# How to WP-CLI

## Introduction

Using WP-CLI, you can perform specific tasks using the command line instead of accessing the WordPress admin panel. In the following section, we will cover step by step some of these tasks.

- How-to install WordPress (https://make.wordpress.org/cli/handbook/how-to-install/)
- How to put the website in maintenance mode (https://make.wordpress.org/cli/handbook/how-to-maintenance-mode)
- How-to start the webserver (https://make.wordpress.org/cli/handbook/)
- How to create a custom plugin (https://make.wordpress.org/cli/handbook/how-to-create-custom-plugins)

---

# Internal API

WP-CLI includes a number of utilities which are considered stable and meant to be used by commands.

This also means functions and methods not listed here are considered part of the private API. They may change or disappear at any time.

*Internal API documentation is generated from the WP-CLI codebase on every release. To suggest improvements, please submit a pull request.*

---

## Registration

- **WP_CLI::add_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-hook/)** - Schedule a callback to be executed at a certain point.
- **WP_CLI::do_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-do-hook/)** - Execute callbacks registered to a given hook.
- **WP_CLI::add_wp_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-wp-hook/)** - Add a callback to a WordPress action or filter.
- **WP_CLI::add_command() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-command/)** - Register a command to WP-CLI.

## Output

- **WP_CLI\Utils\format_items() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-format-items/)** - Render a collection of items as an ASCII table, JSON, CSV, YAML, list of ids, or count.
- **WP_CLI\Utils\make_progress_bar() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-make-progress-bar/)** - Create a progress bar to display percent completion of a given operation.
- **WP_CLI::colorize() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-colorize/)** - Colorize a string for output.
- **WP_CLI::line() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-line/)** - Display informational message without prefix, and ignore `--quiet`.
- **WP_CLI::log() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-log/)** - Display informational message without prefix.
- **WP_CLI::success() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-success/)** - Display success message prefixed with "Success: ".
- **WP_CLI::debug() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-debug/)** - Display debug message prefixed with "Debug: " when `--debug` is used.
- **WP_CLI::warning() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-warning/)** - Display warning message prefixed with "Warning: ".
- **WP_CLI::error() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-error/)** - Display error message prefixed with "Error: " and exit script.
- **WP_CLI::halt() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-halt/)** - Halt script execution with a specific return code.
- **WP_CLI::error_multi_line() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-error-multi-line/)** - Display a multi-line error message in a red box. Doesn't exit script.

## Input

- **WP_CLI\Utils\launch_editor_for_input() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-launch-editor-for-input/)** - Launch system's $EDITOR for the user to edit some text.
- **WP_CLI\Utils\get_flag_value() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-flag-value/)** - Return the flag value or, if it's not set, the $default value.
- **WP_CLI\Utils\report_batch_operation_results() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-report-batch-operation-results/)** - Report the results of the same operation against multiple resources.
- **WP_CLI\Utils\parse_str_to_argv() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-parse-str-to-argv/)** - Parse a string of command line arguments into an $argv-esqe variable.
- **WP_CLI::confirm() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-confirm/)** - Ask for confirmation before running a destructive operation.
- **WP_CLI::read_value() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-read-value/)** - Read a value, from various formats.
- **WP_CLI::has_config() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-has-config/)** - Confirm that a global configuration parameter does exist.
- **WP_CLI::get_config() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-get-config/)** - Get values of global configuration parameters.

# Execution

- **WP_CLI::launch() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-launch/)** - Launch an arbitrary external process that takes over I/O.
- **WP_CLI::launch_self() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-launch-self/)** - Run a WP-CLI command in a new process reusing the current runtime arguments.
- **WP_CLI::runcommand() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-runcommand/)** - Run a WP-CLI command.
- **WP_CLI::run_command() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-run-command/)** - Run a given command within the current process using the same global parameters.

# System

- **WP_CLI\Utils\get_home_dir() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-home-dir/)** - Get the home directory.
- **WP_CLI\Utils\trailingslashit() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-trailingslashit/)** - Appends a trailing slash.
- **WP_CLI\Utils\normalize_path() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-normalize-path/)** - Normalize a filesystem path.
- **WP_CLI\Utils\get_temp_dir() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-temp-dir/)** - Get the system's temp directory. Warns user if it isn't writable.
- **WP_CLI\Utils\get_php_binary() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-php-binary/)** - Get the path to the PHP binary used when executing WP-CLI.
- **WP_CLI::get_php_binary() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-get-php-binary/)** - Get the path to the PHP binary used when executing WP-CLI.

# Misc

- **WP_CLI\Utils\write_csv() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-write-csv/)** - Write data as CSV to a given file.
- **WP_CLI\Utils\http_request() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-http-request/)** - Make a HTTP request to a remote URL.
- **WP_CLI\Utils\get_named_sem_ver() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-named-sem-ver/)** - Compare two version strings to get the named semantic version.
- **WP_CLI\Utils\parse_ssh_url() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-parse-ssh-url/)** - Parse a SSH url for its host, port, and path.
- **WP_CLI\Utils\basename() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-basename/)** - Locale-independent version of basename()
- **WP_CLI\Utils\isPiped() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-ispiped/)** - Checks whether the output of the current script is a TTY or a pipe / redirect
- **WP_CLI\Utils\proc_open_compat() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-proc-open-compat/)** - Windows compatible `proc_open()`. Works around bug in PHP, and also deals with *nix-like `ENV_VAR=blah cmd` environment variable prefixes.
- **WP_CLI\Utils\esc_like() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-esc-like/)** - First half of escaping for LIKE special characters % and _ before preparing for MySQL.

---

# Identify a Plugin or Theme Conflict

Is WP-CLI not working with your WordPress install, but you don't know why? It might be a plugin or theme conflicting with the WP-CLI load process.

To verify, first run `wp --skip-plugins --skip-themes` to execute WP-CLI without loading plugins or themes. If WP-CLI runs as expected with plugins and themes skipped, then the problem is caused by one of those plugins or the active theme.

To see if the source of the problem is the active theme, run `wp --skip-themes` to see if WP-CLI loads as expected.

To see if the source of the problem is an active plugin, run wp `--skip-plugins=<plugin-slug>` for each active plugin to see which plugin blocks the full execution of WP-CLI. For instance, skipping the Akismet plugin would be:

```
wp --skip-plugins=akismet
```

Or, use `xargs` to try the entire list of active plugins:

```
wp plugin list --field=name --status=active --skip-plugins | xargs -n1 -I % wp --skip-plugins=% plugin get % --field=name
```

---

# External Resources

## Blog posts & tutorials

- scribu: A Command-Line Interface for WordPress (https://scribu.net/wordpress/a-command-line-interface-for-wordpress.html)

- madewithlove: WP-CLI: WordPress Command Line Tools (https://madewithlove.com/blog/wp-cli-wordpress-command-line-tools/)
- Mika Epstein: Command Line WP (https://halfelf.org/2012/command-line-wp/)
- Treehouse Blog: Tame WordPress from the Command Line with wp-cli (https://blog.teamtreehouse.com/tame-wordpress-from-the-command-line-with-wp-cli)
- Matt Wiebe: Why You Should Develop Plugins With wp-cli (https://mattwiebe.wordpress.com/2014/01/15/why-you-should-develop-plugins-with-wp-cli/)
- Torque: Using WP CLI to Set Up a Test Version of Your Site (https://torquemag.io/using-wp-cli-to-set-up-a-test-version-of-your-site/)
- WP Bullet Guides: Useful Bash Scripts Utilizing WP-CLI (https://guides.wp-bullet.com/category/wp-cli/)

# Slides

- Francesco Laffi: WPday Bologna 2013: wp-cli (https://speakerdeck.com/francescolaffi/wpday-bologna-2013-wp-cli)
- Michael Bastos: Command Line WordPress Step by Step (https://docs.google.com/presentation/d/1iDYvmM_52ww_iB4aGMAYL_KJ1pnElAVm40c7IrZtyz8/edit)
- Kelly Dwan: Importing (any!) Content with WP-CLI (https://www.slideshare.net/slideshow/nerds-2014wpcli/39052931/)
- Jaime Martinez: Simplify your day to day tasks with WP-CLI (https://slid.es/jmslbam/wp-cli-wp-meetup010-25-nov-2013)

# Videos

- scribu: Command Line Learnings For Make Benefit Glorious Nation Of WordPress (https://wordpress.tv/2013/02/23/cristi-burka-command-line-learnings-for-make-benefit-glorious-nation-of-wordpress/)
- Mike Schroder: Magical WordPress Management using WP-CLI (https://wordpress.tv/2013/08/06/mike-schroder-magical-wordpress-management-using-wp-cli/)
- Daniel Bachhuber: WordPress at the Command Line (https://wordpress.tv/2012/08/21/daniel-bachhuber-wordpress-at-the-command-line/)
- AJ Morris: Managing WordPress from the Command Line (https://wordpress.tv/2016/03/26/aj-morris-using-wp-cli-to-create-your-own-managed-wordpress-hosting/)
- Shawn Hooper: WP-CLI – Save Time by Managing WordPress from the Command Line (https://wordpress.tv/2015/09/15/shawn-hooper-wp-cli-save-time-managing-command-line/)

---

# Running Commands Remotely

## Using an SSH connection

WP-CLI accepts an `--ssh=[<scheme>:][<user>@]<host>[:<port>][<path>]` global parameter for running a command against a remote WordPress install. This argument works similarly to how the SSH connection is parameterized in tools like `scp` or `git`.

Under the hood, WP-CLI proxies commands to the ssh executable, which then passes them to the WP-CLI installed on the remote machine. The syntax for `--ssh=[<scheme>:][<user>@]<host>[:<port>][<path>]` is interpreted according to the following rules:

- The **scheme** argument defaults to `ssh` and alternately accepts options for `vagrant`, `docker` and `docker-compose`.
- If you provide just the **host** (e.g. `wp --ssh=example.com`), the user will be inferred from your current system user, the port will be the default SSH port (22) and the path will be the SSH user's home directory.
- You can override the **user** by adding it as a prefix terminated by the at sign (e.g. `wp --ssh=admin_user@example.com`).
- You can override the **port** by adding it as a suffix prepended by a colon (e.g. `wp --ssh=example.com:2222`).
- You can override the **path** by adding it as a suffix (e.g. `wp --ssh=example.com~/webapps/production`). The path comes immediately after the port, or after the TLD of the host if you didn't explicitly set a port.
- You can alternatively provide a known **alias**, stored in `~/.ssh/config` (e.g. `wp --ssh=rc` for the `@rc` alias).

**Note: you need to have a copy of WP-CLI installed on the remote server, accessible as `wp`.**

Furthermore, `--ssh=<host>` won't load your `~/.bash_profile` if you have a shell alias defined, or are extending the `$PATH` environment variable. If this affects you, here's a more thorough explanation (https://make.wordpress.org/cli/handbook/running-commands-remotely/#making-wp-cli-accessible-on-a-remote-server) of how you can make `wp` accessible.

### Scheme

You can utilize the scheme component of the ssh argument to define a shorthand for connecting to local containerized or virtualized machines.

The **scheme** argument is set to `ssh` by default, but it also accepts `vagrant`, `docker`, and `docker-compose` as alternate options.

#### docker

To use Docker, the command is: `wp rewrite flush --ssh=docker:<name>` The Docker container's name can be found by using the `docker ps` command.

#### docker-compose

For Docker Compose, the command is: `wp option get home_url --ssh=docker-compose:<name>` The Docker container's name can be located in the `docker-compose.yml` file.

#### vagrant

With Vagrant, you can use the command: `wp rewrite flush --ssh=vagrant:<name>`

# Aliases

WP-CLI aliases are shortcuts you register in your `wp-cli.yml` or `config.yml` to effortlessly run commands against any WordPress install.

For instance, when you are working locally, have registered a new rewrite rule and need to flush rewrites inside of your Vagrant-based virtual machine, you can run:

```
# Run the flush command on the development environment
$ wp @dev rewrite flush
Success: Rewrite rules flushed.
```

Then, once the code goes to production, you can run:

```
# Run the flush command on the production environment
$ wp @prod rewrite flush
Success: Rewrite rules flushed.
```

You don't need to SSH into machines, change directories, and generally spend a full minute to get to a given WordPress install, you can just let WP-CLI know what machine to work with and it knows how to make the actual connection.

It can also easily utilize Vagrant's ssh helper command to figure out the SSH parameters, by piping the WP-CLI command to `vagrant ssh` using the `vagrant` scheme like `--ssh=vagrant:default` where `default` is the Vagrant machine name/id, or if defined as an alias like the examples below. Some Vagrant boxes ship this by default (https://github.com/Chassis/Chassis/tree/main) so you can use WP-CLI from the host machine out-of-the-box.

Additionally, alias groups let you register groups of aliases. If you want to run a command against both configured example sites, you can use a group like `@both`:

```
# Run the update check on both environments
$ wp @both core check-update
Success: WordPress is at the latest version.
Success: WordPress is at the latest version.
```

Aliases can be registered in your project's `wp-cli.yml` file, or your user's global `~/.wp-cli/config.yml` file:

```
@prod:
  ssh: dev_user@example.com~/webapps/production
@dev:
  ssh: vagrant@192.168.50.10/srv/www/example.dev
@local:
  ssh: vagrant:default
@both:
  - @prod
  - @dev
```

You can find more information about how to set up your configuration files in the Config section (https://make.wordpress.org/cli/handbook/config/#config-files).

# Running custom commands remotely

If you want to run a custom command on a remote server, that custom command needs to be installed on the remote server, but it does not have to be installed on the local machine you're launching `wp` from.

You can use the WP-CLI package manager remotely to install custom commands to remote machines.

Example:

```
# The command is not installed on either local or remote machine
$ wp db ack
Error: 'ack' is not a registered subcommand of 'db'. See 'wp help db'.
$ wp @dev db ack
Error: 'ack' is not a registered subcommand of 'db'. See 'wp help db'.

# To make the command work on the remote machine, we can install it remotely
# through the WP-CLI package manager
$ wp @dev package install runcommand/db-ack
Installing package runcommand/db-ack (dev-master)
Updating /home/vagrant/.wp-cli/packages/composer.json to require the package...
Using Composer to install the package...
---
Loading composer repositories with package information
Updating dependencies
Resolving dependencies through SAT
Dependency resolution completed in 0.311 seconds
Analyzed 4726 packages to resolve dependencies
Analyzed 162199 rules to resolve dependencies
Package operations: 1 install, 0 updates, 0 removals
Installs: runcommand/db-ack:dev-master aff8ccc
 - Installing runcommand/db-ack (dev-master aff8ccc)
Writing lock file
Generating autoload files
---
Success: Package installed.

# Now we can run the command remotely, even though it is not installed locally
$ wp @dev db ack test_email@example.com
wp_users:user_email
9:test_email@example.com
```

# Making WP-CLI accessible on a remote server

Running a command remotely through SSH requires having `wp` accessible on the `$PATH` on the remote server. Because SSH connections don't load `~/.bashrc` or `~/.zshrc`, you may need to specify a custom `$PATH` when using `wp --ssh=<host>`. A few ways to make `wp` available on the remote server are:

## Copy WP-CLI binary to $HOME / bin

In many Linux distros, `$HOME/bin` is in the `$PATH` by default, so a way to make `wp` accessible is to create a `$HOME/bin` directory, if it doesn't already exist, and move the WP-CLI binary into `$HOME/bin/wp`:

```
mkdir -p ~/bin
cd ~/bin
curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
chmod +x wp-cli.phar
mv wp-cli.phar wp
```

If `$HOME/bin` is not already in your path, then you can define it in your `~/.bashrc` file or equivalent for your remote server's specific shell:

```
#.bashrc
PATH="$HOME/bin:$PATH"
```

## Specify the $PATH in $HOME / .ssh/ environment

Another way to achieve this is to specify the `$PATH` in the remote machine user's `~/.ssh/environment` file, provided that that machine's `sshd` has been configured with `PermitUserEnvironment=yes` (see OpenSSH documentation (https://en.wikibooks.org/wiki/OpenSSH/Client_Configuration_Files#.7E.2F.ssh.2Fenvironment)).

## Remote non-interactive shell resolves aliases and runs wp as alias with php

Some webhosts are configured very restrictive:

- They do not allow you to execute your own shellscripts, so everything from method WP-CLI binary fails.

- `sshd` is configured with `PermitUserEnvironment=no`, so customizing ~/.ssh/environment has no effect and fails too.
- Also using the before_ssh hook on the client machine will not help you, as in all cases you cannot run `wp` on the remote.

The solution: On the remote configure `~/.bashrc` like this:

### Configure the remote non-interactive shell to resolve aliases

- By default aliases in non-interactive shells are not resolved (https://unix.stackexchange.com/questions/425319/how-do-i-execute-a-remote-alias-over-an-ssh/425323#425323) but you can change that:
    - Somewhere very much on top of your `~/.bashrc` (certainly before the first alias definitions) insert:

```
shopt -s expand_aliases
```

    - Be also sure that there's no mechanism which exits the non-interactive shell too early! If you have something like the following construct on top (e.g. Ubuntu is configured like this) then outcomment or delete all lines of the construct:

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

### Alias wp to php which runs the WP-CLI binary

- Somewhere after the `shopt -s expand_aliases` line insert:
    - `alias wp="php ~/bin/wp"`
    - or
    - `alias wp="php ~/bin/wp-cli.phar"`
- In other words: You have an alias "wp" which is a one liner where php runs the WP-CLI binary by stating the path to the `wpi-cli.phar` file, wherever it may be, under whatever name it may have.
    - `php` is allowed on basically any webhost.
    - And the `wp-cli.phar` file itself must not even have the execute flag set (="can be entirely passive"), as formally the file gets run (=interpreted) by `php`. **Note:** An executable `~/bin/wp` file (the renamed `wp-cli.phar` with an execute flag) of course also gets run by the `php` interpreter eventually. But the invocation in the shell is formally different. And that is what makes the crucial difference here.

## Using before_ssh hook on client machine

Alternatively, in case you cannot make it work from within the server, you can achieve the same effect by hooking into the `before_ssh` hook, and defining an environment variable with the command you'd like to run:

```
WP_CLI::add_hook( 'before_ssh', function() {

    $host = WP_CLI\Utils\parse_ssh_url(
        WP_CLI::get_runner()->config['ssh'],
        PHP_URL_HOST
    );

    switch( $host ) {
        case 'example.com':
            putenv( 'WP_CLI_SSH_PRE_CMD=export PATH=$HOME/bin:$PATH' );
            break;
    }
} );
```

If you put the code above in a `pre-ssh.php` file, you can load it for your entire environment by requiring it from your `~/.wp-cli/config.yml` file:

```
require:
  - pre-ssh.php
```

---

# How to WP-CLI

## Introduction

Using WP-CLI, you can perform specific tasks using the command line instead of accessing the WordPress admin panel. In the following section, we will cover step by step some of these tasks.

- How-to install WordPress (https://make.wordpress.org/cli/handbook/how-to-install/)
- How to put the website in maintenance mode (https://make.wordpress.org/cli/handbook/how-to-maintenance-mode)
- How-to start the webserver (https://make.wordpress.org/cli/handbook/)
- How to create a custom plugin (https://make.wordpress.org/cli/handbook/how-to-create-custom-plugins)

---

# Internal API

WP-CLI includes a number of utilities which are considered stable and meant to be used by commands.

This also means functions and methods not listed here are considered part of the private API. They may change or disappear at any time.

*Internal API documentation is generated from the WP-CLI codebase on every release. To suggest improvements, please submit a pull request.*

---

## Registration

- **WP_CLI::add_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-hook/)** - Schedule a callback to be executed at a certain point.
- **WP_CLI::do_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-do-hook/)** - Execute callbacks registered to a given hook.
- **WP_CLI::add_wp_hook() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-wp-hook/)** - Add a callback to a WordPress action or filter.
- **WP_CLI::add_command() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-command/)** - Register a command to WP-CLI.

