PHP 5 Documentation Group

Edited by
Gabor Hojsty

DG

the definitive
PHP 5 REFERENCE
in print

Installation, Configuration, Language Reference, Function Reference & More!

PHP 5 Documentation Group

Mehdi Achour | Friedhelm Betz | Antony Dovgal

Nuno Lopes | Philip Olson | Georg Richter

Damien Seguy | Jakub Vrana | *and several others*

Edited by **Gabor Hojtsy**

Lnez Caldwell
PUBLISHING
Toronto and Little Ridge

Table of Contents

V. Features

Appendices

Chapter 10. Basic Syntax

10.1 Escaping from HTML

When PHP parses a file, it looks for opening and closing tags, which tell PHP to start and stop interpreting the code between them. Parsing in this manner allows PHP to be embedded in all sorts of different documents, as everything outside of a pair of opening and closing tags is ignored by the PHP parser. Most of the time you will see PHP embedded in HTML documents, as in this example:

```
<p>This is going to be ignored.</p>
<?php echo 'While this is going to be parsed.'; ?>
<p>This will also be ignored.</p>
```

You can also use more advanced structures:

Example 10-1. Advanced Escaping

```
<?php
if ($expression) {
    ?>
    <strong>This is true.</strong>
    <?php
} else {
    ?>
    <strong>This is false.</strong>
    <?php
}
?>
```

This works as expected, because when PHP hits the ?> closing tags, it simply starts outputting whatever it finds until it hits another opening tag. The example given here is contrived, of course, but for outputting large blocks of text, dropping out of PHP parsing mode is generally more efficient than sending all of the text through `echo()` or `print()`.

There are four different pairs of opening and closing tags which can be used in PHP. Two of those, `<?php ?>` and `<script language="php"> </script>`, are always available. The other two are short tags and ASP style tags, and can be turned on and off from the php.ini configuration file. As such, while some people find short tags and ASP style tags convenient, they are less portable, and generally not recommended.

Note: *If you are embedding* PHP *within* XML *or* XHTML *you will need to use the* `<?php ?>` *tags to remain compliant with standards.*

Example 10-2. PHP Opening and Closing Tags

```
    <?php echo 'if you want to serve XHTML or XML documents, ¬
1.  do like this'; ?>

    <script language="php">
2.      echo 'some editors (like FrontPage) don't like ¬
        processing instructions';
    </script>

    <? echo 'this is the simplest, an SGML processing ¬
3.  instruction'; ?>
    <?= expression ?> This is a shortcut for "<? echo ¬
    expression ?>"

    <% echo 'You may optionally use ASP-style tags'; %>
    <%= $variable; # This is a shortcut for "<% echo . . ." %>
```

While the tags seen in examples one and two are both always available, example one is the most commonly used, and recommended, of the two.

Short tags (example three) are only available when they are enabled via the short_open_tag php.ini configuration file directive, or if PHP was configured with the --enable-short-tags option.

Note: *If you are using* PHP 3 *you may also enable short tags via the* `short_tags()` *function. This is only available in* PHP 3*!*

ASP style tags (example four) are only available when they are enabled via the asp_tags php.ini configuration file directive.

Note: *Support for* ASP *tags was added in* 3.0.4.

Note: *Using short tags should be avoided when developing applications or libraries that are meant for redistribution, or deployment on* PHP *servers which are not under your control, because short tags may not be supported on the target server. For portable, redistributable code, be sure not to use short tags.*

10.2 Instruction separation

As in C or Perl, PHP requires instructions to be terminated with a semicolon at the end of each statement. The closing tag of a block of PHP code automatically implies a semicolon; you do not need to have a semicolon terminating the last line of a PHP block. The closing tag for the block will include the immediately trailing newline if one is present.

```
<?php
    echo 'This is a test';
?>

<?php echo 'This is a test' ?>

<?php echo 'We omitted the last closing tag';
```

