

Commands Cookbook



- [Overview](#)
 - [Command types](#)
- [Anatomy of a command](#)
 - [Required registration arguments](#)
 - [Quick and dirty execution](#)
 - [Optional registration arguments](#)
 - [Annotating with PHPDoc](#)
 - [Shortdesc](#)
 - [Longdesc](#)
 - [Argument aliases](#)
 - [YAML parameter options](#)
 - [Docblock tags](#)
 - [WP_CLI::add_command\(\)'s third \\$args parameter](#)
 - [Command internals](#)
 - [Accepting arguments](#)
 - [Effectively reusing WP-CLI internal APIs](#)
 - [Help rendering](#)
- [Writing tests](#)
- [Adding commands globally](#)
- [Distribution](#)
 - [Include in a plugin or theme](#)
 - [Distribute as a stand-alone command](#)
 - [Git repository](#)
 - [ZIP file](#)

Creating your own custom WP-CLI command can be easier than it looks — and you can use `wp scaffold package` ([repo](#)) to dynamically generate everything but the command itself.

Overview

WP-CLI's goal is to offer a complete alternative to the WordPress admin; for any action you might want to perform in the WordPress admin, there should be an equivalent WP-CLI command. A

command is an atomic unit of WP-CLI functionality. `wp plugin install` ([doc](#)) is one such command, as is `wp plugin activate` ([doc](#)). Commands are useful to WordPress users because they can offer simple, precise interfaces for performing complex tasks.

But, the WordPress admin is a Swiss Army knife of infinite complexity. There's no way just this project could handle every use case. This is why WP-CLI includes a set of [common internal commands](#), while also offering a [rich internal API](#) for third-parties to write and register their own commands.

WP-CLI commands can be distributed as standalone packages, or bundled with WordPress plugins or themes. Standalone packages can be published as Composer packages and discovered through [Packagist](#). You can use `wp scaffold package` ([repo](#)) to dynamically generate everything but the command itself.

Packages are to WP-CLI as plugins are to WordPress. There are distinct differences in the approach you should take to creating a WP-CLI package. While WP-CLI is an ever-growing alternative to /wp-admin it is important to note that you must first write your package to work with the WP-CLI internal API before considering how you work with WordPress APIs.

Command types

Bundled commands:

- Usually cover functionality offered by a standard WordPress installation. There are exceptions to this rule though, notably `wp search-replace` ([doc](#)).
- Do not depend on other components such as plugins, themes etc.
- Are maintained by the WP-CLI team.

Third-party commands:

- Can be defined in plugins or themes.
- Can be easily scaffolded as standalone projects with `wp scaffold package` ([repo](#)).
- Can be distributed independent of a plugin or theme as Composer packages, Git repositories, local paths, or ZIP files.

All commands:

- Follow the [documentation standards](#).

Anatomy of a command

WP-CLI supports registering any callable class, function, or closure as a command.

`WP_CLI::add_command()` ([doc](#)) is used for both internal and third-party command registration.

The **synopsis** of a command defines which **positional** and **associative** arguments a command accepts. Let's take a look at the synopsis for `wp plugin install`:

```
$ wp plugin install
usage: wp plugin install <plugin|zip|url>... [--version=<version>] [--force] [--activate] [--activate-network]
```

In this example, `<plugin|zip|url>...` is the accepted **positional** argument. In fact, `wp plugin install` accepts the same positional argument (the slug, ZIP, or URL of a plugin to install) multiple times. `[--version=<version>]` is one of the accepted **associative** arguments. It's used to denote the version of the plugin to install. Notice, too, the square brackets around the argument definition; square brackets mean the argument is optional.

WP-CLI also has a [series of global arguments](#) which work with all commands. For instance, including `--debug` means your command execution will display all PHP errors, and add extra verbosity to the WP-CLI bootstrap process.

Required registration arguments

When registering a command, `WP_CLI::add_command()` requires two arguments:

1. `$name` is the command's name within WP-CLI's namespace (e.g. `plugin install` or `post list`).
2. `$callable` is the implementation of the command, as a callable class, function, or closure.

