the end of this article.

Hello World

In its most basic form, the CLI interface can be used just like any PHP script. Here's an example:

```
 <?php
print "Hello World!";
?>
```

Filename: helloworld.php

If you're running on UNIX based system, you first need to make this script executable with:

```
 chmod +x helloworld.php
```

You can execute this script by typing:

```
 php helloworld.php
```

You can avoid specifically having to call the php binary first on UNIX-based systems by placing a SheBang (http://c2.com/cgi/wiki?SheBang) right at the start of the script. This "points" the shell at the binary that should be used to execute the script. Here's an example:

```
 #!/usr/local/bin/php
<?php
print "Hello World!";
?>
```

Note the path to your PHP executable may vary, `/usr/bin/php` being another common location.

I can now execute this script with:

```
 $ helloworld.php
```

On UNIX-based systems, you may also consider dropping the .php file extension, as it's not required in order to execute the script, and because there is some benefit to be had by doing so; if, later, you choose to replace it with a script that's written in another language, while maintaining the functionality, there'll be no confusion.

On Windows, you can achieve almost the same thing by associating the PHP file extension with the PHP binary. You'll find this explained at the end of Replacing Perl Scripts with PHP Scripts (http://www.phpbuilder.com/columns/jayesh20021111.php3).

Here, I'll stick to using a .php extension, and I'll assume you'll be executing the scripts by first calling the PHP binary, as this makes the most effective cross-platform approach.

Stream In, Stream Out

When it comes to displaying output from your command line scripts, using the normal echo (http://www.php.net/echo) or print (http://www.php.net/print) commands is not the best way to go, for two reasons.

First, on Windows, see what happens when you execute the following from a command prompt:

```
 <?php
$i = 0;
while ( $i < 10 ) {
print $i."n";
sleep(1);
$i++;
}
?>
```

Filename: printproblem.php

Instead of seeing the numbers displayed at the moment they are printed, with one-second intervals between them, they're all flushed out at once when the script terminates (note that running the same script on Linux results in the numbers being displayed as they happen).

The other reason is more a question of good practice. PHP (the underlying shell, in fact) allows you to direct normal output that's meant for the user to what's known as the "standard out", while diverting any error messages to the "standard error". The advantage in making this division is that it allows you to log errors separately from messages generated by the normal, smooth operation of the script, which can be very useful when running batch jobs.

Part of PHP's CLI interface are three "streams" with which you can interact in more or less the same way as you would a file resource returned from fopen() (http://www.php.net/fopen). The streams are identified with the strings:

```
 php://stdin (read)
php://stdout (write)
php://stderr (write)
```

With the PHP 4.3.0+ CLI binary, these three streams are automatically available, identified with the constants STDIN, STDOUT and STDERR respectively. Here's how I can use the STDOUT to fix the above script so that it behaves correctly on Windows:

```
 <?php
$i = 0;
while ( $i < 10 ) {
// Write some output
fwrite(STDOUT, $i."n");
sleep(1);
$i++;
}
?>
```

Filename: streamout.php

The count should now be displayed at the moment at which fwrite() (http://www.php.net/fwrite) is called.

Note that, on Windows, PHP takes care of linefeed characters for you, so that they display correctly (you weren't trying to use `<br>` , were you?).

To be able to read input from the script's user, you can use STDIN combined with fgets() (http://www.php.net/fgets), fread() (http://www.php.net/fread), fscanf() (http://www.php.net/fscanf) or fgetc() (http://www.php.net/fgetc). For example:

```
 <?php
fwrite(STDOUT, "Please enter your namen");

// Read the input
$name = fgets(STDIN);

fwrite(STDOUT, “Hello $name”);

// Exit correctly
exit(0);
?>
```

Filename: nameplease.php

What happens when the script is executed? Well, when the interpreter reaches the `fgets()` command (it sees the code reading from STDIN), execution pauses and waits for the user to hit enter. On pressing return, `fgets()` receives everything the user typed from the point at which execution paused, until the moment he or she pressed return. This includes the return linefeed character — see what happens if you display the contents of the `$name` variable after passing it through nl2br() (http://www.php.net/nl2br).

This “pause / continue” behaviour is actually controlled by the terminal (the command prompt). It works on a line-by-line basis and is known as Canonical Mode Input Processing (http://www.opengroup.org/onlinepubs/007908799/xbd/termios.html#tag_008_001_006).

Note that, on UNIX only, it's possible to have your script process input in Non-canonical Mode (http://www.opengroup.org/onlinepubs/007908799/xbd/termios.html#tag_008_001_007) using the pcntl_signal() (http://www.php.net/pcntl_signal), which allows you to respond directly to particular key sequences.

Note also that I've started to use the exit() (http://www.php.net/exit) command at the end of the script, passing it an integer value. This is another question of “good manners”. Common practice on UNIX-based systems is for a script to return a status code when execution halts, zero being the standard for denoting the “All OK”. Other codes (integers between 1 and 254) are used to identify problems that caused the script to terminate prematurely (it’s up to you to define the error codes for you script, which is commonly achieved by defining a set of constants at the start). If another command line script (perhaps written in another language) is used to execute your script, it will rely on the status code returned from exit() to determine whether you script ran successfully or not. For some more thoughts on the subject, see the Advanced Bash-Scripting Guide (http://www.faqs.org/docs/abs/HTML/) on Exit and Exit Status (http://www.faqs.org/docs/abs/HTML/exit-status.html).

Go to page: 1 (/php-command-line-1) | 2 (/php-command-line-1-2/) | 3 (/php-command-line-1-3/)

(http://www.sitepoint.com/author/harry-fuecks/)

Harry Fuecks (http://www.sitepoint.com/author/harry-fuecks/)

(https://twitter.com/hfuecks)
(https://plus.google.com/105166600541125352541/)

You might also like:

[Command line PHP using Symfony Console (http://www.sitepoint.com/command-line-php-using-symfony-console/)](http://www.sitepoint.com/command-line-php-using-symfony-console/)

Comments for this thread are now closed. ×

2 Comments SitePoint 1 Login

Sort by Best Share Favorite

bill • 3 years ago

Is your name really harry fuecks?

2 • Share ›

bill • 3 years ago

Otherwise, great article!

1 • Share ›

Subscribe Add Disqus to your site DISQUS

About

[About us (/about-us/)](/about-us/)
[Advertise (/advertising)](/advertising)
[Legals (/legals)](/legals)
[Feedback (mailto:feedback@sitepoint.com)](mailto:feedback@sitepoint.com)
[Write for Us (/write-for-us)](/write-for-us)

Our Sites

[Learnable (https://learnable.com)](https://learnable.com)
[Reference (http://reference.sitepoint.com)](http://reference.sitepoint.com)
[Hosting Reviews (/hosting-reviews/)](/hosting-reviews/)
[Web Foundations (/web-foundations/)](/web-foundations/)

Connect

(/feed) (/newsletter) (https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (https://plus.google.com/+sitepoint)