## Output

- **WP_CLI\Utils\format_items() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-format-items/)** - Render a collection of items as an ASCII table, JSON, CSV, YAML, list of ids, or count.
- **WP_CLI\Utils\make_progress_bar() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-make-progress-bar/)** - Create a progress bar to display percent completion of a given operation.
- **WP_CLI::colorize() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-colorize/)** - Colorize a string for output.
- **WP_CLI::line() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-line/)** - Display informational message without prefix, and ignore `--quiet`.
- **WP_CLI::log() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-log/)** - Display informational message without prefix.
- **WP_CLI::success() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-success/)** - Display success message prefixed with "Success: ".
- **WP_CLI::debug() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-debug/)** - Display debug message prefixed with "Debug: " when `--debug` is used.
- **WP_CLI::warning() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-warning/)** - Display warning message prefixed with "Warning: ".
- **WP_CLI::error() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-error/)** - Display error message prefixed with "Error: " and exit script.
- **WP_CLI::halt() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-halt/)** - Halt script execution with a specific return code.
- **WP_CLI::error_multi_line() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-error-multi-line/)** - Display a multi-line error message in a red box. Doesn't exit script.

## Input

- **WP_CLI\Utils\launch_editor_for_input() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-launch-editor-for-input/)** - Launch system's $EDITOR for the user to edit some text.
- **WP_CLI\Utils\get_flag_value() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-flag-value/)** - Return the flag value or, if it's not set, the $default value.
- **WP_CLI\Utils\report_batch_operation_results() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-report-batch-operation-results/)** - Report the results of the same operation against multiple resources.
- **WP_CLI\Utils\parse_str_to_argv() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-parse-str-to-argv/)** - Parse a string of command line arguments into an $argv-esqe variable.
- **WP_CLI::confirm() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-confirm/)** - Ask for confirmation before running a destructive operation.
- **WP_CLI::read_value() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-read-value/)** - Read a value, from various formats.
- **WP_CLI::has_config() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-has-config/)** - Confirm that a global configuration parameter does exist.
- **WP_CLI::get_config() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-get-config/)** - Get values of global configuration parameters.

## Execution

- **WP_CLI::launch() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-launch/)** - Launch an arbitrary external process that takes over I/O.
- **WP_CLI::launch_self() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-launch-self/)** - Run a WP-CLI command in a new process reusing the current runtime arguments.

- **WP_CLI::runcommand() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-runcommand/)** - Run a WP-CLI command.
- **WP_CLI::run_command() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-run-command/)** - Run a given command within the current process using the same global parameters.

## System

- **WP_CLI\Utils\get_home_dir() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-home-dir/)** - Get the home directory.
- **WP_CLI\Utils\trailingslashit() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-trailingslashit/)** - Appends a trailing slash.
- **WP_CLI\Utils\normalize_path() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-normalize-path/)** - Normalize a filesystem path.
- **WP_CLI\Utils\get_temp_dir() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-temp-dir/)** - Get the system's temp directory. Warns user if it isn't writable.
- **WP_CLI\Utils\get_php_binary() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-php-binary/)** - Get the path to the PHP binary used when executing WP-CLI.
- **WP_CLI::get_php_binary() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-get-php-binary/)** - Get the path to the PHP binary used when executing WP-CLI.

## Misc

- **WP_CLI\Utils\write_csv() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-write-csv/)** - Write data as CSV to a given file.
- **WP_CLI\Utils\http_request() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-http-request/)** - Make a HTTP request to a remote URL.
- **WP_CLI\Utils\get_named_sem_ver() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-get-named-sem-ver/)** - Compare two version strings to get the named semantic version.
- **WP_CLI\Utils\parse_ssh_url() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-parse-ssh-url/)** - Parse a SSH url for its host, port, and path.
- **WP_CLI\Utils\basename() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-basename/)** - Locale-independent version of basename()
- **WP_CLI\Utils\isPiped() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-ispiped/)** - Checks whether the output of the current script is a TTY or a pipe / redirect
- **WP_CLI\Utils\proc_open_compat() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-proc-open-compat/)** - Windows compatible `proc_open()`. Works around bug in PHP, and also deals with *nix-like `ENV_VAR=blah cmd` environment variable prefixes.
- **WP_CLI\Utils\esc_like() (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-esc-like/)** - First half of escaping for LIKE special characters % and _ before preparing for MySQL.

---

# Identify a Plugin or Theme Conflict

Is WP-CLI not working with your WordPress install, but you don't know why? It might be a plugin or theme conflicting with the WP-CLI load process.

To verify, first run `wp --skip-plugins --skip-themes` to execute WP-CLI without loading plugins or themes. If WP-CLI runs as expected with plugins and themes skipped, then the problem is caused by one of those plugins or the active theme.

To see if the source of the problem is the active theme, run `wp --skip-themes` to see if WP-CLI loads as expected.

To see if the source of the problem is an active plugin, run wp `--skip-plugins=<plugin-slug>` for each active plugin to see which plugin blocks the full execution of WP-CLI. For instance, skipping the Akismet plugin would be:

```
wp --skip-plugins=akismet
```

Or, use `xargs` to try the entire list of active plugins:

```
wp plugin list --field=name --status=active --skip-plugins | xargs -n1 -I % wp --skip-plugins=% plugin get % --field=name
```

---

# External Resources

## Blog posts & tutorials

- scribu: A Command-Line Interface for WordPress (https://scribu.net/wordpress/a-command-line-interface-for-wordpress.html)
- madewithlove: WP-CLI: WordPress Command Line Tools (https://madewithlove.com/blog/wp-cli-wordpress-command-line-tools/)
- Mika Epstein: Command Line WP (https://halfelf.org/2012/command-line-wp/)
- Treehouse Blog: Tame WordPress from the Command Line with wp-cli (https://blog.teamtreehouse.com/tame-wordpress-from-the-command-line-with-wp-cli)
- Matt Wiebe: Why You Should Develop Plugins With wp-cli (https://mattwiebe.wordpress.com/2014/01/15/why-you-should-develop-plugins-with-wp-cli/)
- Torque: Using WP CLI to Set Up a Test Version of Your Site (https://torquemag.io/using-wp-cli-to-set-up-a-test-version-of-your-site/)
- WP Bullet Guides: Useful Bash Scripts Utilizing WP-CLI (https://guides.wp-bullet.com/category/wp-cli/)

# Slides

- Francesco Laffi: WPday Bologna 2013: wp-cli (https://speakerdeck.com/francescolaffi/wpday-bologna-2013-wp-cli)
- Michael Bastos: Command Line WordPress Step by Step (https://docs.google.com/presentation/d/1iDYvmM_52ww_iB4aGMAYL_KJ1pnElAVm40c7IrZtyz8/edit)
- Kelly Dwan: Importing (any!) Content with WP-CLI (https://www.slideshare.net/slideshow/nerds-2014wpcli/39052931/)
- Jaime Martinez: Simplify your day to day tasks with WP-CLI (https://slid.es/jmslbam/wp-cli-wp-meetup010-25-nov-2013)

# Videos

- scribu: Command Line Learnings For Make Benefit Glorious Nation Of WordPress (https://wordpress.tv/2013/02/23/cristi-burka-command-line-learnings-for-make-benefit-glorious-nation-of-wordpress/)
- Mike Schroder: Magical WordPress Management using WP-CLI (https://wordpress.tv/2013/08/06/mike-schroder-magical-wordpress-management-using-wp-cli/)
- Daniel Bachhuber: WordPress at the Command Line (https://wordpress.tv/2012/08/21/daniel-bachhuber-wordpress-at-the-command-line/)
- AJ Morris: Managing WordPress from the Command Line (https://wordpress.tv/2016/03/26/aj-morris-using-wp-cli-to-create-your-own-managed-wordpress-hosting/)
- Shawn Hooper: WP-CLI – Save Time by Managing WordPress from the Command Line (https://wordpress.tv/2015/09/15/shawn-hooper-wp-cli-save-time-managing-command-line/)

---

# Running Commands Remotely

## Using an SSH connection

WP-CLI accepts an `--ssh=[<scheme>:][<user>@]<host>[:<port>][<path>]` global parameter for running a command against a remote WordPress install. This argument works similarly to how the SSH connection is parameterized in tools like `scp` or `git`.

Under the hood, WP-CLI proxies commands to the ssh executable, which then passes them to the WP-CLI installed on the remote machine. The syntax for `--ssh=[<scheme>:][<user>@]<host>[:<port>][<path>]` is interpreted according to the following rules:

- The **scheme** argument defaults to `ssh` and alternately accepts options for `vagrant`, `docker` and `docker-compose`.
- If you provide just the **host** (e.g. `wp --ssh=example.com`), the user will be inferred from your current system user, the port will be the default SSH port (22) and the path will be the SSH user's home directory.
- You can override the **user** by adding it as a prefix terminated by the at sign (e.g. `wp --ssh=admin_user@example.com`).
- You can override the **port** by adding it as a suffix prepended by a colon (e.g. `wp --ssh=example.com:2222`).
- You can override the **path** by adding it as a suffix (e.g. `wp --ssh=example.com~/webapps/production`). The path comes immediately after the port, or after the TLD of the host if you didn't explicitly set a port.
- You can alternatively provide a known **alias**, stored in `~/.ssh/config` (e.g. `wp --ssh=rc` for the `@rc` alias).

**Note: you need to have a copy of WP-CLI installed on the remote server, accessible as `wp`.**

Furthermore, `--ssh=<host>` won't load your `~/.bash_profile` if you have a shell alias defined, or are extending the `$PATH` environment variable. If this affects you, here's a more thorough explanation (https://make.wordpress.org/cli/handbook/running-commands-remotely/#making-wp-cli-accessible-on-a-remote-server) of how you can make `wp` accessible.

### Scheme

You can utilize the scheme component of the ssh argument to define a shorthand for connecting to local containerized or virtualized machines.

The **scheme** argument is set to `ssh` by default, but it also accepts `vagrant`, `docker`, and `docker-compose` as alternate options.

#### docker

To use Docker, the command is: `wp rewrite flush --ssh=docker:<name>` The Docker container's name can be found by using the `docker ps` command.

#### docker-compose

For Docker Compose, the command is: `wp option get home_url --ssh=docker-compose:<name>` The Docker container's name can be located in the `docker-compose.yml` file.

#### vagrant

With Vagrant, you can use the command: `wp rewrite flush --ssh=vagrant:<name>`

# Aliases

WP-CLI aliases are shortcuts you register in your `wp-cli.yml` or `config.yml` to effortlessly run commands against any WordPress install.

For instance, when you are working locally, have registered a new rewrite rule and need to flush rewrites inside of your Vagrant-based virtual machine, you can run:

```
# Run the flush command on the development environment
$ wp @dev rewrite flush
Success: Rewrite rules flushed.
```

Then, once the code goes to production, you can run:

```
# Run the flush command on the production environment
$ wp @prod rewrite flush
Success: Rewrite rules flushed.
```

You don't need to SSH into machines, change directories, and generally spend a full minute to get to a given WordPress install, you can just let WP-CLI know what machine to work with and it knows how to make the actual connection.

It can also easily utilize Vagrant's ssh helper command to figure out the SSH parameters, by piping the WP-CLI command to `vagrant ssh` using the `vagrant` scheme like `--ssh=vagrant:default` where `default` is the Vagrant machine name/id, or if defined as an alias like the examples below. Some Vagrant boxes ship this by default (https://github.com/Chassis/Chassis/tree/main) so you can use WP-CLI from the host machine out-of-the-box.

Additionally, alias groups let you register groups of aliases. If you want to run a command against both configured example sites, you can use a group like `@both`:

```
# Run the update check on both environments
$ wp @both core check-update
Success: WordPress is at the latest version.
Success: WordPress is at the latest version.
```

Aliases can be registered in your project's `wp-cli.yml` file, or your user's global `~/.wp-cli/config.yml` file:

```
@prod:
  ssh: dev_user@example.com~/webapps/production
@dev:
  ssh: vagrant@192.168.50.10/srv/www/example.dev
@local:
  ssh: vagrant:default
@both:
  - @prod
  - @dev
```

You can find more information about how to set up your configuration files in the Config section (https://make.wordpress.org/cli/handbook/config/#config-files).

# Running custom commands remotely

If you want to run a custom command on a remote server, that custom command needs to be installed on the remote server, but it does not have to be installed on the local machine you're launching `wp` from.

You can use the WP-CLI package manager remotely to install custom commands to remote machines.

Example:

```
# The command is not installed on either local or remote machine
$ wp db ack
Error: 'ack' is not a registered subcommand of 'db'. See 'wp help db'.
$ wp @dev db ack
Error: 'ack' is not a registered subcommand of 'db'. See 'wp help db'.

# To make the command work on the remote machine, we can install it remotely
# through the WP-CLI package manager
$ wp @dev package install runcommand/db-ack
Installing package runcommand/db-ack (dev-master)
Updating /home/vagrant/.wp-cli/packages/composer.json to require the package...
Using Composer to install the package...
---
Loading composer repositories with package information
Updating dependencies
Resolving dependencies through SAT
Dependency resolution completed in 0.311 seconds
Analyzed 4726 packages to resolve dependencies
Analyzed 162199 rules to resolve dependencies
Package operations: 1 install, 0 updates, 0 removals
Installs: runcommand/db-ack:dev-master aff8ccc
 - Installing runcommand/db-ack (dev-master aff8ccc)
Writing lock file
Generating autoload files
---
Success: Package installed.

# Now we can run the command remotely, even though it is not installed locally
$ wp @dev db ack test_email@example.com
wp_users:user_email
9:test_email@example.com
```

# Making WP-CLI accessible on a remote server

Running a command remotely through SSH requires having `wp` accessible on the `$PATH` on the remote server. Because SSH connections don't load `~/.bashrc` or `~/.zshrc`, you may need to specify a custom `$PATH` when using `wp --ssh=<host>`. A few ways to make `wp` available on the remote server are:

## Copy WP-CLI binary to $HOME / bin

In many Linux distros, `$HOME/bin` is in the `$PATH` by default, so a way to make `wp` accessible is to create a `$HOME/bin` directory, if it doesn't already exist, and move the WP-CLI binary into `$HOME/bin/wp`:

```
mkdir -p ~/bin
cd ~/bin
curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
chmod +x wp-cli.phar
mv wp-cli.phar wp
```

If `$HOME/bin` is not already in your path, then you can define it in your `~/.bashrc` file or equivalent for your remote server's specific shell:

```
#.bashrc
PATH="$HOME/bin:$PATH"
```

## Specify the $PATH in $HOME / .ssh/ environment

Another way to achieve this is to specify the `$PATH` in the remote machine user's `~/.ssh/environment` file, provided that that machine's `sshd` has been configured with `PermitUserEnvironment=yes` (see OpenSSH documentation (https://en.wikibooks.org/wiki/OpenSSH/Client_Configuration_Files#.7E.2F.ssh.2Fenvironment)).

## Remote non-interactive shell resolves aliases and runs wp as alias with php

Some webhosts are configured very restrictive:

- They do not allow you to execute your own shellscripts, so everything from method WP-CLI binary fails.

- `sshd` is configured with `PermitUserEnvironment=no`, so customizing ~/.ssh/environment has no effect and fails too.
- Also using the before_ssh hook on the client machine will not help you, as in all cases you cannot run `wp` on the remote.

The solution: On the remote configure `~/.bashrc` like this:

### Configure the remote non-interactive shell to resolve aliases

- By default aliases in non-interactive shells are not resolved (https://unix.stackexchange.com/questions/425319/how-do-i-execute-a-remote-alias-over-an-ssh/425323#425323) but you can change that:
    - Somewhere very much on top of your `~/.bashrc` (certainly before the first alias definitions) insert:

```
shopt -s expand_aliases
```

    - Be also sure that there's no mechanism which exits the non-interactive shell too early! If you have something like the following construct on top (e.g. Ubuntu is configured like this) then outcomment or delete all lines of the construct:

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

### Alias wp to php which runs the WP-CLI binary

- Somewhere after the `shopt -s expand_aliases` line insert:
    - `alias wp="php ~/bin/wp"`
    - or
    - `alias wp="php ~/bin/wp-cli.phar"`
- In other words: You have an alias "wp" which is a one liner where php runs the WP-CLI binary by stating the path to the `wpi-cli.phar` file, wherever it may be, under whatever name it may have.
    - `php` is allowed on basically any webhost.
    - And the `wp-cli.phar` file itself must not even have the execute flag set (="can be entirely passive"), as formally the file gets run (=interpreted) by `php`. **Note:** An executable `~/bin/wp` file (the renamed `wp-cli.phar` with an execute flag) of course also gets run by the `php` interpreter eventually. But the invocation in the shell is formally different. And that is what makes the crucial difference here.

## Using before_ssh hook on client machine

Alternatively, in case you cannot make it work from within the server, you can achieve the same effect by hooking into the `before_ssh` hook, and defining an environment variable with the command you'd like to run:

```
WP_CLI::add_hook( 'before_ssh', function() {

    $host = WP_CLI\Utils\parse_ssh_url(
        WP_CLI::get_runner()->config['ssh'],
        PHP_URL_HOST
    );

    switch( $host ) {
        case 'example.com':
            putenv( 'WP_CLI_SSH_PRE_CMD=export PATH=$HOME/bin:$PATH' );
            break;
    }
} );
```

If you put the code above in a `pre-ssh.php` file, you can load it for your entire environment by requiring it from your `~/.wp-cli/config.yml` file:

```
require:
  - pre-ssh.php
```

---

# Bug Reports

Think you've found a bug? We'd love for you to help us get it fixed.

Before you create a new issue, you should:

- Search existing issues (https://github.com/search?q=org%3Awp-cli+label%3Abug+is%3Aopen+sort%3Aupdated-desc&type=issues) to see if there's an existing resolution to it, or if it's already been fixed in a newer version.
- Check our documentation on common issues and their fixes (https://make.wordpress.org/cli/handbook/common-issues/). It's worth reading through the GitHub issues linked on the page, as the error listed may not be exactly the error you're experiencing.
- Reproduce the issue in a fresh installation of WordPress (e.g. Twenty Sixteen or similar, with no plugins active). If the issue only reproduces in a custom environment, then the issue is a bug in your environment, not WP-CLI (make sure `WP_DEBUG` is enabled, which will often give you more visibility into the issue). You may be able to track down the error to a specific plugin or theme (https://make.wordpress.org/cli/handbook/identify-plugin-theme-conflict/).

Once you've done a bit of searching and discovered there isn't prior art for your bug, please create a new GitHub issue in the appropriate repository. Providing the summary, steps to reproduce, environmental details, and other specifics identified below will help guarantee you are submitting a full bug report.

**Summary**

In narrative form, such as “While testing feature x, I encountered z.”

> ***Summary:***
>
> *After installing the following package on newest nightly version of the WP-CLI, I receive "is not a registered wp command error"*

**Steps to reproduce the bug**

Please be as detailed and specific as possible. It's a good idea to go back and follow your steps a second time to make sure another user can recreate the issue by following the steps directly without assuming any actions.

> ***Steps to reproduce:***
>
> 1. *Run command 'wp package install wp-cli/scaffold-package-command:@stable'*
> 2. *Allow installation process to complete*
> 3. *'Success: Package installed.' is displayed*
> 4. *Run 'wp help scaffold package'*
> 5. *Observe 'Error: 'scaffold package' is not a registered wp command.'*

If you have a great deal of output to share, please create a GitHub gist (https://gist.github.com/) and link to it in the issue.

**Environment**

Your bug may also be environment-specific. Because WP-CLI is a tool whose behavior is different from system to system, please include some environmental details in your issue if you think they're relevant.

```
# What PHP environment does WP-CLI run in?
wp cli info
# Are you running suhosin? If so, make sure you've added `suhosin.executor.include.whitelist = phar` to your php.ini
php -m | grep -i suhosin
# Can you share the results of which wp ?
which -a wp
stat $(which wp)
# Are you running any packages? If one is causing a problem, you can use `--skip-packages` to skip loading them
wp package list
```

**Results and impacts**

Explain how the bug affects your usage, including severity level (what were the expected results, what were the actual results? who, how and to what degree affected?).

> ***Severity*** *- High*
>
> ***Expected Results:*** *Running* `wp help scaffold package` *displays help information for installed command.*
>
> ***Actual Results:*** *'Error: 'scaffold package' is not a registered wp command' is displayed.*
>
> *This command is crucial for the daily function of our business. 1.2k other users also have installed this package and will be affected.*
>
> *The installed package works without issue using WP-CLI* `1.2.0-alpha-3182ac4`

**Workarounds**

Do any workarounds exist? If so, what are they?