In the following example, each instance of `wp foo` is functionally equivalent:

```
// 1. Command is a function
function foo_command( $args ) {
    WP_CLI::success( $args[0] );
}
WP_CLI::add_command( 'foo', 'foo_command' );

// 2. Command is a closure
$foo_command = function( $args ) {
    WP_CLI::success( $args[0] );
};
WP_CLI::add_command( 'foo', $foo_command );
```

```

};

WP_CLI::add_command( 'foo', $foo_command );

// 3. Command is a method on a class
class Foo_Command {
    public function __invoke( $args ) {
        WP_CLI::success( $args[0] );
    }
}

WP_CLI::add_command( 'foo', 'Foo_Command' );

// 4. Command is a method on a class with constructor arguments
class Foo_Command {
    protected $bar;
    public function __construct( $bar ) {
        $this->bar = $bar;
    }
    public function __invoke( $args ) {
        WP_CLI::success( $this->bar . ':' . $args[0] );
    }
}

$instance = new Foo_Command( 'Some text' );
WP_CLI::add_command( 'foo', $instance );

```

Importantly, classes behave a bit differently than functions and closures in that:

- Any public methods on a class are registered as subcommands of the command. For instance, given the examples above, a method `bar()` on the class `Foo` would be registered as `wp foo bar`. But...
- `__invoke()` is treated as a magic method. If a class implements `__invoke()`, the command name will be registered to that method and no other methods of that class will be registered as commands.

Note: Historically, WP-CLI provided a base `WP_CLI_Command` class to extend, however extending this class is not required and will not change how your command behaves.

All commands can be registered to their own top-level namespace (e.g. `wp foo`), or as subcommands to an existing namespace (e.g. `wp core foo`). For the latter, simply include the existing namespace as a part of the command definition.

```

class Foo_Command {
    public function __invoke( $args ) {

```

```
        WP_CLI::success( $args[0] );
    }
}
WP_CLI::add_command( 'core foo', 'Foo_Command' );
```

Quick and dirty execution

Writing a short script for a one-off task, and don't need to register it formally with

`WP_CLI::add_command()`? `wp eval-file` is your ticket ([doc](#)).

Given a `simple-command.php` file:

```
<?php
WP_CLI::success( "The script has run!" );
```

Your "command" can be run with `wp eval-file simple-command.php`. If the command doesn't have a dependency on WordPress, or WordPress isn't available, you can use the `--skip-wordpress` flag to avoid loading WordPress.

Optional registration arguments

WP-CLI supports two ways of registering optional arguments for your command: through the callable's PHPDoc, or passed as a third `$args` parameter to `WP_CLI::add_command()`.

Annotating with PHPDoc

A typical WP-CLI class looks like this:

```
<?php
/**
 * Implements example command.
 */
class Example_Command {

    /**
     * Prints a greeting.
     *
     * ## OPTIONS
     *
     * <name>
```

```

 * : The name of the person to greet.
 *
 * [--type=<type>]
 * : Whether or not to greet the person with success or error.
 * ---
 * default: success
 * options:
 *   - success
 *   - error
 * ---
 *
 * ## EXAMPLES
 *
 *     wp example hello Newman
 *
 * @when after_wp_load
 */
function hello( $args, $assoc_args ) {
    \list( $name ) = $args;

    // Print the message with type
    $type = $assoc_args['type'];
    WP_CLI::$type( "Hello, $name!" );
}

WP_CLI::add_command( 'example', 'Example_Command' );

```

This command's PHPDoc is interpreted in three ways:

Shortdesc

The shortdesc is the first line in the PHPDoc:

```

/**
 * Prints a greeting.

