Shortcode

- The Shortcode API
- History
- Overview
  - Handling Attributes
  - Output
  - Enclosing vs self-closing shortcodes
  - Other features in brief
  - Function reference
  - Limitations
    - Nested Shortcodes
    - Unregistered Names
    - Unclosed Shortcodes
    - Hyphens
    - Square Brackets
    - HTML
    - Registration Count
  - Formal Syntax
    - Names
    - Attributes
    - Self-Closing
    - Escaping
- External Resources
- Default Shortcodes
- Shortcode API functions list

The Shortcode API

The **Shortcode API** is a simple set of functions for creating WordPress shortcodes for use in posts and pages. For instance, the following shortcode (in the body of a post or page) would add a photo gallery of images attached to that post or page: `[ gallery ]`

The API enables plugin developers to create special kinds of content (e.g. forms, content generators) that users can attach to certain pages by adding the corresponding shortcode into the page text.

The Shortcode API makes it easy to create shortcodes that support attributes like this:

```
[ gallery id="123" size="medium" ]
```

The API handles all the tricky parsing, eliminating the need for writing a custom regular expression for each shortcode. Helper functions are included for setting and fetching default attributes. The API supports both self-closing and enclosing shortcodes.

As a quick start for those in a hurry, here's a minimal example of the PHP code required to create a shortcode:

```
// [foobar]
function wporg_foobar_func( $atts ) {
	return "foo and bar";
}
add_shortcode( 'foobar', 'wporg_foobar_func' );
```

This will create `[foobar]` shortcode that returns as: foo and bar

With attributes:

```
// [bartag foo="foo-value"]
function bartag_func( $atts ) {
	$a = shortcode_atts( array(
		'foo' => 'something',
		'bar' => 'something else',
	), $atts );

	return "foo = {$a['foo']}";
}
add_shortcode( 'bartag', 'bartag_func' );
```

This creates a `[bartag]` shortcode that supports two attributes: "foo" and "bar". Both attributes are optional and will take on default options `[foo="something" bar="something else"]` if they are not

provided. The shortcode will return as `foo = {the value of the foo attribute}`.

History

The Shortcode API was introduced in WordPress 2.5.

Overview

Shortcodes are written by providing a handler function. Shortcode handlers are broadly similar to WordPress filters: they accept parameters (attributes) and return a result (the shortcode output).

Shortcode names should be all lowercase and use all letters, but numbers and underscores should work fine too. Be wary of using hyphens (dashes), you'll be better off not using them.

The `add_shortcode` function is used to register a shortcode handler. It takes two parameters: the shortcode name (the string used in a post body), and the callback function name.

Three parameters are passed to the shortcode callback function. You can choose to use any number of them including none of them.

- `$atts`: an associative array of attributes, or an empty string if no attributes are given
- `$content`: the enclosed content (if the shortcode is used in its enclosing form)
- `$tag`: the shortcode tag, useful for shared callback functions

The API call to register the shortcode handler would look something like this:

```
add_shortcode( 'wporgshortcode', 'wporg_shortcode_handler' );
```

When the_content is displayed, the shortcode API will parse any registered shortcodes such as `[myshortcode]`, separate and parse the attributes and content, if any, and pass them the corresponding shortcode handler function. Any string *returned* (not echoed) by the shortcode handler will be inserted into the post body in place of the shortcode itself.

Shortcode attributes are entered like this:

`[wporgshortcode foo="bar" bar="bing"]`

These attributes will be converted into an associative array like the following, passed to the handler function as its `$atts` parameter:

```
array( 'foo' => 'bar', 'bar' => 'bing' )
```

The array keys are the attribute names; array values are the corresponding attribute values. In addition, the zeroeth entry (`$atts[0]`) will hold the string that matched the shortcode regex, but ONLY IF that is different from the callback name.

Handling Attributes

The raw `$atts` array may include any arbitrary attributes that are specified by the user. (In addition, the zeroeth entry of the array may contain the string that was recognized by the regex; see the note below.)