Note: *The closing tag of a* PHP *block at the end of a file is optional, and in some cases omitting it is helpful when using* `include()` *or* `require()`*, so unwanted whitespace will not occur at the end of files, and you will still be able to add headers to the response later. It is also handy if you use output buffering, and would not like to see added unwanted whitespace at the end of the parts generated by the included files.*

10.3 Comments

PHP supports C, C++ and Unix shell-style (Perl style) comments. For example:

```
<?php
    echo 'This is a test'; // This is a one-line ¬
    c++ style comment
    /* This is a multi line comment
       yet another line of comment */
    echo 'This is yet another test';
    echo 'One Final Test'; # This is a one-line ¬
    shell-style comment
?>
```

The "one-line" comment styles only comment to the end of the line or the current block of PHP code, whichever comes first. This means that HTML code after `// ... ?>` or `# ... ?>` *will* be printed: `?>` breaks out of PHP mode and returns to HTML mode, and `//` or `#` cannot influence that. If the `asp_tags` configuration directive is enabled, it behaves the same with `// %>` and `# %>`. However, the `</script>` tag doesn't break out of PHP mode in a one-line comment.

```
<h1>This is an <?php # echo 'simple';?> example.</h1>
<p>The header above will say 'This is an example'.</p>
```

C style comments end at the first `*/` encountered. Make sure you don't nest C style comments. It is easy to make this mistake if you are trying to comment out a large block of code.

```
<?php
/*
    echo 'This is a test'; /* This comment will ¬
    cause a problem */
*/
?>
```

Appendix K. List of Reserved Words

The following is a listing of predefined identifiers in PHP. None of the identifiers listed here should be used as identifiers in any of your scripts. These lists include keywords and predefined variable, constant, and class names. These lists are neither exhaustive or complete.

List of Keywords

These words have special meaning in PHP. Some of them represent things which look like functions, some look like constants, and so on—but they're not, really: they are language constructs. You cannot use any of the following words as constants, class names, function or method names. Using them as variable names is generally OK, but could lead to confusion.

`and`, `or`, `xor`, `__FILE__`, `exception` (PHP 5), `__LINE__`, `array()`, `as`, `break`, `case`, `class`, `const`, `continue`, `declare`, `default`, `die()`, `do`, `echo()`, `else`, `elseif`, `empty()`, `enddeclare`, `endfor`, `endforeach`, `endif`, `endswitch`, `endwhile`, `eval()`, `exit()`, `extends`, `for`, `foreach`, `function`, `global`, `if`, `include()`, `include_once()`, `isset()`, `list()`, `new`, `print()`, `require()`, `require_once()`, `return()`, `static`, `switch`, `unset()`, `use`, `var`, `while`, `__FUNCTION__`, `__CLASS__`, `__METHOD__`, `final` (PHP 5), `php_user_filter` (PHP 5), `interface` (PHP 5), `implements` (PHP 5), `extends`, `public` (PHP 5), `private` (PHP 5), `protected` (PHP 5), `abstract` (PHP 5), `clone` (PHP 5), `try` (PHP 5), `catch` (PHP 5), `throw` (PHP 5), `cfunction` (PHP 4 only), `old_function` (PHP 4 only), `this` (PHP 5 only).

Predefined Variables

Since PHP 4.1.0, the preferred method for retrieving external variables is with the superglobals mentioned below. Before this time, people relied on either `register_globals` or the long predefined PHP arrays (`$HTTP_*_VARS`). As of PHP 5.0.0, the long PHP predefined variable arrays may be disabled with the `register_long_arrays` directive.

Server variables: $_SERVER

Note: *Introduced in 4.1.0. In earlier versions, use* `$HTTP_SERVER_VARS`.

`$_SERVER` is an array containing information such as headers, paths, and script locations. The entries in this array are created by the webserver. There is no guarantee that every webserver will provide any of these; servers may omit some, or provide others not listed here. That said, a large number of these variables are accounted for in the CGI 1.1 specification, so you should be able to expect those.