```

Longdesc

The longdesc is middle part of the PHPDoc:

```

□ * ## OPTIONS
□ *
□ * <name>
□ * : The name of the person to greet.
□ *
□ * [--type=<type>]
□ * : Whether or not to greet the person with success or error.
□ * ---
□ * default: success
□ * options:
□ *   - success
□ *   - error
□ * ---
□ *
□ * ## EXAMPLES
□ *
□ *     wp example hello Newman

```

Options defined in the longdesc are interpreted as the command's **synopsis**:

- `<name>` is a required positional argument. Changing it to `<name>...` would mean the command can accept one or more positional arguments. Changing it to `[<name>]` would mean that the positional argument is optional and finally, changing it to `[<name>...]` would mean that the command can accept multiple optional positional arguments.
- `[--type=<type>]` is an optional associative argument which defaults to 'success' and accepts either 'success' or 'error'. Changing it to `[--error]` would change the argument to behave as an optional boolean flag.
- `[--field[=<value>]]` allows an optional argument to be used with or without a value. An example of this would be using a global parameter like `--skip-plugins[=<plugins>]` which can either skip loading all plugins, or skip a comma-separated list of plugins.
- `[--status=<status>...]` (note the trailing `...`) declares a **repeating** associative argument.
 - When the same flag is passed more than once (e.g. `--status=active --status=inactive`), WP-CLI collects the values into an array and `$assoc_args['status']` will be `[ 'active', 'inactive' ]`.
 - Without the `...` ellipsis, passing the same flag multiple times uses **last-wins** behaviour—only the final value is kept.
 - Boolean flags (e.g. `[--verbose]`) always use last-wins and are never collected into arrays, regardless of ellipsis.

Note: To accept arbitrary/unlimited number of optional associative arguments you would use the annotation `[--<field>=<value>]`. So for example:

```
* [--<field>=<value>]
* : Allow unlimited number of associative parameters.
```

A command's synopsis is used for validating the arguments, before passing them to the implementation.

Argument aliases

You can define one or more **aliases** for a parameter by appending them after a `|` separator inside the synopsis token. This lets users pass a short form (or any alternative name) and have WP-CLI automatically map it to the canonical argument name in `$assoc_args`.

```
* [--with-dependencies|w]
* : Include dependencies in the operation.
```

With this definition, both `--with-dependencies` and `-w` are accepted, and `$assoc_args['with-dependencies']` is populated in either case.

Multiple aliases are separated by additional `|` characters:

```
* [--verbose|v|wordy]
* : Enable verbose output.
```

Aliases work for value-carrying associative arguments too:

```
* [--format=<format>|f]
* : Output format.
```

Running `wp example hello -f=json` is then equivalent to `wp example hello --format=json`. Note that WP-CLI accepts both `-<alias>=value` (single-dash) and `--<name>=value` (double-dash) forms for short-form aliases.

YAML parameter options

The `---` block after a parameter description allows you to set extra metadata for that parameter using YAML.

`default` sets the value used when the argument is omitted:

```
* [--type=<type>]
* : Whether or not to greet the person with success or error.
* ---
* default: success
```

```
* options:
*   - success
*   - error
*   ---
```

options restricts the accepted values. WP-CLI validates user input against this list and returns an error for any value not in it.

sensitive marks an argument as containing sensitive data (e.g. passwords, API keys):

- When the `--prompt` global flag is used, WP-CLI logs the full command it is about to run to STDOUT.
- Any argument flagged with `sensitive: true` will have its value replaced with `[REDACTED]` in that log line.
- This prevents secrets from leaking into logs or terminal history.

```
* [--password=<password>]
* : Database password.
* ---
* sensitive: true
* ---
```

The longdesc is also displayed when calling the `help` command, for example, `wp help example hello`. Its syntax is [Markdown Extra](#) and here are a few more notes on how it's handled by WP-CLI:

- The longdesc is generally treated as a free-form text. The **OPTIONS** and **EXAMPLES** section names are not enforced, just common and recommended.
- Section names (`## NAME`) are colorized and printed with zero indentation.
- Everything else is indented by 2 characters, option descriptions are further indented by additional 2 characters.
- Word-wrapping is a bit tricky. If you want to utilize as much space on each line as possible and don't get word-wrapping artifacts like one or two words on the next line, follow these rules:
 - Hard-wrap option descriptions at **75 chars** after the colon and a space.
 - Hard-wrap everything else at **90 chars**.

For more details on how you should format your command docs, please see WP-CLI's [documentation standards](#).

Docblock tags

This is the last section and it starts immediately after the longdesc:

```
□ * @when after_wp_load
□ */
```

Here's the list of defined tags:

@subcommand

There are cases where you can't make the method name have the name of the subcommand. For example, you can't have a method named `list`, because `list` is a reserved keyword in PHP.

