Executing PHP files ¶

There are three different ways of supplying the CLI SAPI with PHP code to be executed:

1. Tell PHP to execute a certain file.

   ```
   $ php my_script.php

   $ php -f my_script.php
   ```

   Both ways (whether using the **-f** switch or not) execute the file *my_script.php*. Note that there is no restriction on which files can be executed; in particular, the filename is not required have a *.php* extension.

   > **Nota**:
   >
   > If arguments need to be passed to the script when using **-f** , the first argument must be --.

2. Pass the PHP code to execute directly on the command line.

   ```
   $ php -r 'print_r(get_defined_constants());'
   ```

   Special care has to be taken with regard to shell variable substitution and usage of quotes.

   > **Nota**:
   >
   > Read the example carefully: there are no beginning or ending tags! The **-r** switch simply does not need them, and using them will lead to a parse error.

3. Provide the PHP code to execute via standard input (*stdin*).

   This gives the powerful ability to create PHP code dynamically and feed it to the binary, as shown in this (fictional) example:

   ```
   $ some_application | some_filter | php | sort -u > final_output.txt
   ```

You cannot combine any of the three ways to execute code.

As with every shell application, the PHP binary accepts a number of arguments; however, the PHP script can also receive further arguments. The number of arguments that can be passed to your script is not limited by PHP (and although the shell has a limit to the number of characters which can be passed, this is not in general likely to be hit). The arguments passed to the script are available in the global array *$argv*. The first index (zero) always contains the name of the script as called from the command line. Note that, if the code is executed in-line using the command line switch **-r** , the value of *$argv[0]* will be just a dash (-). The same is true if the code is executed via a pipe from *STDIN*.

A second global variable, *$argc*, contains the number of elements in the *$argv* array (*not* the number of arguments passed to the script).

As long as the arguments to be passed to the script do not start with the - character, there's nothing special to watch out for. Passing an argument to the script which starts with a - will cause trouble because the PHP interpreter thinks it has to handle it itself, even before executing the script. To prevent this, use the argument

list separator --. After this separator has been parsed by PHP, every following argument is passed untouched to the script.

```
# This will not execute the given code but will show the PHP usage
$ php -r 'var_dump($argv);' -h
Usage: php [options] [-f] <file> [args...]
[...]

# This will pass the '-h' argument to the script and prevent PHP from showing its usage
$ php -r 'var_dump($argv);' -- -h
array(2) {
  [0]=>
  string(1) "-"
  [1]=>
  string(2) "-h"
}
```

However, on Unix systems there's another way of using PHP for shell scripting: make the first line of the script start with *#!/usr/bin/php* (or whatever the path to your PHP CLI binary is if different). The rest of the file should contain normal PHP code within the usual PHP starting and end tags. Once the execution attributes of the file are set appropriately (e.g. **chmod +x test**), the script can be executed like any other shell or perl script:

Exemplo #1 Execute PHP script as shell script

```
#!/usr/bin/php
<?php
var_dump($argv);
?>
```

Assuming this file is named *test* in the current directory, it is now possible to do the following:

```
$ chmod +x test
$ ./test -h -- foo
array(4) {
  [0]=>
  string(6) "./test"
  [1]=>
  string(2) "-h"
  [2]=>
  string(2) "--"
  [3]=>
  string(3) "foo"
}
```

As can be seen, in this case no special care needs to be taken when passing parameters starting with -.

The PHP executable can be used to run PHP scripts absolutely independent of the web server. On Unix systems, the special *#!* (or "shebang") first line should be added to PHP scripts so that the system can automatically tell which program should run the script. On Windows platforms, it's possible to associate *php.exe* with the double click option of the *.php* extension, or a batch file can be created to run scripts through PHP. The special shebang first line for Unix does no harm on Windows (as it's formatted as a PHP comment), so cross platform programs can be written by including it. A simple example of writing a command line PHP program is shown below.

Exemplo #2 Script intended to be run from command line (script.php)

```
#!/usr/bin/php
<?php

if ($argc != 2 || in_array($argv[1], array('--help', '-help', '-h', '-?'))) {
?>

This is a command line PHP script with one option.

  Usage:
  <?php echo $argv[0]; ?> <option>

  <option> can be some word you would like
  to print out. With the --help, -help, -h,
  or -? options, you can get this help.

<?php
} else {
    echo $argv[1];
}
?>
```

The script above includes the Unix shebang first line to indicate that this file should be run by PHP. We are working with a CLI version here, so no HTTP headers will be output.

The program first checks that there is the required one argument (in addition to the script name, which is also counted). If not, or if the argument was **--help** , **-help** , **-h** or **-?** , the help message is printed out, using *$argv[0]* to dynamically print the script name as typed on the command line. Otherwise, the argument is echoed out exactly as received.

To run the above script on Unix, it must be made executable, and called simply as **script.php echothis** or **script.php -h**. On Windows, a batch file similar to the following can be created for this task:

Exemplo #3 Batch file to run a command line PHP script (script.bat)

```
@echo OFF
"C:\php\php.exe" script.php %*
```

Assuming the above program is named *script.php*, and the CLI *php.exe* is in *C:\php\php.exe*, this batch file will run it, passing on all appended options: **script.bat echothis** or **script.bat -h**.

See also the Readline extension documentation for more functions which can be used to enhance command line applications in PHP.

On Windows, PHP can be configured to run without the need to supply the *C:\php\php.exe* or the *.php* extension, as described in Command Line PHP on Microsoft Windows.

add a note

User Contributed Notes 4 notes

up
down

9

php at richardneill dot org ¶

1 year ago

On Linux, the shebang (#!) line is parsed by the kernel into at most two parts. For example:

```
1:  #!/usr/bin/php
2:  #!/usr/bin/env  php
3:  #!/usr/bin/php -n
4:  #!/usr/bin/php -ddisplay_errors=E_ALL
5:  #!/usr/bin/php -n -ddisplay_errors=E_ALL
```

1. is the standard way to start a script. (compare "#!/bin/bash".)

2. uses "env" to find where PHP is installed: it might be elsewhere in the $PATH, such as /usr/local/bin.

3. if you don't need to use env, you can pass ONE parameter here. For example, to ignore the system's PHP.ini, and go with the defaults, use "-n". (See "man php".)

4. or, you can set exactly one configuration variable. I recommend this one, because display_errors actually takes effect if it is set here. Otherwise, the only place you can enable it is system-wide in php.ini. If you try to use ini_set() in your script itself, it's too late: if your script has a parse error, it will silently die.

5. This will not (as of 2013) work on Linux. It acts as if the whole string, "-n -ddisplay_errors=E_ALL" were a single argument. But in BSD, the shebang line can take more than 2 arguments, and so it may work as intended.

Summary: use (2) for maximum portability, and (4) for maximum debugging.

up
down

5

petruzanautico at yah00 dot com dot ar ¶

3 years ago

As you can't use -r and -f together, you can circumvent this by doing the following:

```
php -r '$foo = 678; include("your_script.php");'
```

up
down

2

spencer at aninternetpresence dot net ¶

3 years ago

If you are running the CLI on Windows and use the "-r" option, be sure to enclose your PHP code in double (not single) quotes. Otherwise, your code will not run.

up
down

-1

andrew-by at hotmail dot com ¶

2 years ago

If you want your batch file's window stay opened after executing the PHP script just add "pause" at the last line. So it should look like:

```
@echo OFF
"C:\php\php.exe" script.php %*
pause
```

add a note