> ***Workaround:*** *Rollback to working version of WP-CLI*

**Relevant diagnostics**

Include any crash reports, stack traces or debug output relevant to the issue. Any screenshots or gifs of issue are helpful too!

Have a lot of output? Make your issue easier to understand by creating a GitHub gist (https://gist.github.com/) and linking to it in the issue.

If you include the `--debug` flag when executing WP-CLI (e.g. `wp post list --debug`), you may get more verbosity into the source of the error to include in the report.

```
$ wp package install wp-cli/scaffold-package-command:@stable
Installing package wp-cli/scaffold-package-command (@stable)
Updating /root/.wp-cli/packages/composer.json to require the package...
Using Composer to install the package...
---
Loading composer repositories with package information
Updating dependencies
Resolving dependencies through SAT
Dependency resolution completed in 0.001 seconds
Analyzed 421 packages to resolve dependencies
Analyzed 96 rules to resolve dependencies
Package operations: 1 install, 0 updates, 0 removals
Installs: wp-cli/scaffold-package-command:1.2.0
 - Installing wp-cli/scaffold-package-command (1.2.0)
Writing lock file
Generating autoload files
---
Success: Package installed.
```

```
$ wp help scaffold package --debug
Debug (bootstrap): Loading packages from: /root/.wp-cli/packages/vendor/autoload.php (0.044s)
Debug (bootstrap): No readable global config found (0.193s)
Debug (bootstrap): No project config found (0.194s)
Debug (bootstrap): argv: /usr/local/bin/wp help scaffold package --debug (0.194s)
Debug (bootstrap): ABSPATH defined: /home/fu/public_html/ (0.195s)
Debug (bootstrap): Begin WordPress load (0.195s)
Debug (bootstrap): wp-config.php path: /home/fu/public_html/wp-config.php (0.196s)
Debug (bootstrap): Loaded WordPress (1.323s)
Debug (bootstrap): Running command: help (1.323s)
Error: 'scaffold package' is not a registered wp command.
```

*wp scaffold package does not work either:*

```
$ wp scaffold package
Error: 'scaffold package' is not a registered subcommand of 'scaffold'. See 'wp help scaffold'.
```

# Repository Management

## Package Names

Package names for the official `wp-cli` GitHub organization need to be discussed upfront with the maintainers (https://github.com/orgs/wp-cli/teams/maintainers).

Before creating a new repository, it's important to discuss the repository first. This discussion helps establish consensus on what purpose the codebase serves, and where it lives best in the WP-CLI ecosystem.

Package names for WP-CLI commands must end with the `-command` suffix for clarity, and also to distinguish them from the non-command repositories.

Before new repositories are created for commands, a decision needs to be made whether the command deserves its own repository or whether it should be included within an existing repository instead.

***Example:*** *`wp-cli/scaffold-package-command`.*

# Package Versions

Package versions follow the Semantic Versioning Specification (SemVer) (http://semver.org/).

In layman's terms for us, this typically means:

- *Patch releases* (e.g. 1.0.x) are used for bug fixes, documentation changes, small enhancements, etc.
- *Minor releases* (e.g. 1.x.0) are used when a new command is introduced to a package, or a major enhancement is made to an existing command.
- *Major releases* (e.g. x.0.0) are used when changes to the framework change the WP-CLI API in a breaking way.

# Milestones

Open milestones usually point to the next version to be released.

As the release notes are built using the milestones, both pull requests and issues that are to be merged need to be set to the upcoming milestone, so that these pull requests will be part of the release notes.

# Labels

Labels are an important organizational tool to communicate the state and progress of the work being done. Committers (https://github.com/orgs/wp-cli/teams/committers) are required to keep these up-to-date.

Keeping issues and pull requests accurately labeled helps us make sense of issue tracker usage in aggregate and more easily search closed issues and pull requests for past discussion.

## Groups Of Labels

Labels can be part of a label group, a concept that is applicable to all packages. The actual labels that are available can depend on the actual package, but the groups always have the same semantic role.

### Command

The labels that define what exact command a given issue/pull request applies to are prefixed with `command:`

***Example:*** *`command:cli-update`*

### Scope

The labels that define what scope the current issue/pull request applies to are prefixed with `scope:`.

Used scopes:

- `scope:bootstrap` - Part of the bootstrap process, which loads both WP-CLI as well as WordPress Core.
- `scope:distribution` - Part of the distribution process, where the Phar is being built and releases are produced.
- `scope:documentation` - Part of the handbook, command reference or inline help.
- `scope:framework` - Part of the WP-CLI framework itself, which provides the architecture, API and helper functions to make commands possible.
- `scope:testing` - Part of the unit or functional tests.
- `scope:website` - Part of the website infrastructure on `wp-cli.org` or `make.wordpress.org`.

### State

The labels that defined what state a given issue/pull request is in are prefixed with `state:`.

Used states:

- `state:unconfirmed` - The bug/problem in the issue could not be replicated yet or might be related to the reporter's environment.

- `state:unsupported` - The issue is outside of the scope of a bug report and cannot be supported on GitHub. The reporter should be pointed towards one of the support channels (http://wp-cli.org/#support).

### Command

The labels that define what specific command a given issue/pull request is related to are prefixed with `command:`.

Some examples:

- `command:core` - Relates to one or more of the subcommands under the `wp core` parent command.
- `command:cli-check-update` - Relates to the `wp cli check-update` command.
- `command:post-meta-update` - Relates to the `wp post-meta update` command.

## Required Labels

Some labels have a special meaning and/or might be used for automated workflows down the road. These are required across all official packages.

### Bug

The `bug` label denotes an issue that is a confirmed bug where the code does not produce the expected result.

Bugs are specific to unexpected behavior of existing functionality. Functionality perceived to be missing is not a bug, but an enhancement.

### Upstream Bug

The `upstream-bug` label denotes an issue that is a bug in upstream software (e.g. WordPress core, PHP, etc.) that won't be fixed in WP-CLI.

### Good First Issue

The `good-first-issue` label denotes an issue that is a good entry point for new contributors wanting to get their feet wet.

Good first issues are small in scope and don't require extensive technical expertise or historical project knowledge to get started.

# Commits

No direct commits are to be done against the default branch (`main`/`master`) branch of the packages. All code changes need to go through a pull request workflow.

# Pull Requests

All code changes go through a pull request workflow.

Every submitted pull request needs to go through a code review (https://make.wordpress.org/cli/handbook/code-review/), and needs to be approved by at least one of the committers (https://github.com/orgs/wp-cli/teams/committers).

## Merge Pre-Conditions

Non-trivial pull requests should be preceded by a related issue that defines the problem to solve and allows for discussion of the most appropriate solution before actually writing code.

If a pull request is submitted by one of the committers (https://github.com/orgs/wp-cli/teams/committers), the submitter should set the "Reviewers" for that pull request to `wp-cli/committers` if a general code review is needed, or to one or more specific committer profiles if the expertise of a specific person is needed/wanted.

If a pull request is submitted by an external contributor, the committers (https://github.com/orgs/wp-cli/teams/committers) should be responsive and provide quick feedback to encourage further contributions.

## Unmergeable Contributions

Sometimes, a pull request may not be mergeable, no matter how much additional effort is applied to it (e.g. out of scope, etc.). In these cases, it's best to let the contributor down as softly and firmly as possible, both to encourage future involvement and avoid flame wars.

Make sure to:

1. Thank the contributor for their time and effort.
2. Fully explain the reasoning behind the decision to close the pull request.
3. Link to as much supporting documentation as possible.

If you'd like a template to follow:

> *Thanks _____ for the time you've spent on this pull request.*
>
> *I'm closing this pull request because _____. To clarify further, _____.*
>
> *For more details, please see _____ and _____.*

## Merge Conditions

Apart from needing to be approved, pull requests also need to have their tests in a passing state before they can be merged. When both of these conditions are true, any committer (https://github.com/orgs/wp-cli/teams/committers) can merge the pull request by:

1. Ensuring that all applicable labels have been set.
2. Ensuring that the correct milestone has been set.
3. Ensuring that the branch is deleted after the merge (if applicable).
4. Adapting the title of the pull-request as needed. The ideal pull request title is one that can be copied and pasted into release notes as is.

Once a pull request is merged, the committer should make sure any corresponding issues are also closed and assigned to the correct milestone.

# Releases

## WP-CLI Releases

WP-CLI releases are scheduled and communicated well in advance by the maintainers (https://github.com/orgs/wp-cli/teams/maintainers).

When the release is ready, it is prepared and executed according to the release checklist (https://make.wordpress.org/cli/handbook/contributions/release-checklist/) by one of the maintainers (https://github.com/orgs/wp-cli/teams/maintainers).

## Package Releases

Package releases happen on an as-needed basis and are executed by the maintainers (https://github.com/orgs/wp-cli/teams/maintainers). A package is tagged for a release when:

- It has a reasonable number of changes that have been sitting for a while (couple weeks or more) and would benefit from landing in the nightly build.
- It has a substantial enhancement (e.g. a new command) that's worth getting into the nightly build for pre-release validation.

Tagging a release should be postponed (for a few to several days) if there's an open pull request that makes sense to include in the release.

Package release notes should include the subject of and a link to each merged pull request. It's not necessary to include minor documentation, coding standard, and test suite changes. Feel free to edit the subject for clarity. See wp-cli/scaffold-command v1.0.6 (https://github.com/wp-cli/scaffold-command/releases/tag/v1.0.6) for an example.

Importantly, tagging a package release is also an opportunity to perform a final review of product quality:

- Each merged pull request makes sense to ship, and has sufficient test coverage and documentation.
- The package's README.md is up-to-date.
- No breaking changes were introduced, unless explicitly intended.

Sometime after a new package version is tagged, the `wp-make-coffee` bot submits a pull request against `wp-cli/wp-cli` with the results of `composer update`. When this pull request is merged, the package release will end up in the WP-CLI nightly build.

# Tips & Tools

## GitHub Searches

There are a lot of useful ways to search GitHub. These searches can be put into bookmarks or embedded into pages as a link.

Here are some examples:

- Open issues & pull requests across all WP-CLI repositories, sorted by last updated (https://github.com/search?q=org%3Awp-cli+is%3Aopen+sort%3Aupdated-desc&type=issues)
- Merged pull requests across all WP-CLI repositories that have no milestone, sorted by last updated (https://github.com/search?q=org%3Awp-cli+is%3Amerged+no%3Amilestone+sort%3Aupdated-desc&type=issues)
- Closed issues & pull requests across all WP-CLI repositories that have no label, sorted by last updated (https://github.com/search?q=org%3Awp-cli+is%3Aclosed+no%3Alabel+sort%3Aupdated-desc&type=issues)

---

# Hosting Companies

The following is a list of hosting companies that have WP-CLI installed by default for their customers, i.e. when they first SSH into their server, the `wp` command is already available.

In alphabetical order:

- 2020Media (https://www.2020media.com/)
- 20i (https://www.20i.com/)
- A2 Hosting (https://www.a2hosting.com/)
- Alldomains.Hosting (https://alldomains.hosting/webhosting-server.html)
- Altis Cloud Platform (https://www.altis-dxp.com/)
- AltusHost (https://www.altushost.com/)
- Amimoto (https://amimoto-ami.com/)
- BigCloudy (https://bigcloudy.com/)
- Bluehost (https://www.bluehost.com/)
- Hypernode (https://www.hypernode.com/nl/)
- CloudHostWorld (https://www.cloudhostworld.com)
- Cloudways (https://www.cloudways.com/en/)
- Convesio (https://convesio.com)
- Curanet (https://curanet.dk)
- D9 Hosting (https://d9.hosting)
- DanDomain (https://dandomain.dk)
- DataPerk (https://dataperk.com)
- Dhosting.pl (https://dhosting.pl/)
- dinahosting (https://dinahosting.com/)
- Domaintechnik.AT (https://www.domaintechnik.at/hosting-oesterreich.html)
- Dreamhost (https://www.dreamhost.com/)
- Fasthosts (https://www.fasthosts.co.uk/)
- GoDaddy (https://www.godaddy.com/help/use-wp-cli-to-manage-your-site-12066)
- GreenGeeks (https://www.greengeeks.com/tutorials/wp-cli/)
- Grid Hosting (https://gridhosting.co.uk/)
- Hoasted (https://www.hoasted.com/reseller/)
- Hosting4Agency (https://www.hosting4agency.com/news/wp-cli-come-usarlo/)
- Host4Geeks (https://host4geeks.com/managed-wordpress-hosting)
- Hosterion (https://www.hosterion.com)
- HostGator (https://www.hostgator.com/)
- Hostico (https://hostico.ro)
- Hostinger (https://www.hostinger.com)
- HostPapa (https://www.hostpapa.com)
- HostPresto (https://hostpresto.com)
- HostRiver (https://hostriver.ro/gazduire-wordpress)
- ICDSoft (https:///www.icdsoft.com)
- Infomaniak (https://www.infomaniak.com/en/create-a-website/wordpress-hosting)
- IONOS (https://www.ionos.com/)
- JDM.pl (https://jdm.pl)
- KDAWS.com (https://kdaws.com)
- Kinsta (https://kinsta.com)
- KnownHost (https://www.knownhost.com)
- Krystal Hosting (https://krystal.io/)
- LH.pl (https://www.lh.pl/hosting)
- lima-city (https://www.lima-city.de/webhosting/wordpress)
- Liquid Web (https://liquidweb.com/wordpress)
- ManagedWPHosting (https://www.managedwphosting.nl/)
- MinHost (https://minhost.no)
- Mittwald (https://www.mittwald.de/)
- Monarobase (https://monarobase.net/wordpress)
- MonsterMegs (https://www.monstermegs.com)
- NameHero (https://www.namehero.com)
- NearlyFreeSpeech.NET (https://www.nearlyfreespeech.net/)
- Nexcess.Net (https://www.nexcess.net/)
- Oderland Hosting Services (https://www.oderland.com/)
- One.com (https://www.one.com/)
- Pagely (https://pagely.com/)
- Pantheon (https://pantheon.io)
- Rad Web Hosting (https://radwebhosting.com)
- Rocket.net (https://rocket.net)
- RoseHosting (https://www.rosehosting.com)
- Savvii (https://www.savvii.com/)

- ScanNet (https://www.scannet.dk)
- Seravo.com (https://seravo.com)
- Servebolt.com (https://servebolt.com)
- Simplenet (https://simplenet.io)
- Simply.com (https://www.simply.com)
- Site5 (https://www.site5.com/)
- SiteDistrict (https://sitedistrict.com/)
- SiteGround (https://www.siteground.com/)
- SpinupWP (https://spinupwp.com/)
- The Email Shop (https://theemailshop.co.uk/)
- Templ (https://templ.io/)
- True-False Hosting (https://truefalsehosting.com)
- TrulyWP (https://trulywp.com/)
- TVC.Net (https://tvcnet.com/)
- UNLIMITED.RS (https://unlimited.rs/)
- Veerotech (https://www.veerotech.net/)
- Voteq - Website Solutions (https://voteq.co.uk/)
- Wannafind (https://www.wannafind.dk)
- WNPower Hosting (https://www.wnpower.com/)
- WordPress.com (https://wordpress.com/)
- WP Engine.com (https://wpengine.com)
- WP Provider (https://wpprovider.nl/)
- XEL (https://xel.nl)
- Zenbox (https://zenbox.pl)
- Zenith Media Canada (https://zenithmedia.ca/wordpress-website-hosting/)

The following is a list of hosting companies that use WP-CLI, whereby their customers can write and request the running of standard and custom WP-CLI commands:

- Pressidium (https://pressidium.com)
- Presslabs (https://www.presslabs.com)
- WordPress VIP (https://wpvip.com/)

---

# Troubleshooting Guide

Before you start to report a new issue on the GitHub repository, make sure to check your local installation, as some settings may typically result in an unexpected outcome of your WP-CLI commands.

## How do I get more verbose informations about my WP-CLI installation?

WP-CLI offers the command `wp --info`, which provides you with a lot of information about your WP-CLI install environment. The output will tell you,

- what operating system and shell you work on
- which PHP binary is used to run WP-CLI
- what version of PHP is used
- where to find WP-CLI's root directory
- where to find the vendor directory of WP-CLI
- which WP-CLI binary you are currently working with (phar path)
- where WP-CLI packages are stored
- where to find global and project configuration files for WP-CLI and
- which version of WP-CLI you use.

## What should I do, before I start debugging issues?

Before you start to debug issues, make sure you are using the latest version of WP-CLI. The latest version may already have solved an issue you experience. The command `wp cli update` will upgrade your WP-CLI version or confirm you already use the latest version. If the installation hangs, please ensure that you are allowed to connect to GitHub using SSL (port 443) and git (port 9418) for outbound connections.

## What should I do if the WP-CLI output is different than expected?

Before starting to investigate a bug, you should be aware of the factors that can change the default behavior of WP-CLI and how you can check whether they might be at the root of the issue. There are five main subsystems for modifying this default behavior: environment variables, configuration files, WP-CLI packages, `wp-config.php` file and WordPress extensions (plugins, themes, must-use plugins, drop-ins).

### Environment Variables

The setup of your working environment is a prerequisite for running WP-CLI. You can check your current environment settings with the shell command `env`.

If you want to run WP-CLI remotely using SSH, it is required that the command `wp` is accessible on the path of the remote server. WP-CLI's behavior can also be changed at runtime through the use of environment variables:

- `WP_CLI_CACHE_DIR` – Directory to store the WP-CLI file cache. Default is `~/.wp-cli/cache/`.
- `WP_CLI_CONFIG_PATH` – Path to the global config.yml file. Default is `~/.wp-cli/config.yml`.
- `WP_CLI_DISABLE_AUTO_CHECK_UPDATE` – Disable WP-CLI automatic checks for updates.
- `WP_CLI_PACKAGES_DIR` – Directory to store packages installed through WP-CLI's package management. Default is `* ~/.wp-cli/packages/`.
- `WP_CLI_PHP – PHP` binary path to use when overriding the system default (only works for non-Phar installation).
- `WP_CLI_PHP_ARGS` – Arguments to pass to the PHP binary when invoking WP-CLI (only works for non-Phar installation).
- `WP_CLI_SSH_PRE_CMD` – When using `--ssh=<ssh>`, perform a command before WP-CLI calls WP-CLI on the remote server.
- `WP_CLI_STRICT_ARGS_MODE` – Avoid ambiguity by telling WP-CLI to treat any arguments before the command as global, and after the command as local.

To set an environment variable on demand, you can place the environment variable definition before the WP-CLI command you mean to run (e.g. `EDITOR=vim wp post edit 1`); to overwrite environment variables, use `export VARIABLE=value` in your `~/.bashrc` or `~.zhsrc`.

### WP-CLI Configuration Files

Configuration files let you customize the behavior of WP-CLI to adapt it to your personal needs or those of your project. If the configuration file is incorrect or includes unwanted modifications to the default behavior, the output of WP-CLI will most likely reflect the error.

Note that a project configuration file can override settings in a global configuration file.

Rename or delete your configuration file(s) and compare the result, to find out if a configuration setting might have caused the issue.

### WP-CLI Packages

WP-CLI packages are community-maintained projects built on WP-CLI. They can contain WP-CLI commands, but they can also just extend WP-CLI in some way. While WP-CLI might be perfectly working, a package with errors can cause unexpected results, too. To skip loading all installed packages, use `wp --skip-packages`.

### WordPress Configuration File (`wp-config.php`)

Errors may result from moving or editing `wp-config.php` beyond what WP-CLI supports. If you get a parse error, check the file encoding of your `wp-config.php` (UTF-8 without BOM).

Make sure that the line `require_once(ABSPATH . 'wp-settings.php');` remains in the `wp-config.php` file and don't modify `wp-config.php` beyond constant definitions. If you call WordPress functions within `wp-config.php`, PHP will fail with a fatal error.