That's when the `@subcommand` tag comes to the rescue:

```
□/**
□ * @subcommand list
□ */
□function _list( $args, $assoc_args ) {
□□...
□}

□/**
□ * @subcommand do-chores
□ */
□function do_chores( $args, $assoc_args ) {
□□...
□}
```

@alias

With the `@alias` tag, you can add another way of calling a subcommand. Example:

```
□/**
□ * @alias hi
□ */
□function hello( $args, $assoc_args ) {
□□...
□}
```

```
$ wp example hi Joe
Success: Hello, Joe!
```

@when

This is a special tag that tells WP-CLI when to execute the command. It supports [all registered WP-CLI hooks](#).

Most WP-CLI commands execute after WordPress has loaded. The default behavior for a command is:

```
@when after_wp_load
```

To have a WP-CLI command run before WordPress loads, use:

```
@when before_wp_load
```

Do keep in mind most WP-CLI hooks fire before WordPress is loaded. If your command is loaded from a plugin or theme, `@when` will be essentially ignored.

It has no effect if the command using it is loaded from a plugin or a theme, because by that time WordPress itself will have already been loaded.

@skipglobalargcheck

WP-CLI emits a warning at command-registration time if a command defines an argument whose name collides with an existing [global argument](#) (e.g. `--debug`, `--user`, `--quiet`). This helps command authors avoid confusing behaviour where the global argument takes precedence over the command-specific one.

If you intentionally want to reuse a global argument name (for example, when wrapping another tool that uses the same flag), you can silence the warning with `@skipglobalargcheck`:

```
#!/**
 * @skipglobalargcheck
 * @when before_wp_load
 */
function my_command( $args, $assoc_args ) {
    ...
}
```

WP_CLI::add_command()'s third \$args parameter

Each of the configuration options supported by PHPDoc can instead be passed as the third argument in command registration:

```
$hello_command = function( $args, $assoc_args ) {
    list( $name ) = $args;
```

```

$type = $assoc_args['type'];
WP_CLI::$type( "Hello, $name!" );
if ( isset( $assoc_args['honk'] ) ) {
    WP_CLI::log( 'Honk!' );
}
};

WP_CLI::add_command( 'example hello', $hello_command, array(
    'shortdesc' => 'Prints a greeting.',
    'synopsis' => array(
        array(
            'type'          => 'positional',
            'name'          => 'name',
            'description'   => 'The name of the person to greet.',
            'optional'     => false,
            'repeating'     => false,
        ),
        array(
            'type'          => 'assoc',
            'name'          => 'type',
            'description'   => 'Whether or not to greet the person with success or error.',
            'optional'     => true,
            'default'       => 'success',
            'options'       => array( 'success', 'error' ),
        ),
        array(
            'type'          => 'flag',
            'name'          => 'honk',
            'optional'     => true,
        ),
    ),
    'when' => 'after_wp_load',
    'longdesc' => '## EXAMPLES' . "\n\n" . 'wp example hello Newman',
) );

```

Note that the `longdesc` attribute will be appended to the description of the options generated from the synopsis, so this argument is great for adding examples of usage. If there is no synopsis, the `longdesc` attribute will be used as is to provide a description.

Command internals

Now that you know how to register a command, the world is your oyster. Inside your callback, your command can do whatever it wants.

Accepting arguments

In order to handle runtime arguments, you have to add two parameters to your callable: `$args` and `$assoc_args`.

```
function hello( $args, $assoc_args ) {  
    /* Code goes here*/  
}
```

The `$args` variable will store all the positional arguments:

```
$ wp example hello Joe Doe
```

```
WP_CLI::line( $args[0] ); // Joe  
WP_CLI::line( $args[1] ); // Doe
```

The `$assoc_args` variable will store all the arguments defined like `--key=value` or `--flag` or `--no-flag`:

```
$ wp example hello --name='Joe Doe' --verbose --no-option
```

```
WP_CLI::line( $assoc_args['name'] ); // Joe Doe  
WP_CLI::line( $assoc_args['verbose'] ); // true  
WP_CLI::line( $assoc_args['option'] ); // false
```