In order to help set default values for missing attributes, and eliminate any attributes that are not recognized by your shortcode, the API provides a shortcode_atts() function.

`shortcode_atts()` resembles the `wp_parse_args` function, but has some important differences. Its parameters are:

```
shortcode_atts( $defaults_array, $atts );
```

Both parameters are required. `$defaults_array` is an associative array that specifies the recognized attribute names and their default values. `$atts` is the raw attributes array as passed into your shortcode handler. `shortcode_atts()` will return a normalized array containing all of the keys from the `$defaults_array`, filled in with values from the `$atts` array if present. For example:

```
$a = shortcode_atts( array(
    'title' => 'My Title',
    'foo' => 123,
), $atts );
```

If `$atts` were to contain `array( 'foo' => 456, 'bar' => 'something' )`, the resulting `$a` would be `array( 'title' => 'My Title', 'foo' => 456 )`. The value of `$atts['foo']` overrides the default of 123. `$atts['title']` is not set, so the default ‘My Title’ is used. There is no ‘bar’ item in the defaults array, so it is not included in the result.

Attribute names are always converted to lowercase before they are passed into the handler function. Values are untouched. `[myshortcode FOO="BAR"]` produces `$atts = array( 'foo' => 'BAR' )`.

A suggested code idiom for declaring defaults and parsing attributes in a shortcode handler is as follows:

```
function wporg_shortcode_handler( $atts, $content = null ) {
    $a = shortcode_atts( array(
        'attr_1' => 'attribute 1 default',
        'attr_2' => 'attribute 2 default',
```

```
□□// ...etc
□), $atts );
}
```

This will parse the attributes, set default values, eliminate any unsupported attributes, and store the results in a local array variable named `$a` with the attributes as keys – `$a['attr_1']`, `$a['attr_2']`, and so on. In other words, the array of defaults approximates a list of local variable declarations.

IMPORTANT – Don't use camelCase or UPPER-CASE for your `$atts` attribute names:

`$atts` values are ***lower-cased*** during `shortcode_atts( array( 'attr_1' => 'attr_1 default', // ...etc ), $atts )` processing, so you might want to *just use lower-case*.

NOTE on confusing regex/callback name reference:

The zeroeth entry of the attributes array (`$atts[0]`) will contain the string that matched the shortcode regex, but ONLY if that differs from the callback name, which otherwise appears as the third argument to the callback function.

```
add_shortcode('foo','foo'); // two shortcodes referencing the same callback
 add_shortcode('bar','foo');
    produces this behavior:
 [foo a='b'] ==> callback to: foo(array('a'=>'b'),NULL,"foo");
 [bar a='c'] ==> callback to: foo(array(0 => 'bar', 'a'=>'c'),NULL,"");
```

This is confusing and perhaps reflects an underlying bug, but an overloaded callback routine can correctly determine what shortcode was used to call it, by checking BOTH the third argument to the callback and the zeroeth attribute. (It is NOT an error to have two shortcodes reference the same callback routine, which allows for common code.)

Output

The return value of a shortcode handler function is inserted into the post content output in place of the shortcode macro. **Remember to use return and not echo – anything that is echoed will be output to the browser, but it won't appear in the correct place on the page**.

Shortcodes are parsed after wpautop and wptexturize post formatting has been applied. This means that your shortcode output HTML won't automatically have curly quotes applied, p and br tags added, and so on. If you do want your shortcode output to be formatted, you should call `wpautop()` or `wptexturize()` directly when you return the output from your shortcode handler.

wpautop recognizes shortcode syntax and will attempt not to wrap p or br tags around shortcodes that stand alone on a line by themselves. Shortcodes intended for use in this manner should ensure

that the output is wrapped in an appropriate block tag such as `<p>` or `<div>`.