If you want to use `$_SERVER['HTTP_HOST']` in your `wp-config.php`, you'll need to set a default value in WP-CLI context:

```
if ( defined( 'WP_CLI' ) && WP_CLI && ! isset( $_SERVER['HTTP_HOST'] ) ) {
    $_SERVER['HTTP_HOST'] = `example.com';
}
```

Instead of `$_SERVER['document_root']` use `dirname( __FILE__ )` or similar.

### WordPress Extensions

WordPress Themes and Plugins might conflict with the loading process of WP-CLI or interfere e.g. by redirecting users. They often make assumptions that are not true in the WP-CLI context, like using the hostname of the current request (which is a concept that does not exist within command-line context).

You can bypass single plugins and themes (e.g. `--skip-plugins=akismet`) or skip them entirely (`wp --skip-plugins --skip-themes`).

## What should I do if WP-CLI reports an error?

Common problems are the result of changes in your `wp-config.php` or web server configuration. The section "Common issues (https://make.wordpress.org/cli/handbook/common-issues/)" of the handbook lists the most common error messages and explains how to handle these errors.

Try to reproduce the issue in a fresh installation of WordPress with a default Theme (Twenty …) and no plugins installed. If the issue only reproduces in a custom environment, then the issue is a bug in your environment, not WP-CLI. Also make sure to enable WordPress' debugging mode by setting the constant `define( 'WP_DEBUG', true );` in your `wp-config.php`. It may reveal problems with your existing WordPress installation, not caused by WP-CLI.

Additionally you can use WP-CLI's global parameter `--debug` to show all PHP errors and add verbosity to WP-CLI bootstrap.

## I have checked all above, but still have an issue. Where can I report issues?

If you think you've found a bug, we'd love to hear from you to get it fixed.

Bug reporting for WP-CLI is handled on GitHub. Before you create a new issue, please search existing issues (https://github.com/search?q=org%3Awp-cli+label%3Abug+is%3Aopen+sort%3Aupdated-desc&type=issues) to see if there's an existing resolution to it. If there isn't an open or fixed issue for your bug, please follow our guidelines for submitting a bug report (https://make.wordpress.org/cli/handbook/bug-reports/) to make sure it gets addressed in a timely manner. Providing the summary, steps to

reproduce, environmental details, and other specifics identified below will help guarantee you are submitting a full bug report.

Please provide us with:

- a summary of the issue in narrative form,
- a detailed and specific list of steps to reproduce the issue,
- details of the environment you're working on,
- a description, how the bug affects your usage (i.e. expected results compared with actual results), including * severity level,
- possible workarounds and
- relevant diagnostics, such as crash reports, stack traces or debug output.

You can find a verbose description of these details of an issue report in the guidelines for submitting a bug report (https://make.wordpress.org/cli/handbook/bug-reports/).

---

# Implementation Details

This page contains some history on various implementation details of WP-CLI.

## Bootstrapping WordPress

On a normal web request, your web server calls the `index.php` file in the root of the web directory to bootstrap the WordPress load process:

```
<?php
/**
 * Front to the WordPress application. This file doesn't do anything, but loads
 * wp-blog-header.php which does and tells WordPress to load the theme.
 *
 * @package WordPress
 */

/**
 * Tells WordPress to load the WordPress theme and output it.
 *
 * @var bool
 */
define('WP_USE_THEMES', true);

/** Loads the WordPress Environment and Template */
require( dirname( __FILE__ ) . '/wp-blog-header.php' );
```

You'll notice `index.php` calls `wp-blog-header.php`, which then calls `wp-load.php`, which then calls `wp-config.php`, which then calls `wp-settings.php`.

This last file, `wp-settings.php`, is WordPress' primary bootstrap file. It loads your plugins, active theme, and calls the `init` action.

On the command line, WP-CLI follows a similar process to bootstrap WordPress. However, instead of loading `index.php`, using the `wp` command starts with this:

```
<?php

// Can be used by plugins/themes to check if WP-CLI is running or not
define( 'WP_CLI', true );
define( 'WP_CLI_VERSION', trim( file_get_contents( WP_CLI_ROOT . '/VERSION' ) ) );
define( 'WP_CLI_START_MICROTIME', microtime( true ) );

// Set common headers, to prevent warnings from plugins
$_SERVER['SERVER_PROTOCOL'] = 'HTTP/1.0';
$_SERVER['HTTP_USER_AGENT'] = '';
$_SERVER['REQUEST_METHOD'] = 'GET';
$_SERVER['REMOTE_ADDR'] = '127.0.0.1';

include WP_CLI_ROOT . '/php/utils.php';
include WP_CLI_ROOT . '/php/dispatcher.php';
include WP_CLI_ROOT . '/php/class-wp-cli.php';
include WP_CLI_ROOT . '/php/class-wp-cli-command.php';

\WP_CLI\Utils\load_dependencies();

WP_CLI::get_runner()->start();
```

WP-CLI includes a good amount of setup code prior to calling `wp-settings.php`. Its bootstrapping process is different than a web request in a couple of notable ways.

## `wp-config.php` is parsed, and then executed

Rather than calling `wp-config.php` directly, WP-CLI gets the contents of `wp-config.php`, parses out the `require_once ABSPATH . 'wp-settings.php';` statement, and loads the constants into scope with `eval()`. Read "How WP-CLI loads WordPress (https://make.wordpress.org/cli/2013/10/24/how-wp-cli-loads-wordpress/)" for a narrative on the historical reasons. After that, WP-CLI used a custom `wp-settings-cli.php` until v0.24.0 [#2278 (https://github.com/wp-cli/wp-cli/issues/2278)], but parsing `wp-config.php` was kept for backwards compatibility purposes. See also #1631 (https://github.com/wp-cli/wp-cli/issues/1631).

## WordPress is loaded inside of a function

WP-CLI loads WordPress with the `WP_CLI::get_runner()->load_wordpress()` method, meaning WordPress plugins and themes aren't loaded in global scope. Any global variables used in plugins or themes need to be explicitly globalized. See #2089 (https://github.com/wp-cli/wp-cli/issues/2089) for the history of this decision.

Once `WP_CLI::get_runner()->load_wordpress()` calls `wp-settings.php`, WordPress handles the rest of the bootstrap process.

# Command Help Text

The `wp help <command>` has been through several incarnations.

Since WP-CLI 0.3, it invoked a static `help()` method in the command class, if it existed. (48a8887d (https://github.com/wp-cli/wp-cli/commit/48a8887d46be25e0c0ad326975729ec816c17331))

Since WP-CLI 0.6, it looked for a `<command>.1` ROFF file and displayed it using `man`. The ROFF file was compiled from a corresponding `<command>.txt` markdown file and from PHPDoc metadata. (#24 (https://github.com/wp-cli/wp-cli/issues/24)).

Since WP-CLI 0.11, it generates the help text on the fly. (#548 (https://github.com/wp-cli/wp-cli/pull/548))

# WP_ADMIN

Most WP-CLI commands perform administrative actions and they need access to code defined in `wp-admin/includes`. This code can be loaded on-demand or preemptively.

The question is: should the `WP_ADMIN` constant be set to `true` or `false`?

Initially, WP-CLI just loaded the wp-admin code and didn't mess with the WP_ADMIN constant at all.

Then, it sort of pretended it was doing a front-end page load, for doing integration testing (#69 (https://github.com/wp-cli/wp-cli/issues/69)). [1]

Then it pretended it was loading wp-admin, to side-step caching plugins (#164 (https://github.com/wp-cli/wp-cli/issues/164)).

Then it stopped pretending it was loading wp-admin (#352 (https://github.com/wp-cli/wp-cli/issues/352)), because we found a better way to side-step caching plugins. [2]

---

- [1]: It turned out that the official WordPress testing suite had a better solution: the `go_to()` method.
- [2]: The solution was rolling our own `wp-settings.php` file.

# Common Issues

## Error: Can't connect to the database

A few possibilities:

a. you're using MAMP, but WP-CLI is not using the MAMP PHP binary.

You can check which PHP WP-CLI is using by running `wp --info`.

If you need to specify an alternate PHP binary, see using a custom PHP binary (https://make.wordpress.org/cli/handbook/installing/#using-a-custom-php-binary).

b. it's a WordPress multisite install.

c. the database credentials in `wp-config.php` are actually incorrect.

## Running `wp --info` produces HTML output

If you run `wp --info` on a server with Phar support disabled, you may see:

```
$ wp --info
Content-type: text/html; charset=UTF-8
```

When using the WP-CLI Phar, you'll need to whitelist Phar support in your `php.ini`:

```
suhosin.executor.include.whitelist = phar
```

## PHP Fatal error: Cannot redeclare wp_unregister_GLOBALS()

If you get this fatal error running the `wp` command, you may have moved or edited `wp-config.php` beyond what wp-cli supports:

```
PHP Fatal error: Cannot redeclare wp_unregister_GLOBALS() (previously declared in /var/www/foo.com/wp-includes/load.php:18) in /var/
```

One of WP-CLI's requirements is that the line:

```
require_once(ABSPATH . 'wp-settings.php');
```

remains in the `wp-config.php` file, so if you've modified or moved it, put it back there. It gets matched by a regex when WP-CLI runs.

## PHP Fatal error: Call to undefined function `<WordPress function>`

Before WP-CLI can load `wp-settings.php`, it needs to know all of the constants defined in `wp-config.php` (database connection details and so on). Because WP-CLI doesn't want WordPress to load yet when it's pulling the constants (https://github.com/wp-cli/wp-cli/blob/main/php/wp-cli.php#L22) out of `wp-config.php`, it uses regex (https://github.com/wp-cli/wp-cli/blob/main/php/WP_CLI/Runner.php#L324) to strip the `require_once(ABSPATH . 'wp-settings.php');` statement.

If you've modified your `wp-config.php` in a way that calls WordPress functions, PHP will fail out with a fatal error, as your `wp-config.php` is calling a WordPress function before WordPress has been loaded to define it.

Example:

```
$ wp core check-update
PHP Fatal error:  Call to undefined function add_filter() in phar:///usr/local/bin/wp/php/WP_CLI/Runner.php(952) : eval()'d code on
```

Modifying `wp-config.php` beyond constant definitions is not best practice. You should move any modifications to a WordPress mu-plugin (https://codex.wordpress.org/Must_Use_Plugins), which will retain the functionality of your modifications while allowing wp-cli to parse your wp-config.php without throwing a PHP error, as well as preventing other errors.

See: #1631 (https://github.com/wp-cli/wp-cli/issues/1631)

## PHP Fatal error: Call to undefined function cli\posix_isatty()

Please ensure you have the php-process extension installed. For example for Centos 6: `yum install php-process`

## PHP Fatal error: Allowed memory size of 999999 bytes exhausted (tried to allocate 99 bytes)

If you run into a PHP fatal error relating to memory when running `wp package install`, you're likely running out of memory.

WP-CLI uses Composer under the hood to manage WP-CLI packages. However, Composer is a bit of a memory hog, so you'll need to increase your memory limit to accommodate it.

Edit your `php.ini` as a permanent fix:

```
# Find your php.ini for PHP-CLI
$ php -i | grep php.ini
Configuration File (php.ini) Path => /usr/local/etc/php/7.0
Loaded Configuration File => /usr/local/etc/php/7.0/php.ini
# Increase memory_limit to 512M or greater
$ vim /usr/local/etc/php/7.0/php.ini
memory_limit = 512M
```

Set `memory_limit` on the fly as a temporary fix:

```
$ php -d memory_limit=512M "$(which wp)" package install <package-name>
```

If your PHP process is still running out of memory, try these steps:

1. Restart PHP.
2. Check for additional php.ini files:

```
$ php -i | grep additional
Scan this dir for additional .ini files => /usr/local/etc/php/7.1/conf.d
# Edit the additional file(s) and increase the memory_limit to 512M or greater
$ vim /usr/local/etc/php/7.1/conf.d
memory_limit = 512M
```

## Error: YIKES! It looks like you're running this as root.

Running WP-CLI as root is extremely dangerous. When you execute WP-CLI as root, any code within your WordPress instance (including third-party plugins and themes you've installed) will have full privileges to the entire server. This can enable malicious code within the WordPress instance to compromise the entire server.

The WP-CLI project strongly discourages running WP-CLI as root.

See also: #973 (https://github.com/wp-cli/wp-cli/pull/973#issuecomment-35842969)

## PHP notice: Undefined index on `$_SERVER` superglobal

The `$_SERVER` superglobal is an array typically populated by a web server with information such as headers, paths, and script locations. PHP CLI doesn't populate this variable, nor does WP-CLI, because many of the variable details are meaningless at the command line.

Before accessing a value on the `$_SERVER` superglobal, you should check if the key is set:

```
if ( isset( $_SERVER['HTTP_X_FORWARDED_PROTO'] ) && 'https' === $_SERVER['HTTP_X_FORWARDED_PROTO'] ) {
  $_SERVER['HTTPS']='on';
}
```

When using `$_SERVER['HTTP_HOST']` in your `wp-config.php`, you'll need to set a default value in WP-CLI context:

```
if ( defined( 'WP_CLI' ) && WP_CLI && ! isset( $_SERVER['HTTP_HOST'] ) ) {
    $_SERVER['HTTP_HOST'] = 'example.com';
}
```

See also: #730 (https://github.com/wp-cli/wp-cli/issues/730)

## PHP notice: Use of undefined constant STDOUT

The `STDOUT` constant is defined by the PHP CLI. If you receive an error notice that `STDOUT` is missing, it's likely because you're not running WP-CLI by PHP CLI. Please review your server configuration accordingly.

## PHP Parse error: syntax error, unexpected '?' in ... /php/WP_CLI/Runner.php ... eval()'d code on line 1

If you get this error running the `wp` command, the most likely cause is a Unicode `BOM` (https://en.wikipedia.org/wiki/Byte_order_mark) at the start of your `wp-config.php`. This issue will be addressed in a future release of WP-CLI, but in the meantime you can solve the issue by running:

```
$ sed -i '1s/^\xEF\xBB\xBF//' $(wp config path)
```

or by manually removing the `BOM` using your favorite editor.

See also: wp-cli/search-replace-command#71 (https://github.com/wp-cli/search-replace-command/issues/71)

## Can't find wp-content directory / use of `$_SERVER['document_root']`

`$_SERVER['document_root']` is defined by the webserver based on the incoming web request. Because this type of context is unavailable to PHP CLI, `$_SERVER['document_root']` is unavailable to WP-CLI. Furthermore, WP-CLI can't safely mock `$_SERVER['document_root']` as it does with `$_SERVER['http_host']` and a few other `$_SERVER` values.

If you're using `$_SERVER['document_root']` in your `wp-config.php` file, you should instead use `dirname( __FILE__ )` or similar.

See also: #785 (https://github.com/wp-cli/wp-cli/issues/785)

## Conflict between global parameters and command arguments

All of the global parameters (https://make.wordpress.org/cli/handbook/references/config/#global-parameters) (e.g. `--url=<url>`) may conflict with the arguments you'd like to accept for your command. For instance, adding a RSS widget to a sidebar will not populate the feed URL for that widget:

```
$ wp widget add rss sidebar-1 1 --url="http://www.smashingmagazine.com/feed/" --items=3
Success: Added widget to sidebar.
```

- Expected result: widget has the feed URL set.
- Actual result: widget is added with the number of items set to 3, but with empty feed URL.

Use the `WP_CLI_STRICT_ARGS_MODE` environment variable to tell WP-CLI to treat any arguments before the command as global, and after the command as local:

```
WP_CLI_STRICT_ARGS_MODE=1 wp --url=wp.dev/site2 widget add rss sidebar-1 1 --url="http://wp-cli.org/feed/"
```

In this example, `--url=wp.dev/site2` is the global argument, setting WP-CLI to run against 'site2' on a WP multisite install. `--url="http://wp-cli.org/feed/"` is the local argument, setting the RSS feed widget with the proper URL.

See also: #3128 (https://github.com/wp-cli/wp-cli/pull/3128)

## Warning: Some code is trying to do a URL redirect

Most of the time, it's some plugin or theme code that disables wp-admin access to non-admins.

Quick fix, other than disabling the protection, is to pass the user parameter: `--user=some_admin`

See also: #477 (https://github.com/wp-cli/wp-cli/issues/477)

## Cannot create a post with Latin characters in the title on Windows

Considering the following example:

```
wp post create --post_title="Perícias Contábeis"
```

Using UTF-8 in PHP arguments doesn't work on Windows for PHP <= 7.0, however it will work for PHP >= 7.1, as it was fixed as part of Support for long and UTF-8 path (http://php.net/manual/en/migration71.windows-support.php). A workaround for PHP <= 7.0 is to use the `--prompt` option:

```
echo "Perícias Contábeis" | wp post create --post_type=page --post_status=publish --prompt=post_title
```

See also: #4714 (https://github.com/wp-cli/wp-cli/issues/4714)

## The installation hangs

If the installation seems to hang forever while trying to clone the resources from GitHub, please ensure that you are allowed to connect to Github using SSL (port 443) and Git (port 9418) for outbound connections.

## W3 Total Cache Error: some files appear to be missing or out of place.

W3 Total Cache object caching can cause this problem. Disabling object caching and optionally removing `wp-content/object-cache.php` will allow WP-CLI to work again.

See also: #587 (https://github.com/wp-cli/wp-cli/issues/587)

## The automated updater doesn't work for versions before 3.4

The `wp core update` command is designed to work for WordPress 3.4 and above. To be able to update an older website to latest WordPress, you could try one of the following alternatives:

1. **Fully-automated:** Run `wp core download --force` to download latest WordPress and replace it with your files (don't worry, `wp-config.php` will remain intact). Then, run `wp core update-db` to update the database. Since the procedure isn't ideal, run once again `wp core download --force` and the new version should be available.
2. **Semi-automated:** Run `wp core download --force` to download all files and replace them in your current installation, then navigate to `/wp-admin/` and run the database upgrade when prompted.

## PHP Fatal error: Maximum function nesting level of '#' reached, aborting!

You're encountering a limitation associated with PHP Xdebug (https://xdebug.org/). Here are a few solutions to the problem:

1. Temporarily turn off Xdebug while executing the CLI command: `XDEBUG_MODE=off wp <command> ...`
2. Increase the value of `xdebug.max_nesting_level` in your `php.ini` file. You can learn more about it here - `xdebug.max_nesting_level` (https://xdebug.org/docs/all_settings#max_nesting_level)

If you're looking for a quick fix while still retaining debugging tools, you can change the `max_nesting_level` value to execute the command just once:

```
php -d xdebug.max_nesting_level=512 wp <command> ...
```

If you're not in the process of debugging code, it's advisable to completely disable Xdebug. For instructions on how to permanently switch off Xdebug, please refer to the documentation of your local environment or your hosting provider.

You might experience similar issues with the alternate error message: `Error: Xdebug has detected a possible infinite loop, and aborted your script with a stack depth of '#' frames`

---

# Doctor Guides

Here are some helpful guides for using `wp doctor`:

- Default doctor diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-default-checks/)
- Customize doctor diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-customize-config/)
- Write a custom check to perform an arbitrary assertion (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-write-custom-check/)
- Write a check for verifying contents of WordPress files (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-file-contents/)
- Write a check for asserting the value of a given option (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-option-value/)
- Write a check for asserting the value of a given constant (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-constant-value/)
- Check status of a given plugin (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-plugin-status/)

---

# Commands Cookbook

Creating your own custom WP-CLI command can be easier than it looks — and you can use `wp scaffold package` (repo (https://github.com/wp-cli/scaffold-package-command)) 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 (https://developer.wordpress.org/cli/commands/plugin/install/)) is one such command, as is `wp plugin activate` (doc (https://developer.wordpress.org/cli/commands/plugin/activate/)). 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 (https://developer.wordpress.org/cli/commands/), while also offering a rich internal API (https://make.wordpress.org/cli/handbook/internal-api/) for third-parties to write and register their own commands.

WP-CLI commands can be distributed as standalone packages (https://wp-cli.org/package-index/), or bundled with WordPress plugins or themes. For the former, you can use `wp scaffold package` (repo (https://github.com/wp-cli/scaffold-package-command)) 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 install WordPress. There are exceptions to this rule though, notably `wp search-replace` (doc (https://developer.wordpress.org/cli/commands/search-replace/)).
- 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 (https://github.com/wp-cli/scaffold-package-command)).
- Can be distributed independent of a plugin or theme in the Package Index (https://wp-cli.org/package-index/).