Also, you can combine argument types:

```
$ wp example hello --name=Joe foo --verbose bar
```

```
WP_CLI::line( $assoc_args['name'] ); // Joe  
WP_CLI::line( $assoc_args['verbose'] ); // true  
WP_CLI::line( $args[0] ); // foo  
WP_CLI::line( $args[1] ); // bar
```

Effectively reusing WP-CLI internal APIs

As an example, say you were tasked with finding all unused themes on a multisite network ([#2523](#)). If you had to perform this task manually through the WordPress admin, it would probably take hours, if not days, of effort. However, if you're familiar with writing WP-CLI commands, you could

complete the task in 15 minutes or less.

Here's what such a command looks like:

```
/**
 * Find unused themes on a multisite network.
 *
 * Iterates through all sites on a network to find themes which aren't enabled
 * on any site.
 */
$find_unused_themes_command = function() {
    []$response = WP_CLI::launch_self( 'site list', array(), array( 'format' => 'json' ), false,
    true );
    []$sites = json_decode( $response->stdout );
    []$unused = array();
    []$used = array();
    []foreach( $sites as $site ) {
        []WP_CLI::log( "Checking {$site->url} for unused themes..." );
        []$response = WP_CLI::launch_self( 'theme list', array(), array( 'url' => $site->url, 'format'
        => 'json' ), false, true );
        []$themes = json_decode( $response->stdout );
        []foreach( $themes as $theme ) {
            []if ( 'no' == $theme->enabled && 'inactive' == $theme->status && ! in_array( $theme->name,
            $used ) ) {
                [][]$unused[ $theme->name ] = $theme;
            } else {
                [][]if ( isset( $unused[ $theme->name ] ) ) {
                    [][]unset( $unused[ $theme->name ] );
                }
                [][]$used[] = $theme->name;
            }
        }
    }
    []WP_CLI\Utils\format_items( 'table', $unused, array( 'name', 'version' ) );
};
WP_CLI::add_command( 'find-unused-themes', $find_unused_themes_command, array(
    []'before_invoke' => function(){
        []if ( ! is_multisite() ) {
            []WP_CLI::error( 'This is not a multisite installation.' );
        }
    }
}
```

```
} ,  
 ) );
```

Let's run through the [internal APIs](#) this command uses to achieve its goal:

- `WP_CLI::add_command()` ([doc](#)) is used to register a `find-unused-themes` command to the `$find_unused_themes_command` closure. The `before_invoke` argument makes it possible to verify the command is running on a multisite install, and error if not.
- `WP_CLI::error()` ([doc](#)) renders a nicely formatted error message and exits.
- `WP_CLI::launch_self()` ([doc](#)) initially spawns a process to get a list of all sites, then is later used to get the list of themes for a given site.
- `WP_CLI::log()` ([doc](#)) renders informational output to the end user.
- `WP_CLI\Utils\format_items()` ([doc](#)) renders the list of unused themes after the command has completed its discovery.

Help rendering

Your command's PHPDoc (or registered definition) is rendered using the `help` command. The output is ordered like this:

1. Short description
2. Synopsis
3. Long description (OPTIONS, EXAMPLES etc.)
4. Global parameters

Writing tests

WP-CLI makes use of a Behat-based testing framework, which you should use too. Behat is a great choice for your WP-CLI commands because:

- It's easy to write new tests, which means they'll actually get written.
- The tests interface with your command in the same manner as your users interface with your command.

Behat tests live in the `features/` directory of your project. Here's an example from `features/cli-info.feature`:

```
Feature: Review CLI information
```

```
Scenario: Get the path to the packages directory
```

Given an empty directory

When I run ``wp cli info --format=json``

Then STDOUT should be JSON containing:

```
""  
  
{"wp_cli_packages_dir_path":"/tmp/wp-cli-home/.wp-cli/packages/"}
```

When I run ``WP_CLI_PACKAGES_DIR=/tmp/packages wp cli info --format=json``

Then STDOUT should be JSON containing:

```
""  
  
{"wp_cli_packages_dir_path":"/tmp/packages/"}
```

Functional tests typically follow this pattern:

- **Given** some background,
- **When** a user performs a specific action,
- **Then** the end result should be X (and Y and Z).