If the shortcode produces a lot of HTML then `ob_start` can be used to capture output and convert it to a string as follows:

```
function wporg_shortcode() {
	ob_start();
	?> <HTML> <here> ... <?php
	return ob_get_clean();
}
```

Enclosing vs self-closing shortcodes

The examples above show self-closing shortcode macros such as `[myshortcode]`. The API also supports enclosing shortcodes such as `[myshortcode]content[/myshortcode]`.

If a shortcode macro is used to enclose content, its handler function will receive a second parameter containing that content. Users might write shortcodes in either form, so it is necessary to allow for either case by providing a default value for the second parameter to your handler function:

```
function wporg_shortcode_handler( $atts, $content = null )
```

`empty( $content )` can be used to distinguish between the self-closing and enclosing cases.

When content is enclosed, the complete shortcode macro including its content will be replaced with the function output. It is the responsibility of the handler function to provide any necessary escaping or encoding of the raw content string, and include it in the output.

Here’s a trivial example of an enclosing shortcode:

```
function wporg_caption_shortcode( $atts, $content = null ) {
	return '<span class="caption">' . $content . '</span>';
}
add_shortcode( 'caption', 'wporg_caption_shortcode' );
```

When used like this:

```
My Caption
```

The output would be:

```
<span class="caption">My Caption</span>
```

Since `$content` is included in the return value without any escaping or encoding, the user can include raw HTML:

```
<a href="http://example.com/">My Caption</a>
```

Which would produce:

```
<span class="caption"><a href="http://example.com/">My Caption</a></span>
```

This may or may not be intended behaviour – if the shortcode should not permit raw HTML in its output, it should use an escaping or filtering function to deal with it before returning the result.

The shortcode parser uses a single pass on the post content. This means that if the `$content` parameter of a shortcode handler contains another shortcode, it won't be parsed:

```
Caption: [myshortcode]
```

This would produce:

```
<span class="caption">Caption: [myshortcode]</span>
```

If the enclosing shortcode is intended to permit other shortcodes in its output, the handler function can call do_shortcode() recursively:

```
function wporg_caption_shortcode( $atts, $content = null ) {
    return '<span class="caption">' . do_shortcode($content) . '</span>';
}
```

In the previous example, this would ensure the `[myshortcode]` macro in the enclosed content is parsed, and its output enclosed by the caption span:

```
<span class="caption">Caption: The result of myshortcode's handler function</span>
```

The parser does not handle mixing of enclosing and non-enclosing forms of the same shortcode as you would want it to. For example, if you have:

```
[myshortcode example='non-enclosing' /] non-enclosed content [myshortcode] enclosed content
[/myshortcode]
```

Instead of being treated as two shortcodes separated by the text " non-enclosed content ", the parser treats this as a single shortcode enclosing " non-enclosed content `[myshortcode]` enclosed

content”.

Enclosing shortcodes support attributes in the same way as self-closing shortcodes. Here’s an example of the `caption_shortcode()` improved to support a ‘class’ attribute:

```
function wporg_caption_shortcode( $atts, $content = null ) {
	$a = shortcode_atts( array(
		'class' => 'caption',
	), $atts );

	return '<span class="' . esc_attr($a['class']) . '">' . $content . '</span>';
}
```

```
My Caption
```

```
<span class="headline">My Caption</span>
```

Other features in brief

- The parser supports xhtml-style closing shortcodes like `[myshortcode /]`, but this is optional.
- Shortcode macros may use single or double quotes for attribute values, or omit them entirely if the attribute value does not contain spaces. `[myshortcode foo='123' bar=456]` is equivalent to `[myshortcode foo="123" bar="456"]`. Note the attribute value in the last position may not end with a forward slash because the feature in the paragraph above will consume that slash.
- For backwards compatibility with older ad-hoc shortcodes, attribute names may be omitted. If an attribute has no name it will be given a positional numeric key in the `$atts` array. For example, `[myshortcode 123]` will produce `$atts = array( 0 => 123 )`. Positional attributes may be mixed with named ones, and quotes may be used if the values contain spaces or other significant characters.
- The shortcode API has test cases. The tests — which contain a number of examples of error cases and unusual syntax — can be found at http://svn.automattic.com/wordpress-tests/trunk/tests/shortcode.php

Function reference

The following Shortcode API functions are available:

```
function add_shortcode( $tag, $func )
```

Registers a new shortcode handler function. `$tag` is the shortcode string as written by the user (without braces), such as “myshortcode”. $func is the handler function name.