All commands:

- Follow the documentation standards (https://make.wordpress.org/cli/handbook/documentation-standards/).

# Anatomy of a command

WP-CLI supports registering any callable class, function, or closure as a command. `WP_CLI::add_command()` (doc (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-command/)) 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 (https://make.wordpress.org/cli/handbook/config/) 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 );

// 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 (https://developer.wordpress.org/cli/commands/eval-file/)).

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.

*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.

The longdesc is also displayed when calling the `help` command, for example, `wp help example hello`. Its syntax is Markdown Extra (http://michelf.ca/projects/php-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.
- Sections 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 (https://make.wordpress.org/cli/handbook/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 (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-hook/).

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.

## 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*/
}
```

`$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
```

`$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 (https://github.com/wp-cli/wp-cli/issues/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 (https://make.wordpress.org/cli/handbook/internal-api/) this command uses to achieve its goal:

- `WP_CLI::add_command()` (doc (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-command/)) 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 (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-error/)) renders a nicely formatted error message and exits.
- `WP_CLI::launch_self()` (doc (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-launch-self/)) 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 (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-log/)) renders informational output to the end user.
- `WP_CLI\Utils\format_items()` (doc (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-utils-format-items/)) 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 (https://github.com/wp-cli/scaffold-package-command) to get started.

# Distribution

Now that you've produce 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';
}
```

## 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 recommended 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's 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
```

---

# Governance

This document describes the governance structure of the WP-CLI project.

## Who's in charge?

WP-CLI has one maintainer: schlessera (http://github.com/schlessera). On a day to day basis, final decision-making authority resides with him.

The project roadmap is an organic, evolving vision developed between the project's maintainers, committers, and contributors. Generally, we try to make decisions in alignment with the project's philosophy (https://make.wordpress.org/cli/handbook/philosophy/) and committers credo (https://make.wordpress.org/cli/handbook/committers-credo/).

## Why are some people members of the WP-CLI organization?

On occasion, we grant write access to contributors (https://github.com/orgs/wp-cli/teams/committers) who have demonstrated, over a period of time, that they are capable and invested in moving the project forward. These committers are expected to:

- Hold product and code quality in the highest regard.
- Exhibit stellar judgement and communication.
- Participate in the project on a consistent basis.

Becoming a committer can seem glamorous, but it also comes with expectations of responsibility, commitment, and humility.

andreascreten (https://github.com/andreascreten) is the original author of WP-CLI. scribu (https://github.com/scribu) and danielbachhuber (https://github.com/danielbachhuber) have been long time maintainers. schlessera (https://github.com/schlessera) is the current maintainer.

The maintainer is the person most directly responsible for the reliability and longevity of the project. They are expected to:

- Hold WP-CLI's overall quality in the utmost regard.
- Refine the project's vision and direction, and make sure it's understood by all.
- Enable everyone from new users to experienced committers.
- Take care of the boring day-to-day work (keeping CI passing, triaging new issues, etc.).

# What's the connection between WP-CLI and WordPress?

As of December 2017, WP-CLI is a formal WordPress project (https://make.wordpress.org/core/2016/12/28/supporting-the-future-of-wp-cli/).

Practically speaking, this means we get to take advantage of the WordPress project's people and software infrastructure:

- Much of the WP-CLI documentation is hosted on the wordpress.org domain.
- We get to use the `#cli` channel in the WordPress Slack organization (https://make.wordpress.org/chat/).
- If we need help with something, we can ask.
- Two core committers (dd32 (https://github.com/dd32) and pento (https://github.com/pento)) have ownership permissions to the WP-CLI GitHub organization, in the unlikely event everyone else gets hit by a bus.

Other than this, WP-CLI operates independently.

# How does all of this work?

The WP-CLI GitHub organization (https://github.com/wp-cli) contains all project code repositories. Some commands are bundled with WP-CLI, while others are independently installable. Commands are bundled with WP-CLI when they're stable, closely aligned with the project philosophy (https://make.wordpress.org/cli/handbook/philosophy/), and useful to a majority of users.

wp-cli/wp-cli (https://github.com/wp-cli/wp-cli) is the main project repository, which pulls in command packages and other dependencies through Composer. Composer defines which version of which dependencies is included in the build. Changes to command packages are included in WP-CLI proper when a stable release is tagged for the package.

Nightly Phar builds are created by a Travis job that calls deploy.sh (https://github.com/wp-cli/wp-cli/blob/master/ci/deploy.sh) and pushes the build artifact to the builds repository (http://github.com/wp-cli/builds). Releases are prepared manually in accordance to the release checklist (https://make.wordpress.org/cli/handbook/release-checklist/).

The `wp-cli.org` domain is currently owned by andreascreten (https://github.com/andreascreten). DNS is managed through a Cloudflare (https://www.cloudflare.com/) account that danielbachhuber (https://github.com/danielbachhuber) holds credentials to.

Much of the WP-CLI documentation (command pages, etc.) is editable through wp-cli/handbook (https://github.com/wp-cli/handbook/), and then synced to WordPress.org. The WP-CLI.org homepage is hosted on GitHub Pages (http://github.com/wp-cli/wp-cli.github.com).

From time to time, you may see a pull request from the `wp-make-coffee` bot. These originate from a donated WebFaction server running some cron jobs calling bash scripts:

```
5 4 * * 1,3,5 source ~/.bash_profile; WP_CLI_DIR=~/wp-cli bash ~/wp-cli/utils/auto-composer-update.sh > ~/auto-composer-update.log 2>
5 6 * * 1,3,5 source ~/.bash_profile; WP_CLI_DIR=~/wp-cli-bundle bash ~/wp-cli-bundle/utils/auto-composer-update.sh > ~/auto-composer
```

The wpcli (https://twitter.com/wpcli) Twitter account is managed by schlessera (https://github.com/schlessera).

If you subscribe for email updates (https://make.wordpress.org/cli/subscribe/), your email address is registered with WordPress.com through Automattic's Jetpack plugin.

---

# How to start the webserver

You can use the command `wp server` to launches PHP's built-in web server for a specific WordPress installation. By default, the webserver will start using the localhost and default port 8080 but, you can change them using the `--host` and `--port` options.

## Step 1 - Start the web server using the default settings

```
$ wp server
PHP 7.2.24-0ubuntu0.18.04.4 Development Server started at Thu Jun  4 17:40:13 2020
Listening on http://localhost:8080
Document root is ../wpdemo.test
Press Ctrl-C to quit.
```

The command above starts the server using the default settings. We can now open our browser and visit the link http://localhost:8080 to access our WordPress installation.

## Step 2 - Start the web server using different settings

If you want to specify a different port number or host, you can pass them to the following flags: `--port` and `--host`

```
$ wp server --port=9090 --host=192.168.0.4
```

The command above will start the webserver and listen for requests on http://192.168.0.4:9090 (http://192.168.0.4:9090) you can open the browser and visit the page to access your WordPress installation.

---

# Quick Start

Congratulations! You've installed WP-CLI (https://make.wordpress.org/cli/handbook/installing/) for the first time, and are ready to level-up your use of WordPress. This page contains a brief introduction to WP-CLI with some example usage.

# Introduction

WP-CLI is a command line interface for WordPress. The project'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.

For instance, because you can install a plugin from the WordPress admin, you can also install a plugin (https://developer.wordpress.org/cli/commands/plugin/install/) with WP-CLI:

```
$ wp plugin install akismet
Installing Akismet (3.1.8)
Downloading install package from https://downloads.wordpress.org/plugin/akismet.3.1.8.zip...
Unpacking the package...
Installing the plugin...
Plugin installed successfully.
```

And, because you can also activate plugins from the WordPress admin, you can activate a plugin (https://developer.wordpress.org/cli/commands/plugin/activate/) with WP-CLI:

```
$ wp plugin activate akismet
Success: Plugin 'akismet' activated.
```

One key difference between using the WordPress admin and WP-CLI: performing any action takes many fewer clicks. As you become more familiar with the command line, you'll notice performing a given task with WP-CLI is generally much faster than performing the same task through the WordPress admin. Investing time upfront into learning how to better use WP-CLI pays dividends in the long term.

# Common Terms

Throughout your usage of WP-CLI, you'll hear certain terms used over and over again.

For instance, a *command* is an atomic unit of WP-CLI functionality. `wp plugin install` is one such command, as is `wp plugin activate`. Commands represent a name (e.g. 'plugin install') and a callback, and are registered with `WP_CLI::add_command()` (doc (https://make.wordpress.org/cli/handbook/internal-api/wp-cli-add-command/)).

The *synopsis* 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 (https://make.wordpress.org/cli/handbook/config/) 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.

# Practical Examples

Ready to dive in? Here are some common examples of how WP-CLI is used:

**Download and install WordPress in seconds**

1. Download the latest version of WordPress with `wp core download` (doc (https://developer.wordpress.org/cli/commands/core/download/)).

```
$ wp core download --path=wpclidemo.dev
Creating directory '/srv/www/wpclidemo.dev/'.
Downloading WordPress 4.6.1 (en_US)...
Using cached file '/home/vagrant/.wp-cli/cache/core/wordpress-4.6.1-en_US.tar.gz'...
Success: WordPress downloaded.
```

2. Create a new wp-config.php file with `wp config create` (doc (https://developer.wordpress.org/cli/commands/config/create/)).

```
$ cd wpclidemo.dev
$ wp config create --dbname=wpclidemo --dbuser=root --prompt=dbpass
1/10 [--dbpass=<dbpass>]:
Success: Generated 'wp-config.php' file.
```

3. Create the database based on wp-config.php with `wp db create` (doc (https://developer.wordpress.org/cli/commands/db/create/)).

```
$ wp db create
Success: Database created.
```

4. Install WordPress with `wp core install` (doc (https://developer.wordpress.org/cli/commands/core/install/)).

```
$ wp core install --url=wpclidemo.dev --title="WP-CLI" --admin_user=wpcli --admin_password=wpcli --admin_email=info@wp-cli.org
Success: WordPress installed successfully.
```

That's it!

**Update plugins to their latest version**

Use `wp plugin update --all` (doc (https://developer.wordpress.org/cli/commands/plugin/update/)) to update all plugins to their latest version.

```
$ wp plugin update --all
Enabling Maintenance mode...
Downloading update from https://downloads.wordpress.org/plugin/akismet.3.1.11.zip...
Unpacking the update...
Installing the latest version...
Removing the old version of the plugin...
Plugin updated successfully.
Downloading update from https://downloads.wordpress.org/plugin/nginx-champuru.3.2.0.zip...
Unpacking the update...
Installing the latest version...
Removing the old version of the plugin...
Plugin updated successfully.
Disabling Maintenance mode...
Success: Updated 2/2 plugins.
+------------------------+-------------+-------------+---------+
| name                   | old_version | new_version | status  |
+------------------------+-------------+-------------+---------+
| akismet                | 3.1.3       | 3.1.11      | Updated |
| nginx-cache-controller | 3.1.1       | 3.2.0       | Updated |
+------------------------+-------------+-------------+---------+
```

**Add a user as a super-admin**

On multisite, use `wp super-admin add` (doc (https://developer.wordpress.org/cli/commands/super-admin/add/)) to grant super admin capabilities to an existing user.

```
$ wp super-admin add wpcli
Success: Granted super-admin capabilities.
```

**Regenerate thumbnails**

If you've added or changed an image size registered with `add_image_size()`, you may want to use `wp media regenerate` (doc (https://developer.wordpress.org/cli/commands/media/regenerate/)) so your theme displays the correct image size.

```
wp media regenerate --yes
Found 1 image to regenerate.
1/1 Regenerated thumbnails for "charlie-gpa" (ID 4).
Success: Finished regenerating the image.
```

**Search and replace URLs**

If you're moving a database from one domain to another, `wp search-replace` (doc (https://developer.wordpress.org/cli/commands/search-replace/)) makes it easy to update all URL references in your database.

To see which links will be replaced in your database, use the --dry-run flag:

```
wp search-replace 'http://oldsite.com' 'http://newsite.com' --dry-run
```

When ready to replace the links, run:

```
wp search-replace 'http://oldsite.com' 'http://newsite.com'
```

Wondering what's next? Browse through all of WP-CLI's commands (https://developer.wordpress.org/cli/commands/) to explore your new world. For more detailed information about creating custom commands, visit the WP-CLI Commands Cookbook (https://make.wordpress.org/cli/handbook/guides/commands-cookbook/). Or, catch up with shell friends (https://make.wordpress.org/cli/handbook/shell-friends/) to learn about helpful command line utilities.

---

# Committers Credo

Some people have write access to WP-CLI repositories. These people are identified in the following teams:

- maintainers (https://github.com/orgs/wp-cli/teams/maintainers) - Project leadership
- committers (https://github.com/orgs/wp-cli/teams/committers) - Trusted contributors

This “Committers credo” is a living document. It’s meant to establish generally agreed upon standards for committers, and will continue to evolve over time.

## Product quality

WP-CLI is a project depended upon by the entire WordPress ecosystem. This trust is earned by producing a consistent, dependable, and high quality product. WP-CLI committers hold product and code quality in the highest regard.

Practically-speaking:

- Given a choice between Good, Fast, and Cheap, we pick Good and Slow. Creating great software is a journey, not a destination.
- Product decisions are made in alignment with our core philosophy (https://make.wordpress.org/cli/handbook/philosophy/).
- Code review is an integral part of our development workflow (https://make.wordpress.org/cli/handbook/code-review/). Pull requests need approval of at least one other committer before merging. If you’d like more than one code review, please don’t hesitate to request one. You can request a review from a specific person, or from the `@wp-cli/committers` team.
- If you need help with something, ask for help. Seriously, ask for help any time you feel like you need help. No sense getting stuck.

## Stellar judgement

A great product is a reflection of thoughtful, deliberate, and considered decision-making. WP-CLI committers exhibit stellar judgement with making decisions on new features, fixing bugs, merging others pull requests, and generally working on the project.

The basis of this decision-making ensures:

- New features fit within the scope and philosophy of the WP-CLI project (https://make.wordpress.org/cli/handbook/philosophy/).
- Bug fixes are prioritized, because bugs negatively affect the product quality.
- Pull requests include tests covering the scope of the change.
- Potential code additions are evaluated on their maintenance burden as well as their usefulness.

## Consistent participation

Effective involvement with the WP-CLI project requires consistent involvement, clarity of communication, and a propensity to help others.

As a widely adopted project, WP-CLI has a diverse user base. Helping this user base has tremendous opportunity for impact. WP-CLI committers prioritize clearing others’ blockers over their own work, knowing this is a key contributor to a stable community.

Notably, the idea that "pending code reviews represent blocked threads of execution (http://glen.nu/ramblings/oncodereview.php)" can be applied broadly across all forms of collaboration. Prioritize providing feedback to others over your own work, if your feedback is blocking their progress. Request a code review (https://make.wordpress.org/cli/handbook/code-review/) for a pull request by assigning the `@wp-cli/committers` team for review.

Committer participation isn't just committing code. Often, you can have a huge amount of impact by investing a little bit of your time into:

- Refining an existing piece of documentation, or drafting a new one.
- Helping a contributor with whatever is needed to finish up their pull request.
    - If a contributor abandons a pull request (e.g. no activity in two weeks) that's close to a mergeable state, you can perform any necessary cleanup by committing directly to their branch (https://help.github.com/articles/committing-changes-to-a-pull-request-branch-created-from-a-fork/) and get it over the finish line.
- Triaging issues by spending 5-10 minutes further diagnosing the report and commenting with additional information you've discovered.

---

# Installing

## Recommended installation

The recommended way to install WP-CLI is by downloading the Phar build (archives similar to Java JAR files, see this article for more detail (http://php.net/manual/en/phar.using.intro.php)), marking it executable, and placing it on your PATH.

First, download wp-cli.phar (https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar) using `wget` or `curl`. For example:

```
curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
```

Then, check if it works:

```
php wp-cli.phar --info
```

To be able to type just `wp`, instead of `php wp-cli.phar`, you need to make the file executable and move it to somewhere in your PATH. For example:

```
chmod +x wp-cli.phar
sudo mv wp-cli.phar /usr/local/bin/wp
```

Now try running `wp --info`. If WP-CLI is installed successfully, you'll see output like this:

```
OS:  Linux 4.10.0-42-generic #46~16.04.1-Ubuntu SMP Mon Dec 4 15:57:59 UTC 2017 x86_64
Shell:   /usr/bin/zsh
PHP binary:    /usr/bin/php
PHP version:     7.1.12-1+ubuntu16.04.1+deb.sury.org+1
php.ini used:   /etc/php/7.1/cli/php.ini
MySQL binary:
MySQL version:
SQL modes:
WP-CLI root dir:        /home/wp-cli/.wp-cli
WP-CLI packages dir:    /home/wp-cli/.wp-cli/packages/
WP-CLI global config:   /home/wp-cli/.wp-cli/config.yml
WP-CLI project config:
WP-CLI version: 1.5.0
```

Voila! You're now an official WP-CLI user.

Wondering what to do next? Check out the quick start guide (https://make.wordpress.org/cli/handbook/quick-start/) for a short introduction and some example usage.

## Updating WP-CLI

If you have installed WP-CLI using the recommended Phar method, you can update it at any time by running `wp cli update` (although if WP-CLI is owned by root, that may be `sudo wp cli update`). If you installed WP-CLI using the Composer or Git-based installations, see the specific instructions for updating associated with each method below.

When you run `wp cli update`, you'll be prompted to confirm that you wish to update with a message similar to the following:

```
You have version 0.21.1. Would you like to update to 0.23.1? [y/n]
```

After you accept, you should see a success message:

```
Success: Updated WP-CLI to 0.23.1
```

If you're already running the latest version of WP-CLI, you'll see this message:

```
WP-CLI is at the latest version.
```

Want to live life on the edge? Run `wp cli update --nightly` to use the latest nightly build of WP-CLI. The nightly build is more or less stable enough for you to use in your local environment, and always includes the latest and greatest.

For more information about `wp cli update`, including flags and options that can be used, read the full docs page on the update command (https://wp-cli.org/commands/cli/update/).

## Tab completions

### Bash & Z-Shell

WP-CLI also comes with a tab completion script for *Bash* and *Z-Shell*. Just download wp-completion.bash (https://github.com/wp-cli/wp-cli/raw/main/utils/wp-completion.bash) and source it from `~/.bash_profile`:

```
source /FULL/PATH/TO/wp-completion.bash
```

To have this change take effect in your currently active shell, run `source ~/.bash_profile` afterwards.

### Oh My Zsh

If you're using the *Oh My Zsh* framework, you can enable the built-in `wp-cli` plugin (https://github.com/robbyrussell/oh-my-zsh/tree/master/plugins/wp-cli), by adding it to the `plugins=(wp-cli git [...])` line in your `~/.zshrc` file.

Note: the Oh My Zsh plugin comes with the bash completion script included, so it's unnecessary to have both.

To have this change take effect in your currently active shell, run `source ~/.zshrc` afterwards.

### Fish

If you're using the *Fish* shell, you can download wp.fish (https://github.com/wp-cli/wp-cli/raw/main/utils/wp.fish) and move it to `~/.config/fish/completions/wp.fish`. Afterwards just type `wp` and press `TAB`, and fish will automatically source `wp.fish`.

## Dash/Alfred workflow

If you're using *Dash* and *Alfred*, you can add a custom Alfred workflow to look up WP-CLI command information.

### Setup

Open Dash and download the WP-CLI docset (https://github.com/wp-cli/dash-docset-generator): Dash › Preferences › Downloads › User Contributions › Search for WP-CLI

Still in Dash, activate the Alfred integration: Dash › Preferences › Integration › Alfred

### Usage

Open Alfred and try searching for a specific command. For example, this gives you an overview of the `plugin` command and its subcommands: `wp-cli plugin`

# Alternative installation methods

## Installing via Git

If you intend to work on WP-CLI itself, see the Setting up (https://make.wordpress.org/cli/handbook/pull-requests/#setting-up) section in Pull Requests (https://make.wordpress.org/cli/handbook/pull-requests/).