Convinced? Head on over to [wp-cli/scaffold-package-command](#) to get started.

Adding commands globally

If you want to have custom commands available globally, without needing to create a full plugin or package, you can use WP-CLI's `require` configuration option.

Edit your [global config file](#) (usually found at `~/.wp-cli/config.yml`) to require a PHP file:

```
require:  
- ~/.wp-cli/commands.php
```

Then create the file and add your custom commands:

```
<?php  
  
WP_CLI::add_command( 'hello-world', function () {  
    WP_CLI::success( "Hello World!" );  
} );
```

Any commands registered in that file will be globally available whenever you run WP-CLI. This is great for personal helper commands that you want available across all your WordPress projects.

You can also pass the `--require=<path>` flag on the command line, or set the `WP_CLI_REQUIRE` environment variable, to load a custom PHP file for a single invocation.

Distribution

Now that you've produced a command you're proud of, it's time to share it with the world. There are two common ways of doing so.

Include in a plugin or theme

One way to share WP-CLI commands is by packaging them in your plugin or theme. Many people do so by conditionally loading (and registering) the command based on the presence of the `WP_CLI` constant.

```
if ( defined( 'WP_CLI' ) && WP_CLI ) {  
    require_once dirname( __FILE__ ) . '/inc/class-plugin-cli-command.php';  
}
```

Alternatively, you can use the `cli_init` hook to register your commands. This hook is fired during the WP-CLI Runner startup process, providing a dedicated event for command registration without needing to check for the `WP_CLI` constant.

```
/**  
 * Register custom WP-CLI commands using the cli_init hook.  
 */  
function myplugin_register_cli_commands() {  
    require_once dirname( __FILE__ ) . '/inc/class-plugin-cli-command.php';  
    WP_CLI::add_command( 'myplugin', 'MyPlugin_CLI_Command' );  
}  
add_action( 'cli_init', 'myplugin_register_cli_commands' );
```

Both approaches are valid. Use the `WP_CLI` constant check when you need to conditionally load code based on whether WP-CLI is present. Use the `cli_init` hook when you want to hook into a specific point in the WP-CLI initialization process.

Distribute as a stand-alone command

Standalone WP-CLI commands can be installed from any git repository, ZIP file or folder. The only technical requirement is to include a valid `composer.json` file with an autoload declaration. We recommend including `"type": "wp-cli-package"` to distinguish your project explicitly as a WP-CLI package.

Here's a full `composer.json` example from the server command:

```
{
  "name": "wp-cli/server-command",
  "description": "Start a development server for WordPress",
  "type": "wp-cli-package",
  "homepage": "https://github.com/wp-cli/server-command",
  "license": "MIT",
  "authors": [
    {
      "name": "Package Maintainer",
      "email": "packagemaintainer@homepage.com",
      "homepage": "https://www.homepage.com"
    }
  ],
  "require": {
    "php": ">=5.3.29"
  },
  "autoload": {
    "files": [ "command.php" ]
  }
}
```

Note the `autoload` declaration, which loads `command.php`.

Once you've added a valid `composer.json` file to your project repository, WP-CLI users can pull it in via the package manager from the location you opted to store it in. Here are a few examples of storage locations and the corresponding syntax of installing it via the package manager:

Git repository

To install a package that is found in a git repository, you can provide either the HTTPS or the SSH link to the git repository to the `package install` command.

```
# Installing the package using an HTTPS link
$ wp package install https://github.com/wp-cli/server-command.git
```

```
# Installing the package using an SSH link
$ wp package install git@github.com:wp-cli/server-command.git
```

ZIP file

You can install a package from a ZIP file by providing the path to that file to the `wp package install` command.

```
# Installing the package using a ZIP file
$ wp package install ~/Downloads/server-command-main.zip
```

📦 #4

📦 cocobook-sync 📦	2026-10-28 10:58:36 UTC
📦 cocobook-sync 📦	2026-10-29 01:42:30 UTC