Only one handler function may exist for a given shortcode. Calling `add_shortcode()` again with the same $tag name will overwrite the previous handler.

```
function remove_shortcode( $tag )
```

Deregisters an existing shortcode. `$tag` is the shortcode name as used in `add_shortcode()`.

```
function remove_all_shortcodes()
```

Deregisters all shortcodes.

```
function shortcode_atts( $pairs, $atts )
```

Process a raw array of attributes `$atts` against the set of defaults specified in `$pairs`. Returns an array. The result will contain every key from `$pairs`, merged with values from `$atts`. Any keys in `$atts` that do not exist in $pairs are ignored.

```
function do_shortcode( $content )
```

Parse any known shortcode macros in the `$content` string. Returns a string containing the original content with shortcode macros replaced by their handler functions output.

do_shortcode() is registered as a default filter on ‘the_content’ with a priority of 11.

Limitations

Nested Shortcodes

The shortcode parser correctly deals with nested shortcode macros, provided their handler functions support it by recursively calling do_shortcode():

```
[tag-a]
   [tag-b]
      [tag-c]
   [/tag-b]
[/tag-a]
```

However the parser will fail if a shortcode macro is used to enclose another macro of the same name:

```
[tag-a]
   [tag-a]
   [/tag-a]
[/tag-a]
```

This is a limitation of the context-free regexp parser used by `do_shortcode()` – it is very fast but does not count levels of nesting, so it can't match each opening tag with its correct closing tag in these cases.

In future versions of WordPress, it may be necessary for plugins having a nested shortcode syntax to ensure that the `wptexturize()` processor does not interfere with the inner codes. It is recommended that for such complex syntax, the no_texturize_shortcodes filter should be used on the outer tags. In the examples used here, tag-a should be added to the list of shortcodes to not texturize. If the contents of tag-a or tag-b still need to be texturized, then you can call `wptexturize()` before calling `do_shortcode()` as described above.

Unregistered Names

Some plugin authors have chosen a strategy of not registering shortcode names, for example to disable a nested shortcode until the parent shortcode's handler function is called. This may have unintended consequences, such as failure to parse shortcode attribute values. For example:

```
[tag-a unit="north"]
   [tag-b size="24"]
      [tag-c color="red"]
   [/tag-b]
[/tag-a]
```

Starting with version 4.0.1, if a plugin fails to register tag-b and tag-c as valid shortcodes, the `wptexturize()` processor will output the following text prior to any shortcode being parsed:

```
[tag-a unit="north"]
   [tag-b size=”24”]
      [tag-c color=”red”]
   [/tag-b]
[/tag-a]
```

Unregistered shortcodes should be considered normal plain text that have no special meaning, and the practice of using unregistered shortcodes is discouraged. If you must enclose raw code between shortcode tags, at least consider using the no_texturize_shortcodes filter to prevent texturization of the contents of tag-a:

```
add_shortcode( 'tag-a', 'wporg_tag_a_handler' );
add_filter( 'no_texturize_shortcodes', 'wporg_ignore_tag_a' );

function wporg_ignore_tag_a( $list ) {
  $list[] = 'tag-a';
  return $list;
}
```

Unclosed Shortcodes

In certain cases the shortcode parser cannot correctly deal with the use of both closed and unclosed shortcodes. For instance in this case the parser will only correctly identify the second instance of the shortcode:

```
[tag]
[tag]
   CONTENT
[/tag]
```

However in this case the parser will identify both:

```
[tag]
   CONTENT
[/tag]
[tag]
```

Hyphens

Take caution when using hyphens in the name of your shortcodes. In the following instance WordPress may see the second opening shortcode as equivalent to the first (basically WordPress sees the first part before the hyphen):

```
[tag]
[tag-a]
```

It all depends on which shortcode is defined first. If you are going to use hyphens then define the shortest shortcode first.