## Installing nightly via Phar

The "nightly" is the bleeding-edge version of WP-CLI, built straight from the main branch (https://github.com/wp-cli/wp-cli/commits/main).

Just follow the normal installation instructions (/#install), except change the URL to the phar file:

https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli-nightly.phar (https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli-nightly.phar)

## Installing via Composer

**As part of a project**

Add the following line to your project's `composer.json` file:

```
"require" : {
        "wp-cli/wp-cli-bundle": "*"
}
```

To add any additional suggested packages seen in the `wp-cli-bundle` package, such as `psy/psysh`, run:

```
composer require --dev $(composer suggests --by-package | awk '/wp-cli\/wp-cli-bundle/' RS= | grep -o -P '(?<=- ).*(?=:)')
```

Optionally (if run on a server or for e.g. in a virtual machine locally) you can automate setting up the command and making it available in the users path. Let's assume *Composer* installed into `/var/www/vendor` (you can get the composer `vendor-dir` config variable specific to your machine via `composer config --list | grep "\[vendor-dir\]"`), we can add the following `scripts`/commands to the `composer.json` file. The second and third line set up bash completion (https://github.com/wp-cli/wp-cli/blob/main/utils/wp-completion.bash) so we don't have to remember every single command:

```
"scripts" : {
        "post-install-cmd" : [
                "[[ -f /usr/bin/wp ]] || sudo ln -s /var/www/vendor/wp-cli/wp-cli/bin/wp /usr/bin/wp",
                "source /var/www/vendor/wp-cli/wp-cli/utils/wp-completion.bash",
    "[[ -f ~/.bash_profile ]] || touch ~/.bash_profile",
                "source ~/.bash_profile"
        ]
}
```

Above script assumes that your current shell is `bash`, which might not be the case for all users. Example for a vagrant box (added to the `scripts`-block):

```
cat /etc/passwd | grep $(whoami)
vagrant:x:1000:1000::/home/vagrant:/bin/bash
```

In case you got `bash` available and installed for your OS, you can switch dynamically:

```
"scripts" : {
        "post-update-cmd" : [
                "/bin/bash -c \"[[ -f /usr/local/bin/wp ]] || sudo ln -s /var/www/vendor/wp-cli/wp-cli/bin/wp /usr/bin/wp\"",
                "/bin/bash -c \"source /var/www/vendor/wp-cli/wp-cli/utils/wp-completion.bash\"",
    "/bin/bash -c \"[[ -f ~/.bash_profile ]] || touch ~/.bash_profile\"",
                "/bin/bash -c \"source ~/.bash_profile\""
        ]
}
```

**As a project**

Needs `php` and `composer` (or `php composer.phar`) set up as console commands.

```
composer create-project wp-cli/wp-cli-bundle --prefer-source
```

Then run `wp-cli-bundle/vendor/wp-cli/wp-cli/bin/wp` or add `wp-cli-bundle/vendor/wp-cli/wp-cli/bin` folder to `PATH` for global `wp` command (on Windows, use `wp-cli/bin/wp.bat` instead).

To update, you'll need to:

```
cd wp-cli-bundle
git pull origin main
composer install
```

**Global require**

If you prefer to have PHP tools installed globally via Composer and have something like `~/.composer/vendor/bin` in your PATH (or `C:\Users\you\AppData\Roaming\Composer\vendor\bin` on Windows), you can just run:

```
composer global require wp-cli/wp-cli-bundle
```

To update everything globally, run `composer global update`.

#### Installing a specific version

If you want to install a specific version of WP-CLI then append the version numbers behind the packages

```
composer create-project wp-cli/wp-cli-bundle:2.1.0 --no-dev
```

The version must be in a format (https://getcomposer.org/doc/04-schema.md#version) that Composer can understand and can be found on packagist.org (https://packagist.org/packages/wp-cli/wp-cli).

#### Installing bleeding-edge

If you want to install bleeding-edge then use `dev-main`:

```
composer create-project wp-cli/wp-cli-bundle:dev-main --no-dev
```

### Installing globally as a project

You can specify a custom install path for WP-CLI, like so:

```
composer create-project wp-cli/wp-cli-bundle /usr/share/wp-cli --no-dev
```

Then, just symlink the binary:

```
sudo ln -s /usr/share/wp-cli-bundle/vendor/wp-cli/wp-cli/bin /usr/bin/wp
```

## Installing via Homebrew

```
brew install wp-cli
```

Here's the formula (https://github.com/Homebrew/homebrew-core/blob/master/Formula/w/wp-cli.rb).

## Installing via Docker

The Docker community maintains WordPress and WP-CLI images (https://hub.docker.com/_/wordpress/).

To include the WP-CLI image in your own project:

```
image: wordpress:cli
```

## Installing on Windows

Install via composer as described above or use the following method.

Make sure you have php installed and in your path (http://php.net/manual/en/faq.installation.php#faq.installation.addtopath) so you can execute it globally.

Download wp-cli.phar (https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar) manually and save it to a folder, for example `c:\wp-cli`

Create a file named `wp.bat` in `c:\wp-cli` with the following contents:

```
@ECHO OFF
php "c:/wp-cli/wp-cli.phar" %*
```

Add `c:\wp-cli` to your path:

```
setx path "%path%;c:\wp-cli"
```

You can now use WP-CLI from anywhere in Windows command line.

## Installing via .deb package

On Debian or Ubuntu, just download and open one of the .deb packages: https://github.com/wp-cli/builds/tree/gh-pages/deb (https://github.com/wp-cli/builds/tree/gh-pages/deb)

## Installing on Fedora 30+

```
su -c 'dnf install wp-cli'
```

## Installing on CentOS

```
su -c 'yum install wp-cli'
```

## Using a custom PHP binary

In some cases, like for MAMP installs, you might not want to use the default PHP binary.

To use the latest PHP version provided by MAMP, you'll need to modify your `PATH` environment variable with the following added to your `~/.bash_profile` or `~/.zsh_profile`:

```
PHP_VERSION=$(ls /Applications/MAMP/bin/php/ | sort -n | tail -1)
export PATH=/Applications/MAMP/bin/php/${PHP_VERSION}/bin:$PATH
```

To use a specific PHP version provided by MAMP, you'll need to determine the path to the PHP version's executable, and modify your `PATH` environment variable with the following added to your `~/.bash_profile` or `~/.zsh_profile`:

```
export PATH=/Applications/MAMP/bin/php/php5.5.26/bin:$PATH
```

Note there's no assignment of the `PHP_VERSION` variable in this case, because we aren't dynamically looking up the latest PHP version.

Once you have added that and saved the file, reload the file with:

```
source ~/.bash_profile
```

After you've done that, run `wp --info` to make sure the change has been applied correctly.

For Composer and Git-based WP-CLI installation, you can alternatively set the `WP_CLI_PHP` environment variable, if you don't want to modify `PATH` for some reason.

## Installing on MediaTemple

See http://razorfrog.com/installing-wp-cli-on-mediatemple-grid-server/ (http://razorfrog.com/installing-wp-cli-on-mediatemple-grid-server/)

---

# Plugin Integration Tests

This guide demonstrates how to run integration tests on both Travis CI and locally. The scaffolding uses the WordPress Core "unit tests" that serve to unit-test WordPress Core. Unit tests are useful for testing plugins and themes as well, but if they are used in that way, it turns those tests into "integration tests" - they test the integration between a plugin/theme and WordPress Core. Proper unit tests for a plugin or theme would not load WordPress.

# Running tests on Travis CI

If you host your plugin on GitHub and enable Travis CI (https://docs.travis-ci.com/), the tests will be run automatically after every commit you make to the plugin.

All you need to do to enable this is copy and then edit the following files from the WP-CLI sample plugin (https://github.com/wp-cli/sample-plugin):

- `.travis.yml`, `phpunit.xml.dist` and `phpcs.ruleset.xml` files
- `tests` folder

See the docs for an explanation of what each file does (https://developer.wordpress.org/cli/commands/scaffold/plugin-tests/). You will then need to specify your unit tests in the `tests/` folder.

# Running tests locally

Running tests locally can be beneficial during development as it is quicker than committing changes and waiting for Travis CI to run the tests.

We're going to assume that:

- You already have a plugin called `my-plugin`
- You have installed `git`, `svn`, `php`, `apache` (How To Install Linux, Apache, MySQL, PHP (LAMP) stack on Ubuntu 16.04 (https://www.digitalocean.com/community/tutorials/how-to-install-linux-apache-mysql-php-lamp-stack-on-ubuntu-16-04))

So, let's get started:

## 1. Install PHPUnit (https://github.com/sebastianbergmann/phpunit#installation)

The version of PHPUnit to install depends on both PHP and WordPress versions. See: PHPUnit Compatibility and WordPress Versions – Make WordPress Core (https://make.wordpress.org/core/handbook/references/phpunit-compatibility-and-wordpress-versions/)

## 2. Generate the plugin test files

```
wp scaffold plugin-tests my-plugin
```

This command will generate all the files needed for running tests, including a `.travis.yml` file.

## 3. Initialize the testing environment locally

`cd` into the plugin directory and run the install script (you will need to have `wget` installed).

```
bash bin/install-wp-tests.sh wordpress_test root '' localhost latest
```

The install script first installs a copy of WordPress in the `/tmp` directory (by default) as well as the WordPress unit testing tools. Then it creates a database to be used while running tests. The parameters that are passed to `install-wp-tests.sh` setup the test database.

- `wordpress_test` is the name of the test database (**all data will be deleted!**)
- `root` is the MySQL user name
- `''` is the MySQL user password
- `localhost` is the MySQL server host
- `latest` is the WordPress version; could also be `3.7`, `3.6.2` etc.

NOTE: This script can be run multiple times without errors, but it will *not* overwrite previously existing files. So if your DB credentials change, or you want to switch to a different instance of mysql, simply re-running the script won't be enough. You'll need to manually edit the `wp-config.php` that's installed in `/tmp`.

## 4. Run the plugin tests

```
phpunit
```

If you have trouble running the install script or PHPUnit, check Support section (http://wp-cli.org/#support) for help and answers to common issues.

# Integrating WP Unit Testing in Windows

Tried and gotten stuck setting up unit tests for your project in Windows?

First, know that some WP-CLI commands don't work in Windows, so you'll want to use Cygwin. Cygwin is also preferred by the WordPress core project (https://make.wordpress.org/core/handbook/tutorials/working-with-patches/).

Second, the `bin/install-wp-tests.sh` script doesn't work directly in Windows. Windows 10 introduced a Windows Subsystem for Linux (http://www.pcworld.com/article/3106463/windows/how-to-get-bash-on-windows-10-with-the-anniversary-update.html) but older versions require extra effort. Cygwin is preferred because it runs bash scripts by default.

Third, the bash installation script uses `which`, a Linux command not available by default in Windows. `cURL`, `svn` and `wget` all can be installed in Windows as separate packages.

Lastly, sometimes the bash script fails to build WordPress because of how it uses tmp folders. If it fails, then you can manually install WordPress to a writeable directory, and then manually create `wp-tests-config.php`.

---

# Documentation Standards

To promote similarity and consistency between WP-CLI commands, we've produced these documentation standards we'd encourage you to follow. Commonality is a key component of usability, because it reduces the mental overhead required when switching between commands.

Pull requests against the WP-CLI project are reviewed for these standards. Please follow these standards for your custom commands as well.

## Command annotation

Here's an example of the PHPdoc annotating the `wp cron event schedule` command:

```
/**
 * Schedule a new cron event.
 *
 * ## OPTIONS
 *
 * <hook>
 * : The hook name.
 *
 * [<next-run>]
 * : A Unix timestamp or an English textual datetime description compatible with `strtotime()`. Defaults to now.
 *
 * [<recurrence>]
 * : How often the event should recur. See `wp cron schedule list` for available schedule names. Defaults to no recurrence.
 *
 * [--<field>=<value>]
 * : Associative args for the event.
 *
 * ## EXAMPLES
 *
 *     # Schedule a new cron event.
 *     $ wp cron event schedule cron_test
 *     Success: Scheduled event with hook 'cron_test' for 2016-05-31 10:19:16 GMT.
 *
 *     # Schedule new cron event with hourly recurrence.
 *     $ wp cron event schedule cron_test now hourly
 *     Success: Scheduled event with hook 'cron_test' for 2016-05-31 10:20:32 GMT.
 *
 *     # Schedule new cron event and pass associative arguments.
 *     $ wp cron event schedule cron_test '+1 hour' --foo=1 --bar=2
 *     Success: Scheduled event with hook 'cron_test' for 2016-05-31 11:21:35 GMT.
 */
```

To break the example down piece by piece:

- "Schedule a new cron event." is the command description. The command description should be under 50 characters and written in an active, present tense.
- The options section should begin with `## OPTIONS`. Keep a blank line before and after the heading.
- Each option should be named in a manner that concisely describes its purpose.
- The example section should start with `## EXAMPLES`. Keep a blank line before and after the heading.
- Include a blank line before and after each example to help visually indicate each as a separate example.
- Each example should have 3 parts.
    - Description
        - Must start with `#` and a space.
        - Sentence should start with a capital letter.
        - Comment must end in full-stops, exclamation marks, or question marks.
        - Eg: `# Create database.`
    - Command
        - Must start with `$` and a space. Eg: `$ wp db create`
    - Sample Output
        - Keep exact output. Note spaces and indentation in the output. Exception: If output is very long, it could be truncated to show suitable section only.
        - Eg: `Success: Database created.`
- If possible, keep at least two examples for each command. One showing basic use and another advanced use. More use cases the better.

For more details on how WP-CLI understands the PHPdoc, please see the commands cookbook (https://make.wordpress.org/cli/handbook/commands-cookbook/).

# Class annotation

For classes representing a collection of subcommands, we'd encourage you to use class-level annotation to provide an introduction to the collection.

```
/**
 * Manage options.
 *
 * ## EXAMPLES
 *
 *     # Get site URL.
 *     $ wp option get siteurl
 *     http://example.com
 *
 *     # Add option.
 *     $ wp option add my_option foobar
 *     Success: Added 'my_option' option.
 *
 *     # Update option.
 *     $ wp option update my_option '{"foo": "bar"}' --format=json
 *     Success: Updated 'my_option' option.
 *
 *     # Delete option.
 *     $ wp option delete my_option
 *     Success: Deleted 'my_option' option.
 */
```

- Class could have multiple examples. Please follow standards for examples on a command doc.

# Messages within command execution

```
$ wp theme activate twentysixteen
Success: Switched to 'Twenty Sixteen' theme.
```

- Message must start with a capital letter.
  - Exception: When message starts with a special key and is wrapped with quotes. Eg - `'movie' is not a registered post type.`
- If single line message, it must end with `.`.
  - Exception: There should be no trailing character in the end when colon `:` is used like this. `Invalid ID: 123`
  - Exception: Message to display in progress bar can omit trailing period. Eg - `Generating comments`
- Filenames and folder names must be wrapped with quotes ( `'` ).
- Roles, sidebar ID, post type key, taxonomy key must be wrapped with quotes.
- Message in the context of ongoing action could end with `...`. Eg - `Downloading from https://github.com/wp-cli/wp-cli/releases/download/v0.23.1/wp-cli-0.23.1.phar...`

# Command parameter description

If there is a set of allowed values for a command parameter, we have set special format (mostly similar to YAML) for default value and available options.

```
[--format=<format>]
: Render output in a particular format.
---
default: table
options:
  - table
  - csv
  - json
  - count
  - yaml
---
```

- Section starts with `---` in the separate line next to the param description.
- In the next line keep default value with `default: value`.
- For options, start with `options:` in the following line.
- Keep value in the following pattern: 2 space, `-`, 1 space, value
- Close section with `---`.

---

# wp-cli/handbook

These files comprise the WP-CLI handbook (make.wordpress.org/cli/handbook (https://make.wordpress.org/cli/handbook/)) and WP-CLI commands directory (developer.wordpress.org/cli/commands (https://developer.wordpress.org/cli/commands/)).

The documentation is located in GitHub to enable a pull request-based editing workflow.

Long-form documentation (e.g. "Commands cookbook") can be edited directly.

Internal API docs and command pages are generated dynamically from the WP-CLI codebase using the `wp handbook` series of commands.

Before running these commands the bash script `bin/install-packages.sh` should be run to install the latest versions of the non-bundled commands in `bin/packages`. Note `wp` must point to the target WP-CLI instance, i.e. the phar/git that contains the docblocks to be generated against, and should be run with `WP_CLI_PACKAGES_DIR=bin/packages` and `WP_CLI_CONFIG_PATH=/dev/null`.

So for instance to generate all dynamically created documentation against the nightly phar run:

```
wp cli update --nightly
bin/install_packages.sh
WP_CLI_PACKAGES_DIR=bin/packages WP_CLI_CONFIG_PATH=/dev/null wp handbook gen-all
```

All documentation is imported automatically into WordPress.org in a two step process:

1. WordPress reads `commands-manifest.json` or `handbook-manifest.json` to understand all pages that need to be created.
2. Each WordPress page has a `markdown_source` attribute specifying a Markdown file to be fetched, converted to HTML, and saved in the database.

For make.wordpress.org/cli, the import process is a WordPress plugin (https://meta.trac.wordpress.org/browser/sites/trunk/wordpress.org/public_html/wp-content/plugins/wporg-cli/inc/class-markdown-import.php) running a WP Cron job every 15 minutes. For developer.wordpress.org/cli, this is a class in the devhub theme (https://meta.trac.wordpress.org/browser/sites/trunk/wordpress.org/public_html/wp-content/themes/pub/wporg-developer/inc/cli.php) running a WP Cron job every 12 hours.

---

# Config

WP-CLI has a series of global parameters (e.g. `--path=<path>` and `--user=<user>`) which work with all commands. They are called *global parameters* because they affect how WP-CLI interacts with WordPress, and have the same behavior across all commands.

```
# `--user=<user>` sets request to a specific WordPress user
$ wp --user=wpcli eval 'echo wp_get_current_user()->user_email;'
wpcli@example.com
```

For repeated usage, WP-CLI can also read options from a YAML configuration file (e.g. `wp-cli.yml`). WP-CLI automatically discovers configuration files on the filesystem based on rules defined below. These configuration files enable specifying default values for both global parameters and subcommand-specific arguments.

```
# WordPress develop includes a `wp-cli.yml` to enable easy use of WP-CLI
$ pwd
/srv/www/wordpress-develop.dev
$ cat wp-cli.yml
path: src/
```

Arguments are interpreted following an order of precedence, from highest priority to lowest:

1. Command-line arguments.
2. `wp-cli.local.yml` file inside the current working directory (or upwards).
3. `wp-cli.yml` file inside the current working directory (or upwards).
4. `~/.wp-cli/config.yml` file (path can be changed by setting the `WP_CLI_CONFIG_PATH` environment variable).
5. WP-CLI defaults.

## Global parameters

The table below lists the available arguments (specified on the command-line) and options (specified in the configuration file).

| Description | Argument | Option |
|---|---|---|
| Path to the WordPress files.<br>Default value: `null` | `--path=<path>` | `path: <path>` |

| Description | Argument | Option |
|---|---|---|
| Perform operation against a remote server over SSH.<br>Default value: `null` | `--ssh=[<user>@]<host>[:<port>][<path>]` | `ssh: [<user>@]<host>[:<port>][<path>]` |
| Perform operation against a remote WordPress install over HTTP.<br>Default value: `null` | `--http=<http>` | `http: <http>` |
| Pretend request came from given URL. In multisite, this argument is how the target site is specified.<br>Default value: `null` | `--url=<url>` | `url: <url>` |
| Set the WordPress user.<br>Default value: `null` | `--user=<id\|login\|email>` | `user: <id\|login\|email>` |
| Skip loading all or some plugins. Note: mu-plugins are still loaded.<br>Default value: `""` | `--skip-plugins[=<plugin>]` | `skip-plugins: <list>` |
| Skip loading all or some themes.<br>Default value: `""` | `--skip-themes[=<theme>]` | `skip-themes: <list>` |
| Skip loading all installed packages.<br>Default value: `false` | `--skip-packages` | `skip-packages: <bool>` |
| Load PHP file before running the command (may be used more than once).<br>Default value: `[]` | `--require=<path>` | `require: <path>` |
| Execute PHP code before running the command (may be used more than once).<br>Default value: `[]` | `--exec=<php-code>` | `exec: <php-code>` |
| Load WordPress in a given context.<br>Default value: `auto` | `--context[=<context>]` | `context: <context>` |
| (Sub)commands to disable.<br>Default value: `[]` | *Not available as a flag* | `disabled_commands: <list>` |
| Whether to colorize the output.<br>Default value: `"auto"` | `--[no-]color` | `color: <bool>` |
| Show all PHP errors; add verbosity to WP-CLI bootstrap.<br>Default value: `false` | `--debug[=<group>]` | `debug: <group>` |
| Prompt the user to enter values for all command arguments, or a subset specified as comma-separated values.<br>Default value: `false` | `--prompt[=<assoc>]` | *Not available as an option* |
| Suppress informational messages.<br>Default value: `false` | `--quiet` | `quiet: <bool>` |
| List of Apache Modules that are to be reported as loaded.<br>Default value: `[]` | *Not available as a flag* | `apache_modules: <list>` |

# Config files

WP-CLI can automatically discover and read options from a few configuration file types (when present):

1. `wp-cli.local.yml` file inside the current working directory (or upwards).
2. `wp-cli.yml` file inside the current working directory (or upwards).
3. `~/.wp-cli/config.yml` file (path can be changed by setting the `WP_CLI_CONFIG_PATH` environment variable).