This is a '*superglobal*', or automatic global, variable. This simply means that it is available in all scopes throughout a script. You don't need to do a global `$_SERVER`; to access it within functions or methods, as you do with `$HTTP_SERVER_VARS`.

`$HTTP_SERVER_VARS` contains the same initial information, but is not an autoglobal. (Note that `$HTTP_SERVER_VARS` and `$_SERVER` are different variables and that PHP handles them as such).

If the `register_globals` directive is set, then these variables will also be made available in the global scope of the script; i.e., separate from the `$_SERVER` and `$HTTP_SERVER_VARS` arrays. For related information, see the security chapter titled **Using Register Globals**. These individual globals are not autoglobals.

You may or may not find any of the following elements in `$_SERVER`. Note that few, if any, of these will be available (or indeed have any meaning) if running PHP on the command line:

'PHP_SELF' The filename of the currently executing script, relative to the document root. For instance, `$_SERVER['PHP_SELF']` in a script at the address http://example.com/test.php/foo.bar would be /test.php/foo.bar. The `__FILE__` constant contains the full path and filename of the current (i.e. included) file. If PHP is running as a command-line processor this variable contains the script name since PHP 4.3.0. Previously it was not available.

'argv' Array of arguments passed to the script. When the script is run on the command line, this gives C-style access to the command line parameters. When called via the GET method, this will contain the query string.

'argc' Contains the number of command line parameters passed to the script (if run on the command line).

'GATEWAY_INTERFACE' What revision of the CGI specification the server is using; i.e. `'CGI/1.1'`.

'SERVER_NAME' The name of the server host under which the current script is executing. If the script is running on a virtual host, this will be the value defined for that virtual host.

'SERVER_SOFTWARE' Server identification string, given in the headers when responding to requests.

'SERVER_PROTOCOL' Name and revision of the information protocol via which the page was requested; i.e. `'HTTP/1.0'`;

'REQUEST_METHOD' Which request method was used to access the page; i.e. `'GET'`, `'HEAD'`, `'POST'`, `'PUT'`. ***Note:*** *php script is terminated after sending headers (it means after producing any output without output buffering), if the request method was* HEAD.

'REQUEST_TIME' The timestamp of the start of the request. Available since PHP 5.1.0.

'QUERY_STRING' The query string, if any, via which the page was accessed.

'DOCUMENT_ROOT' The document root directory under which the current script is executing, as defined in the server's configuration file.

'HTTP_ACCEPT' Contents of the `Accept: header` from the current request, if there is one.

'HTTP_ACCEPT_CHARSET' Contents of the `Accept-Charset: header` from the current request, if there is one. Example: `'iso-8859-1,*,utf-8'`.

'HTTP_ACCEPT_ENCODING' Contents of the `Accept-Encoding: header` from the current request, if there is one. Example: `'gzip'`.

'HTTP_ACCEPT_LANGUAGE' Contents of the `Accept-Language: header` from the current request, if there is one. Example: `'en'`.

'HTTP_CONNECTION' Contents of the `Connection: header` from the current request, if there is one. Example: `'Keep-Alive'`.

'HTTP_HOST' Contents of the `Host: header` from the current request, if there is one.

'HTTP_REFERER' The address of the page (if any) which referred the user agent to the current page. This is set by the user agent. Not all user agents will set this, and some provide the ability to modify `HTTP_REFERER` as a feature. In short, it cannot really be trusted.

'HTTP_USER_AGENT' Contents of the `User-Agent: header` from the current request, if there is one. This is a string denoting the user agent being which is accessing the page. A typical example is: `Mozilla/4.5 [en] (X11; U; Linux 2.2.9 i586)`. Among other things, you can use this value with `get_browser()` to tailor your page's output to the capabilities of the user agent.

'HTTPS' Set to a non-empty value if the script was queried through the HTTPS protocol.

'REMOTE_ADDR' The IP address from which the user is viewing the current page.