To avoid this, use an underscore or simply no separator:

```
[tag]
[tag_a]
```

```
[tag]
[taga]
```

If the first part of the shortcode is different from one another, you can get away with using hyphens:

```
[tag]
[tagfoo-a]
```

Important: Using hyphens can have implications that you may not be aware of; such as if other installed shortcodes also are use hyphens, the use of generic words with hyphens may cause collisions (if shortcodes are used together within the same request):

```
// plugin-A
[is-logged-in]

// plugin-B
[is-admin]
```

Square Brackets

The shortcode parser does not accept square brackets within attributes. Thus the following will fail:

```
[tag attribute="[Some value]"]
```

Tags surrounded by cosmetic brackets are not yet fully supported by wptexturize() or its filters. These codes may give unexpected results:

```
[I put random text near my captions. ]
```

Note: these limitations may change in future versions of WordPress, you should test to be absolutely sure.

HTML

Starting with version 3.9.3, use of HTML is limited inside shortcode attributes. For example, this shortcode will not work correctly because it contains a `>` character:

```
[tag value1="35" value2="25" compare=">"]
```

Version 4.0 is designed to allow validated HTML, so this will work:

```
[tag description="<b>Greetings</b>"]
```

The suggested workaround for HTML limitations is to use HTML encoding for all user input, and then add HTML decoding in the custom shortcode handler. Extra API functions are planned.

Full usage of HTML in shortcode attributes was never officially supported, and this will not be expanded in future versions.

Starting with version 4.2.3, similar limitations were placed on use of shortcodes inside HTML. For example, this shortcode will not work correctly because it is nested inside a scripting attribute:

```
<a onclick="[tag]">
```

The suggested workaround for dynamic attributes is to design a shortcode that outputs all needed HTML rather than just a single value. This will work better:

```
[link onclick="tag"]
```

Also notice the following shortcode is no longer allowed because of incorrect attribute quoting:

```
<a title="[tag attr="id"]">
```

The only way to parse this as valid HTML is to use single quotes and double quotes in a nested manner:

```
<a title="[tag attr='id']">
```

Registration Count

The API is known to become unstable when registering hundreds of shortcodes. Plugin authors should create solutions that rely on only a small number of shortcodes names. This limitation might change in future versions.

Formal Syntax

WordPress shortcodes do not use special characters in the same way as HTML. The square braces may seem magical at first glance, but they are not truly part of any language. For example:

```
[gallery]
```

The gallery shortcode is parsed by the API as a special symbol because it is a registered shortcode. On the other hand, square braces are simply ignored when a shortcode is not registered:

```
[randomthing]
```

The randomthing symbol and its square braces are ignored because they are not part of any registered shortcode.

In a perfect world, any `[*]` symbol could be handled by the API, but we have to consider the following challenges: Square braces are allowed in HTML and are not always shortcodes, angle braces are allowed inside of shortcodes only in limited situations, and all of this code must run through multiple layers of customizeable filters and parsers before output. Because of these language compatibility issues, square braces can't be magical.

The shortcode syntax uses these general parts:

```
[name attributes close]
```

```
[name attributes]Any HTML or shortcode may go here.[/name]
```

Escaped shortcodes are identical but have exactly two extra braces:

```
[[name attributes close]]
```

```
[[name attributes]Any HTML or shortcode may go here.[/name]]
```

Again, the shortcode name must be registered, otherwise all four examples would be ignored.

Names

Shortcode names must never contain the following characters:

- Square braces: `[ ]`
- Angle braces: `< >`
- Ampersand: `&`
- Forward slash: `/`
- Whitespace: space linefeed tab
- Non-printing characters: `x00` – `x20`

It is recommended to also avoid quotes in the names of shortcodes.