Besides the global parameters described above, configuration files can also contain defaults for any subcommand, as well as aliases to one or more WordPress installs.

Here's an annotated example `wp-cli.yml` file:

```
# Global parameter defaults
path: wp-core
url: http://example.com
user: admin
color: false
disabled_commands:
  - db drop
  - plugin install
require:
  - path-to/command.php

# Subcommand defaults (e.g. `wp config create`)
config create:
        dbuser: root
        dbpass:
        extra-php: |
                define( 'WP_DEBUG', true );
                define( 'WP_POST_REVISIONS', 50 );

# Aliases to other WordPress installs (e.g. `wp @staging rewrite flush`)
# An alias can include 'user', 'url', 'path', 'ssh', or 'http'
@staging:
        ssh: wpcli@staging.wp-cli.org
        user: wpcli
        path: /srv/www/staging.wp-cli.org
@production:
        ssh: wpcli@wp-cli.org:2222
        user: wpcli
        path: /srv/www/wp-cli.org

# Aliases can reference other aliases to create alias groups
# Alias groups can be nested
@both:
 - @staging
 - @production

# '_' is a special value denoting configuration options for this wp-cli.yml
_:
        # Merge subcommand defaults from the upstream config.yml, instead of overriding
        merge: true
        # Inherit configuration from an arbitrary YAML file
        inherit: prod.yml
```

# Remote (SSH) configuration

Using the `ssh` option, WP-CLI can be configured to run on a remote system rather than the current system. Along with the SSH protocol, WP-CLI also supports connecting to Docker containers (including docker-compose) and Vagrant VMs.

The connection type can be passed via the scheme of the `--ssh` parameter or `ssh` option.

Supported types are:

- `docker:[<user>@]<container_id>` - Runs WP-CLI in a running Docker container via `docker exec [--user <user>] <container_id> ...`
- `docker-compose:[<user>@]<container_id>` - Runs WP-CLI in a running Docker container via `docker-compose exec [--user <user>] <container_id> ...`
- `docker-compose-run:[<user>@]<container_id>` - Runs WP-CLI in a new Docker container via `docker-compose run [--user <user>] <container_id> ...`
- `vagrant` - Runs WP-CLI in a running Vagrant VM via `vagrant ssh ...`
- `[<user>@]<host>[:<port>]` (ssh) - Runs WP-CLI on a remote machine through an SSH connection via `ssh [-p <port>] [<user>@]<host> ...`

All connection types support an optional `path` suffix to specify a directory to `cd` to before running WP-CLI; `path` is a full system path starting with either `/` or `~`. (If `WP_CLI_SSH_PRE_CMD` is specified, `cd` is run after this pre-command.)

The SSH connection type also supports two advanced connection configuration options, which must be specified via an alias in the YAML configuration:

- `proxyjump` - Specifies a jumpbox connection string, which is passed to `ssh -J`

- `key` - Specifies the key (identify file) to use, which is passed to `ssh -i`

# Context configuration

In WP-CLI v2.6.0, a new global flag `--context=<context>` was added which allows users to select the WordPress context in which WP-CLI is supposed to execute its command(s).

One of the main goals is to allow WP-CLI to run updates on premium plugins and themes without requiring any special setup. From our initial testing, this allows a large range of popular premium extensions to *just work*™ with WP-CLI in terms of their update procedures.

Possible values for this flag:

- `cli`: The context which has been the default before introduction of this flag. This is something in-between a frontend and an admin request, to get around some of the quirks of WordPress when running on the console.
- `admin`: A context that simulates running a command as if it would be executed in the administration backend. This is meant to be used to get around issues with plugins that limit functionality behind an `is_admin()` check.
- `auto`: Switches between `cli` and `admin` depending on which command is being used. Currently, all `wp plugin *` and `wp theme *` commands use `admin`, while all other commands use `cli`.
- `frontend`: [WIP] This does nothing yet.

By default, the `--context` flag was set to `cli` in the initial release (v2.6.0). In WP-CLI v2.7.0 and later versions, the default was changed to `auto`. This gradual deployment allowed hosters and site owners to run tests on v2.6.0 by manually setting the context before the default behavior was changed.

If you are still using WP-CLI v2.6.0 but you want to use the default of `--context=auto`, you can do so by adding the necessary `context: auto` line to your global `wp-cli.yml` configuration file. Feel free to check the documentation on WP-CLI configuration files if this is new to you.

# Environment variables

WP-CLI's behavior can be changed at runtime through the use of environment variables:

- `WP_CLI_CACHE_DIR` - Directory to store the WP-CLI file cache. Default is `~/.wp-cli/cache/`.
- `WP_CLI_CONFIG_PATH` - Path to the global `config.yml` file. Default is `~/.wp-cli/config.yml`.
- `WP_CLI_CUSTOM_SHELL` - Allows the user to override the default `/bin/bash` shell used.
- `WP_CLI_DISABLE_AUTO_CHECK_UPDATE` - Disable WP-CLI automatic checks for updates.
- `WP_CLI_PACKAGES_DIR` - Directory to store packages installed through WP-CLI's package management. Default is `~/.wp-cli/packages/`.
- `WP_CLI_PHP` - PHP binary path to use when overriding the system default (only works for non-Phar installation).
- `WP_CLI_PHP_ARGS` - Arguments to pass to the PHP binary when invoking WP-CLI (only works for non-Phar installation).
- `WP_CLI_SSH_PRE_CMD` - When using `--ssh=<ssh>`, perform a command before WP-CLI calls WP-CLI on the remote server.
- `WP_CLI_STRICT_ARGS_MODE` - Avoid ambiguity by telling WP-CLI to treat any arguments before the command as global, and after the command as local.
- `WP_CLI_SUPPRESS_GLOBAL_PARAMS` - Set to `true` to skip showing the global parameters at the end of the help screen. This saves screen estate for advanced users.
- `WP_CLI_FORCE_USER_LOGIN` - Set to `1` to force the value provided to the `--user` flag to be interpreted as a login instead of an ID, to get around ambiguous types.

To set an environment variable on demand, simply place the environment variable definition before the WP-CLI command you mean to run.

```
# Use vim to edit a post
$ EDITOR=vim wp post edit 1
```

To set the same environment variable value for every shell session, you'll need to include the environment variable definition in your `~/.bashrc` or `~/.zshrc` file

```
# Always use vim to edit a post
export EDITOR=vim
```

---

# WP-CLI Hack Day

Welcome to WP-CLI Hack Day (https://make.wordpress.org/cli/2024/04/02/save-the-date-wp-cli-hack-day-on-friday-april-26th/)! We appreciate you taking time to contribute to the project.

**We'd love to help you submit at least one pull request today**. Given this goal, you'll likely want to **start with something small and attainable**. After you've submitted your first pull request for the day, you're welcome to work on something more ambitious.

When do you submit a pull request during Hack Day, please add `Related https://github.com/wp-cli/wp-cli/issues/5935 (https://github.com/wp-cli/wp-cli/issues/5935)` so we can easily keep track of them. We'll include them in a recap post at the end of the day.

We put together this guide to make contributing as straightforward as possible. Please also join the #cli channel in WordPress.org Slack (https://wordpress.slack.com/messages/C02RP4T41) (sign up instructions (https://make.wordpress.org/chat/)) to chat with other contributors, for questions, etc.

Your leads for the day are: schlessera (https://github.com/schlessera), danielbachhuber (https://github.com/danielbachhuber), swissspidy (https://github.com/swissspidy)

# Open Video Sessions

During the Hack Day, we'll have two open video chat sessions:

- 02:00-03:00 PST / 05:00-06:00 EST / 09:00-10:00 UTC / 11:00-12:00 CEST / 18:00-19:00 JST (https://www.timeanddate.com/worldclock/fixedtime.html?iso=20240426T0900)
- 08:00-09:00 PST / 11:00-12:00 EST / 15:00-16:00 UTC / 17:00-18:00 CEST / 00:00-01:00 JST (https://www.timeanddate.com/worldclock/fixedtime.html?iso=20240426T1500)

These sessions are a great opportunity to discuss remaining issues live, chat about the progress we've made, and to connect with the community. Feel free to join these to talk through any challenges or share your updates!

# Getting Started

If you normally use WP-CLI on your web host or via Brew, you're most likely using the Phar executable (`wp-cli.phar`). This Phar executable file is the "built", singular version of WP-CLI. It is compiled from a couple dozen repositories in the WP-CLI GitHub organization.

In order to make code changes to WP-CLI, you'll need to set up the `wp-cli-dev` development environment on your local machine. Before you can proceed, though, you'll need to make sure you have Composer (https://getcomposer.org/), PHP, and a functioning MySQL or MariaDB server on your local machine.

Once the prequisites are met, clone the GitHub repository and run the installation process:

```
git clone https://github.com/wp-cli/wp-cli-dev wp-cli-dev
cd wp-cli-dev
composer install
composer prepare-tests
```

The `wp-cli-dev` installation process clones all of WP-CLI's repositories to your local machine. After the installation process is complete, you can make changes in whichever repository you like. You'll need to fork the repository in order to push your feature branch, however. GitHub's CLI (https://github.com/cli/cli) is pretty helpful for this:

```
cd core-command
gh repo fork
```

We have started a video tutorial series that you can watch here: WP-CLI Contribution Tutorials (https://www.youtube.com/playlist?list=PL_B8Y6K6MH2d6T7pYa6dloUgk67mfBT4K). These videos will provide an overview of the packages and then go through setting up the development environment to allow you to run automated tests locally.

All WP-CLI pull requests are expected to have tests. Watch a ~10 minute video introduction (https://github.com/wp-cli/wp-cli/issues/5858), or see running and writing tests (https://make.wordpress.org/cli/handbook/contributions/pull-requests/#running-and-writing-tests) for the written version.

# Suggested Tickets

To help you be successful during the day, we curated a list of reasonably approachable and actionable issues.

Feel free to comment directly on the issue if you plan to work on it. We don't usually assign issues, so no need to worry about that.

**New contributors**

- Regenerating a single image size (re-)generates auto-scaled big images & auto-rotated images (https://github.com/wp-cli/media-command/issues/196)
- wp plugin update all doesnt display info which plugin is being updated (https://github.com/wp-cli/extension-command/issues/261)
- No thumbnail when importing PDF file using `media import` (https://github.com/wp-cli/media-command/issues/195)
- Allow theme slug renaming (https://github.com/wp-cli/extension-command/issues/74)
- Subtheme installation fails after installing main theme, if subtheme was tried first, due to left-behind directory (https://github.com/wp-cli/extension-command/issues/410)
- Include more information with wp plugin list (https://github.com/wp-cli/extension-command/issues/241)
- Output only matching IDs in db search (https://github.com/wp-cli/db-command/issues/158)
- DB-Check fails if Database requires SSL (https://github.com/wp-cli/config-command/issues/113)

See issues labeled 'good-first-issue' (https://github.com/issues?q=is%3Aopen+org%3Awp-cli+is%3Aissue+sort%3Aupdated-desc+label%3Agood-first-issue+) or issues labeled 'contributor-day' (https://github.com/issues?q=is%3Aopen+org%3Awp-cli+is%3Aissue+sort%3Aupdated-desc+label%3Acontributor-day) for a broader list.

**Seasoned contributors**

- Plugin Dependencies Support (https://github.com/wp-cli/extension-command/issues/407)

- Set WP_CLI_PACKAGES_DIR in config file (https://github.com/wp-cli/wp-cli/issues/5645)
- Cache wp plugin install from GitHub (https://github.com/wp-cli/extension-command/issues/363)
- Improve speed of import when uploads are available locally (https://github.com/wp-cli/import-command/issues/83)
- Introduce a dedicated search-replace url command (https://github.com/wp-cli/search-replace-command/issues/186)

You're obviously welcome to work on any other issue (https://github.com/issues?q=is%3Aopen+org%3Awp-cli+is%3Aissue+sort%3Aupdated-desc) you'd like too! Bug fixes in particular are very welcome (https://github.com/issues?q=is%3Aopen%20org%3Awp-cli%20is%3Aissue%20sort%3Acreated-desc%20label%3Abug). This day can be a good opportunity to discuss trickier issues and brainstorm approaches.

# Thank You for Contributing

We appreciate your taking the time to participate in the Hack Day and contribute to `wp-cli`. Have fun, and we look forward to seeing you on Slack (https://wordpress.slack.com/messages/C02RP4T41)!

---

# Pull Requests

WP-CLI follows a pull request workflow for changes to its code (and documentation). Whether you want to fix a bug or implement a new feature, the process is pretty much the same:

0. Search existing issues (https://github.com/search?q=org%3Awp-cli+is%3Aopen+sort%3Aupdated-desc&type=issues); if you can't find anything related to what you want to work on, open a new issue in the appropriate repository so that you can get some initial feedback.
    1. Opening an issue before submitting a pull request helps us provide architectural and implementation guidance before you spend too much time on the code.
1. Fork the repository you'd like to modify, either the framework or one of the command packages.
    1. See Setting Up for more details on configuring the codebase for development.
2. Create a branch for each issue you'd like to address. Commit your changes.
3. Push the code changes from your local clone to your fork.
4. Open a pull request. It doesn't matter if the code isn't perfect. The idea is to get it reviewed early and iterate on it.
5. Respond to code review feedback (https://make.wordpress.org/cli/handbook/code-review/) in a timely manner, recognizing development is a collaborative process.
6. Once your pull request has passed code review, it will be merged into the default branch and be in the pipeline for the next release.

New to WP-CLI commands? You may want to start with the commands cookbook (https://make.wordpress.org/cli/handbook/commands-cookbook/) to learn more about how commands work.

There are three classes of repos you might want to edit:

- wp-cli/wp-cli (https://github.com/wp-cli/wp-cli/) is the framework implementation.
- wp-cli/scaffold-command (https://github.com/wp-cli/scaffold-command/) is an example of a command implementation. There are many others.
- wp-cli/handbook (https://github.com/wp-cli/handbook/) contains documentation rendered in the handbook.

# Expectations

When submitting a pull request, there are several expectations to keep in mind.

**Tests are required**

Most of the time, we'll ask that functional or unit tests be added to cover the change. If it's a new feature, the pull request needs tests. If it's fixing a bug, the pull request needs tests.

See the documentation below for more information on writing and running tests.

**Follow WordPress Coding Standards**

While not yet strictly enforced, the WP-CLI project generally follows the WordPress Coding Standards (http://make.wordpress.org/core/handbook/coding-standards/). We may ask you to clean up your pull request if it deviates too much.

**Please refrain from unnecessary code churn**

Code refactoring should not be done just because we can. With a years-old codebase, there's an infinite number of best practice, readability, or consistency improvements that could be made. However, engaging on any of them has non-obvious costs: our time and attention, making Git history more difficult to review, etc. Any code changes should have clear and obvious value.

**Contributions are atomic**

To make it far easier to merge your code, each pull request should only contain one conceptual change. Keeping contributions atomic keeps the pull request discussion focused on one topic and makes it possible to approve changes on a case-by-case basis.

If you submit a pull request with multiple conceptual changes, we'll ask you to resubmit as separate pull requests.

**Make regular progress on your contribution**

Through our code review process (https://make.wordpress.org/cli/handbook/code-review/), we'll work with you to make sure your pull request is ready for merge. But if changes are needed and we haven't heard from you in **two weeks**, we'll consider the pull request abandoned. Someone else may pick it up and make the changes required. Or it may be closed.

If you need to step away for any reason, make a comment on the pull request or the related issue so we can pick things up or put things on hold when needed.

# Setting up

If you haven't submitted a pull request before, you'll want to install WP-CLI for local development. Depending on whether you want to work on a particular command/package or on the entire project as a whole, the process is slightly different.

## Working on a specific command/package

1. Install Composer (https://getcomposer.org/) and hub (https://hub.github.com/) if you don't already have them.
2. Clone the git repository of the command/package you want to work on to your local machine. As an example for working on the `wp core` command: `hub clone wp-cli/core-command`
3. Change into the cloned directory and fork WP-CLI: `cd core-command`.
4. Install all Composer dependencies: `composer install`
5. Verify WP-CLI was installed properly: `vendor/bin/wp --info`

Within this package, you should preferably use `vendor/bin/wp` to run the command. Just using `wp` should work as well, but by doing that you might run the command through a different version of the framework and thus getting an unexpected result.

## Working on the project as a whole

1. Install Composer (https://getcomposer.org/) and hub (https://hub.github.com/) if you don't already have them.
2. Clone the WP-CLI git repository to your local machine: `git clone git@github.com:wp-cli/wp-cli.git ~/wp-cli`
3. Change into the cloned directory and fork WP-CLI: `cd ~/wp-cli`. If you are going to work on the core framework itself, run `hub fork` here to create a pushable repository on GitHub.
4. Install all Composer dependencies: `composer install --prefer-source`
5. Alias the `wp` command to your new WP-CLI install: `alias wp='~/wp-cli/bin/wp'`
6. Verify WP-CLI was installed properly: `wp --info`

Commands bundled with WP-CLI (e.g. `wp scaffold plugin`) will be editable from the `vendor/wp-cli` directory (e.g. `vendor/wp-cli/scaffold-command`). The `--prefer-source` flag when installing WP-CLI ensures each command is installed as a Git clone, making it easier to commit to.

Commands available for standalone installation (e.g. `wp dist-archive`) can be installed from source (e.g. `wp package install git@github.com:wp-cli/dist-archive-command.git`). Run `wp package path <package-name>` to find the appropriate directory to edit.

Importantly, you'll need to fork each repository in order to have an `origin` to push to. Run `hub fork` to fork a repository from the command-line:

```
$ cd vendor/wp-cli/scaffold-command
$ hub fork
Updating danielbachhuber
From https://github.com/wp-cli/scaffold-command
 * [new branch]      master     -> danielbachhuber/master
new remote: danielbachhuber
$ git remote -v
danielbachhuber git@github.com:danielbachhuber/scaffold-command.git (fetch)
danielbachhuber git@github.com:danielbachhuber/scaffold-command.git (push)
```

Once you've done so, you'll have a fork in your GitHub account and new remote you can push to. If you didn't install `hub`, you'll need to fork the target repo through the web UI and manually add your fork as a remote.

# Running and writing tests

There are three types of automated tests:

- code style sniffers, implemented using PHPCS (https://github.com/squizlabs/PHP_CodeSniffer)
- functional tests, implemented using Behat (http://behat.org)
- unit tests, implemented using PHPUnit (http://phpunit.de/)

## Code style sniffers

The sniffers ensure that the code adheres to a given code style, to avoid unneeded discussions about less relevant details like spacing or alignments.

They also check for known sources of bugs and PHP compatibility problems.

To run the sniffs:

```
composer phpcs
```

To fix the errors and warnings that can be automatically fixed:

```
vendor/bin/phpcbf
```

# Functional tests

WP-CLI uses Behat (https://behat.org/) as its functional test suite. Stability between releases is an important contact WP-CLI makes with its users. Functional tests are different than unit tests in that they execute the entire WP-CLI command, and ensure they always work as expected.