Attributes

Attributes are optional. A space is required between the shortcode name and the shortcode attributes. When more than one attribute is used, each attribute must be separated by at least one space.

Each attribute should conform to one of these formats:

```
attribute_name = 'value'
```

```
attribute_name = "value"
```

```
attribute_name = value
```

```
"value"
```

```
value
```

Attribute names are optional and should contain only the following characters for compatibility across all platforms:

- Upper-case and lower-case letters: `A-Z` `a-z`
- Digits: `0-9`
- Underscore: `_`
- Hyphen: `-`

Spaces are not allowed in attribute names. Optional spaces may be used between the name and the `=` sign. Optional spaces may also be used between the `=` sign and the value.

It should be noted that even though attributes can be used with mixed case in the editor, they will always be lowercase after parsing.

Attribute values must never contain the following characters:

- Square braces: `[ ]`
- Quotes: `"`, `'`

Unquoted values also must never contain spaces.

HTML characters `<` and `>` have only limited support in attributes.

The recommended method of escaping special characters in shortcode attributes is HTML encoding. Most importantly, any user input appearing in a shortcode attribute must be escaped or stripped of special characters.

Note that double quotes are allowed inside of single-quoted values and vice versa, however this is not recommended when dealing with user input.

The following characters, if they are not escaped within an attribute value, will be automatically stripped and converted to spaces:

- No-break space: `xC2xA0`
- Zero-width space: `xE2x80x8B`

Self-Closing

The self-closing marker, a single forward slash, is optional. Space before the marker is optional. Spaces are not allowed after the marker.

```
[example /]
```

The self-closing marker is purely cosmetic and has no effect except that it will force the shortcode parser to ignore any closing tag that follows it.

The enclosing type shortcodes may not use a self-closing marker.

Escaping

WordPress attempts to insert curly quotes between the `[name]` and `[/name]` tags. It will process that content just like any other. As of 4.0.1, unregistered shortcodes are also “texturized” and this may give unexpected curly quotes:

```
[randomthing param="test"]
```

A better example would be:

```
<code>[randomthing param="test"]</code>
```

The `<code>` element is always avoided for the sake of curly quotes.

Registered shortcodes are still processed inside of `<code>` elements. To escape a registered shortcode for display on your website, the syntax becomes:

```
[[caption param="test"]]
```

which will output:

```
[caption param="test"]
```

The `<code>` element is optional in that situation.

For enclosing shortcodes, use the following syntax:

```
[[caption]My Caption]
```

External Resources

- WordPress Shortcodes Generator
- Add Shortcode – WordPress Code Snippet Generator – A snippet generator and full documentation about how to add the code to a WordPress website.
- Shortcode summary by Aaron D. Campbell – Explains shortcodes and gives examples including how to incorporate shortcodes into a meta box for sending them to the editor using js.
- Innovative WordPress Shortcodes In Action – a WordPress plugin that shows you how to effectively use shortcodes to change your post content designs.
- WordPress Shortcode API Overview – explanations on usage and example of plugin using shortcodes.
- Simple shortcode-powered BBCode plugin – a simple plugin that adds support for BBCode via shortcode. A good way to see shortcodes in action.

Default Shortcodes

- `[ audio ]`
- `[ wp_caption ]`
- `[ caption ]`
- `[ embed ]`
- `[ gallery ]`
- `[ video ]`
- `[ playlist ]`

Shortcode API functions list

- Function: do_shortcode()
- Function: add_shortcode()
- Function: remove_shortcode()
- Function: remove_all_shortcodes()
- Function: shortcode_atts()
- Function: strip_shortcodes()
- Function: shortcode_exists()
- Function: has_shortcode()
- Function: get_shortcode_regex()

- Function: wp_audio_shortcode()
- Function: wp_video_shortcode()
- Filter: no_texturize_shortcodes

□□□ #1
□ □□□ □□□ 2026-08-29 23:36:33 UTC
□ □□□ □□□ 2026-08-29 23:36:33 UTC