Every repository has a `features/` directory with one or more YAML (https://yaml.org/)-formatted `*.feature` files. Here's an example of what you might see:

```
Feature: Manage WordPress options

  Scenario: Read an individual option
    Given a WP install

       When I run `wp option get home`
       Then STDOUT should be:
      """
         https://example.com
         """
```

In this example:

- `Feature:` documents the scope of the file.
- `Scenario:` describes a specific test.
- `Given` provides the initial environment for the test.
- `When` causes an event to occur.
- `Then` asserts what's expected after the event is complete.

In a slightly more human-friendly form:

> *I have a WordPress installation. When I run `wp option get home`, then the output from the command should be 'https://example.org'.*

Essentially, WP-CLI's functional test suite lets you *describe how a command should work*, and then run that description as a functional test.

Notably, Behat is simply the framework for writing these tests. We've written our own custom `Given`, `When`, and `Then` step definitions (example (https://github.com/wp-cli/wp-cli-tests/blob/560ed5ca2776b6b3b66c79a6e6dc62904ae20b3b/src/Context/GivenStepDefinitions.php#L105-L110), example (https://github.com/wp-cli/wp-cli-tests/blob/560ed5ca2776b6b3b66c79a6e6dc62904ae20b3b/src/Context/WhenStepDefinitions.php#L34-L42)).

## Creating a test database

Before running the functional tests, you'll need a MySQL (or MariaDB) user called `wp_cli_test` with the password `password1` that has full privileges on the MySQL database `wp_cli_test`. To override these credentials you can make use of the database credentials constants of wp-cli-tests (https://github.com/wp-cli/wp-cli-tests#the-database-credentials)

If your user has the correct permissions, the database can also be set up by running `composer prepare-tests`. This will create the database and the user and configure the necessary privileges. Note that this operation is not needed for every test run, it only needs to be run the first time for the initial setup.

**Note: If you are using MySQL >= 8.0, you may experience inconsistencies with WP-CLI successfully connecting to the database. MySQL 8.0 changed the default authentication plugin and some clients (such as PHP) do not yet support this change. More information can be found on this blog post (https://jonathandesrosiers.com/2019/02/trouble-connecting-to-database-when-using-mysql-8-x/).**

## Running the test suite

Then, to run the entire test suite:

```
composer behat
```

Or to test a scenario at a specific line:

```
composer behat -- features/core.feature:10
```

Or to test a single feature:

```
composer behat -- features/core.feature
```

Or to test a single feature with more verbosity:

```
composer behat -- features/core.feature --format pretty
```

To run only the tests that failed during the previous run:

```
composer behat-rerun
```

When writing new tests, to see which step definitions are available:

```
composer behat -- --definitions l
```

More info can be found by using `composer behat -- --help`.

## Unit tests

The unit test files are in the `tests/` directory.

To run the unit tests, execute:

```
composer phpunit
```

To run a specific unit test, you can use:

```
composer phpunit -- filter=<method name>
```

## Running all tests in one go

To run all tests in one go:

```
composer test
```

This will run all the tests that the package is set up to use, based on the presence of the respective configuration files.

Each repository is configured to run all of its active tests on every code push. The wp-cli/automated-tests (https://github.com/wp-cli/automated-tests) repository runs all tests for all repositories on a regular basis.

# Finally...

Thanks! Hacking on WP-CLI should be fun. If you find any of this hard to figure out, let us know so we can improve our process or documentation!

---

# How to put the site in maintenance mode

WP-CLI offers a command to enable, disable maintenance mode and check if maintenance mode is enabled or not.

## Step 1 - Check the status

```
$ wp maintenance-mode status
Maintenance mode is not active.
```

## Step 2 - Enable maintenance mode

```
$ wp maintenance-mode activate
Enabling Maintenance mode...
Success: Activated Maintenance mode.
```

### Step 3 - Disable maintenance mode

```
$ wp maintenance-mode deactivate
Disabling Maintenance mode...
Success: Deactivated Maintenance mode.
```

---

# Shell Friends

As you advance upon your use of WP-CLI, you'll find that a little bit of command line knowledge can have a huge impact on your workflow. Here are some of our favorite shell helper utilities.

## You Should Know

### Search through your bash history

Did you know that every command you run on your shell is saved to history? Search through your history with `CTRL + R`:

```
$ wp core download --version=nightly --force
bck-i-search: wp
```

When 'bck-i-search' appears, your keystrokes will search against commands saved in your bash history. Hit `return` to run the current selection.

Or another way to search history is grepping the output from the `history` command like:

```
$ history | grep wp
```

Any of the commands found in that list can be re-executed by bang-number, so for example if the output says your desired command is #218, you just do `!218`

### Combine WP-CLI commands

In many cases, it can be extremely powerful to be able to pass the results of one command to another. Composability is a key philosophy (https://make.wordpress.org/cli/handbook/philosophy/) of WP-CLI, and there are two common approaches for composing commands.

*Command substitution* passes the output of one command to another command, without any transformation to the output.

`wp post list` only lists posts; it doesn't perform any operation on them. In this example, the command lists page ids as space-separated values.

```
$ wp post list --post_type='page' --format=ids
1164 1186
```

Combining `wp post list` with `wp post delete` lets you easily delete all posts. In this example, `$()` lets us pass the space-separated page ids to `wp post delete`.

```
$ wp post delete $(wp post list --post_type='page' --format=ids)
Success: Trashed post 1164.
Success: Trashed post 1186.
```

If you need a bit more flexibility, *xargs* lets you pass the output of one command to another command, while performing minor transformation on the output.

You may want to assign all editor capabilities to the author role. However, `wp cap list` lists capabilities separated by newlines, and `wp cap add` only accepts space-separated capabilities. Enter, `xargs`, whose default behavior is to split newline output into a space-separated list. Note the `|` shell operator, which passes the results of `wp cap list` to `xargs`. Without `|`, you'll see a WP-CLI error.

```
$ wp cap list 'editor' | xargs wp cap add 'author'
Success: Added 24 capabilities to 'author' role.
```

`wp user generate` only generates users; it doesn't perform supplemental operations. In this example, `wp user generate` passes user ids to `xargs`, which splits the space-separated ids into a list and calls `wp user meta add` for each.

```
$ wp user generate --count=5 --format=ids | xargs -0 -d ' ' -I % wp user meta add % foo bar
Success: Added custom field.
Success: Added custom field.
Success: Added custom field.
Success: Added custom field.
Success: Added custom field.
```

## Define aliases, short macros to common commands

If you find yourself running the same commands quite often, you can define aliases to them for easier access.

Run all three status check commands with one `check-all` alias. In this example, running `alias` creates a `check-all` alias for the current shell session. Save the same statement to your `~/.bashrc` or `~/.zshrc` to always have it available.

```
$ alias check-all='wp core check-update && wp plugin list --update=available && wp theme list --update=available'
$ check-all
+-----------------+-------------+-----------------------------------------------------------------+
| version         | update_type | package_url                                                     |
+-----------------+-------------+-----------------------------------------------------------------+
| 4.7-beta4-39322 | minor       | https://wordpress.org/nightly-builds/wordpress-latest.zip |
+-----------------+-------------+-----------------------------------------------------------------+
+-------------------------+----------+-----------+--------------+
| name                    | status   | update    | version      |
+-------------------------+----------+-----------+--------------+
| akismet                 | inactive | available | 3.1.8        |
| co-authors-plus         | inactive | available | 3.1.1        |
| wp-redis                | inactive | available | 0.2.2        |
| rest-api                | active   | available | 2.0-beta13.1 |
| wp-api-oauth1           | inactive | available | 0.2          |
+-------------------------+----------+-----------+--------------+
+----------------------+----------+-----------+---------+
| name                 | status   | update    | version |
+----------------------+----------+-----------+---------+
| p2                   | inactive | available | 1.4.2   |
+----------------------+----------+-----------+---------+
```

## Save command output

WP-CLI commands send output to both `STDOUT` and `STDERR`, depending on the nature of the message. You may not notice there are two destinations, because WP-CLI renders both inside your shell. However, if you want to capture your output to a file, the distinction matters.

Simply using `>` will capture STDOUT from the command to a file.

```
$ wp import wordpress.wxr --authors=create > import.log
```

Redirect STDERR to STDOUT with `2>&1`, and then capture STDOUT to a log file.

```
$ wp import wordpress.wxr --authors=create > import.log 2>&1
```

When you're capturing output to a file, you won't see the output in your current shell session. However, you can open a second shell session and run `tail -f <file>` to see the output as it's added to the file.

Alternatively, you can use `tee` that writes to both standard output and files. You only have to pipe the output to the command and specify the file name.

```
$ wp import wordpress.wxr --authors=create | tee import.log
```

This will display the output in the current shell screen and also write it to the log file.

Note that if the file already exists, it will be overwritten. If you want to append the output to the file, use the `-a` option.

```
$ wp import wordpress.wxr --authors=create | tee -a import.log
```

# Snippets

Are you an expert in bash or zsh? Share your tips here.

## Take a look at the plugin changelog

Needs `elinks` to browse HTML.

```
wget -qO- http://api.wordpress.org/plugins/info/1.0/akismet|php -r '$seri=unserialize(stream_get_contents(STDIN)); echo $seri->secti
```

Explanation

- set wget quiet & query the WordPress.org Plugin API
- unserialize with php, `stream_get_contents(STDIN)` means "get all from *stdin*"
- echo the changelog part from the API's reply
- fire up elinks (a console browser) to view the changelog

## Start wp-cli with ABSPATH in the current dir and under the current dir's owner

```
#!/bin/bash

sudo -u $(stat . -c %U) -- wp --path="$PWD" "$@"
```

Explanation

The `stat` command returns the owner of the current directory, WordPress root.

## Install and Configure WordPress with WP-CLI

```
wp_install ()
{
    wp core download --path=$1;
    cd $1;
    read -p 'name the database:' dbname;
    wp config create --dbname=$dbname --dbuser=root --dbpass=awoods --dbhost=localhost;
    wp db create;
    wp core install --prompt
}

$ source ~/.bashrc
$ wp_install new-site
```

Explanation

Add this function to your ~/.bashrc are reload your shell (or open a new shell). You'll need to substitute these database credentials with your own. When you need to create a new WordPress site, call this function and specify the name of the directory where you want to create the site. This emulates the web-based install process.

## List all image URL-s in posts

```
wp post list --field=ID|xargs -I % wp post get % --field=post_content|sed -ne 's;.*\(https\?\S\+\(jpe\?g\|png\|gif\)\).*;\1;gp'
```

Explanation

- List all post ID-s
- Get each content (xargs)
- Display only image URL-s (sed)

## Create a page from a file and flag it with the file name

```
wp post create new_page.html --post_type=page --post_title="New Page" --porcelain | xargs -I % wp post meta add % imported_from new_
```

Explanation

- Create a page (--porcelain will return only the new post ID)
- Create post meta with xargs using "-I %" to signify the placeholder template for the new post ID

## Change to a certain WordPress installation's directory from a menu

```
#!/bin/bash

WP_TOP_PATH="/home/"
MENU_TEXT="Choose an installation"
GAUGE_TEXT="Searching for WordPress"

declare -a MENU
WPS="$(wp --allow-root find "$WP_TOP_PATH" --field=version_path)"
WP_TOTAL="$(wc -l <<< "$WPS")"
WP_COUNT="0"

while read -r WP; do
    WP_LOCAL="${WP%wp-includes/version.php}"

    NAME="$(cd "$WP_LOCAL"; sudo -u "$(stat . -c %U)" -- wp --no-debug --quiet option get blogname)"
    if [ -z "$NAME" ]; then
        NAME="(unknown)"
    fi
    MENU+=( "$WP_LOCAL" "$NAME" )

    echo "$((++WP_COUNT * 100 / WP_TOTAL))".
done <<< "$WPS" > >(whiptail --gauge "$GAUGE_TEXT" 7 74 0)

WP_LOCAL="$(whiptail --title "WordPress" --menu "$MENU_TEXT"  $(($(#MENU[*]} / 2 + 7)) 74 10 "${MENU[@]}" 3>&1 1>&2 2>&3)"

if [ $? -ne 0 ] || [ ! -d "$WP_LOCAL" ]; then
    echo "Cannot find '${WP_LOCAL}'" 1>&2
    exit 100
fi

echo "cd ${WP_LOCAL}"
```

Explanation

- Needs `wp-cli/find-command` and `whiptail`
- Find all WordPress installations below `$WP_TOP_PATH` - must be run as root
- Display a progress bar while getting `blogname` of each installation
- Choose one installation from a nice menu and display `cd` command for it

## Discard header row from a CSV

By default, `--format=csv` includes a header row:

```
$ wp user list --format=csv
ID,user_login,display_name,user_email,user_registered,roles
1,daniel,daniel,daniel@handbuilt.co,"2022-12-21 23:05:16",administrator
```

If you'd like to discard the header row, `tail` is pretty helpful:

```
$ wp user list --format=csv | tail -n +2
1,daniel,daniel,daniel@handbuilt.co,"2022-12-21 23:05:16",administrator
```

## Sort plugins or themes by certain column(s)

- **Challenge**: wp-cli's `<plugin|theme> list` commands have no option by which column(s) to sort.
- **Solution**: You can output as `--format=csv` and then simply pipe into a CLI app which has sorting functions built-in.
    - E.g. if you want to sort by status and then by name with miller (https://github.com/johnkerl/miller#readme), short form `mlr`:

```
$ wp plugin list --format=csv | mlr --icsv --opprint sort -f status,name
```

- Explanation:
  - Manpage says: `sort -f {comma-separated field names} Lexical ascending`
  - In our example this means: Sort the plugins lexically ascending, first by status, then by name.

---

# Customize doctor diagnostic checks

Even though `wp doctor` comes with a number of default diagnostic checks (https://make.wordpress.org/cli/handbook/doctor-default-checks/), it's designed with extensibility at its core. Checks are defined at runtime, read from a `doctor.yml` configuration file naming each check with its options.

## doctor.yml format

Let's take a look at the first two checks in the included `doctor.yml`:

```
autoload-options-size:
  check: Autoload_Options_Size
constant-savequeries-falsy:
  check: Constant_Definition
  options:
    constant: SAVEQUERIES
    falsy: true
```

In this example:

- 'autoload-options-size' and 'constant-savequeries-falsy' are the *names* for the checks. Names must be unique amongst all registered checks.
- `Autoload_Options_Size` and `Constant_Definition` are reusable check classes in the `runcommand\Doctor\Checks` namespace. You can use them too, or you can write your own class extending `runcommand\Doctor\Checks\Check` and supply it as 'class: yourNamespace\yourClassName'.
- 'constant' and 'falsy' are configuration options accepted by the `Constant_Definition` class. In this case, we're telling doctor to ensure `SAVEQUERIES` is either false or undefined.

For the sake of completeness, it's also worth noting `Autoload_Options_Size` accepts 'threshold_kb' as an optional configuration option. The default value for 'threshold_kb' is 900, so it doesn't needed be included in the `doctor.yml`.

## Custom doctor.yml configuration files

Run your own doctor checks by creating a `doctor.yml` and supplying it with `wp doctor check --config=doctor.yml`. Use different configurations for different environments by creating separate `prod.yml` and `dev.yml` files.

If you want your custom file to extend an existing doctor config, you can use the magical _ config file option to define which config file to inherit. 'default' is a magic reference to the bundled `doctor.yml`; you also specify an entire file path.

Take a look at this example:

```
_:
  inherit: default
  skipped_checks:
    - autoload-options-size
constant-disallow-file-mods-falsy:
  check: Constant_Definition
  options:
    constant: DISALLOW_FILE_MODS
    falsy: true
plugin-akismet-active
  check: Plugin_Status
  options:
    plugin: akismet
    status: active
plugin-akismet-valid-api-key:
  class: Akismet_Valid_API_Key
  require: akismet-valid-api-key.php
```

This custom `doctor.yml` file:

- Inherits all default diagnostic checks except for 'autoload-options-size'.
- Defines a 'constant-disallow-file-mods-falsy' check to ensure the `DISALLOW_FILE_MODS` constant is falsy.
- Defines a 'plugin-akismet-active' check to ensure Akismet is active.
- Defines a 'plugin-akismet-valid-api-key' custom check in a akismet-valid-api-key.php file (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-write-custom-check/) to ensure Akismet has a valid API key.

# Available check types

Some `wp doctor` check types are configurable, meaning the default setting can be changed, while other check types are abstracted in such a way that they can be reusable. For instance, the `Autoload_Options_Size` check accepts an option 'threshold_kb' while `Plugin_Status` accepts 'plugin' and 'status' as options.

The configurable check types include:

- `check: Autoload_Options_Size`: Accepts 'threshold_kb' as an option to set the threshold in kilobytes. Default value is 900.
- `check: Cron_Count`: Accepts 'threshold_count' as an option to set the threshold of total cron jobs. Default value is 50.
- `check: Cron_Duplicates`: Accepts 'threshold_count' as an option to set the threshold of duplicate cron jobs. Default value is 10.
- `check: Plugin_Active_Count`: Accepts 'threshold_count' as an option to set the threshold of total active plugins. Default is 80.
- `check: Plugin_Deactivated`: Accepts 'threshold_percentage' as an option to set the threshold of percentage deactivated plugins. Default is 40.

The abstracted check types include:

- `check: Constant_Definition`: Assert a given constant as defined, a specific value, or falsy. Learn more (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-constant-value/).
- `check: File_Contents`: Check all or a selection of WordPress files for a given regex pattern. Learn more (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-file-contents/).
- `check: Option_Value`: Assert a given option as a specific value. Learn more (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-option-value/).
- `check: Plugin_Status`: Assert a given plugin as active, installed, or uninstalled. Learn more (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-check-plugin-status/).

Of course, some check types don't need configuration options:

- `check: Core_Update`: Errors when new WordPress minor release is available; warns for major release.
- `check: Core_Verify_Checksums`: Verifies WordPress files against published checksums; errors on failure.
- `check: Plugin_Update`: Warns when there are plugin updates available.
- `check: Theme_Update`: Warns when there are theme updates available.

You can write your own custom check type (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-write-custom-check/) by extending the `runcommand\Doctor\Checks\Check` class.

---

# Roadmap

WP-CLI is released every 3-4 months, typically around the beta period of WordPress's release cycle (https://wordpress.org/about/roadmap/):

- Current: v2.11.0 (https://github.com/wp-cli/wp-cli/releases/tag/v2.11.0) (August 8th, 2024)

Patch versions are released on an as-needed basis, usually to address bugs or regressions. The current version of WP-CLI is the only officially supported version.

The current longer-term goals of the project are:

- Improve contributor onboarding by streamlining development environment setup, testing and documentation.
- Rethink the role of WP-CLI scaffolding as an educational tool and a practical application of best practices.

New features are developed as packages first. Doing so enables:

- Faster feature iteration with less risk.
- More contribution opportunities.
- Reduced complexity and maintenance burden for the core project.

To suggest new ideas, head on over to the wp-cli/ideas (https://github.com/wp-cli/ideas) repository.

For more details on how WP-CLI releases are produced, please see Philosophy (https://make.wordpress.org/cli/handbook/philosophy/) and Governance (https://make.wordpress.org/cli/handbook/governance/). For more information about how WP-CLI handles its requirements, please see WP-CLI PHP Requirements Strategy (https://make.wordpress.org/cli/2019/01/15/wp-cli-php-requirements-strategy/).

---

# Check status of a given plugin

One of the check types included in `wp doctor` is `Plugin_Status`, or the ability to assert that a given plugin should be active, installed, or uninstalled. Although the check type isn't use by any of the default diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-default-checks/), you can use the `Plugin_Status` check type in your custom `doctor.yml` configuration file (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-customize-config/).

As an example, here are two checks using `Plugin_Status`, one which ensures Akismet is active on the system and other which ensures Hello Dolly is uninstalled:

```
plugin-akismet-active:
  check: Plugin_Status
  options:
    name: akismet
    status: active
plugin-hello-uninstalled:
  check: Plugin_Status
  options:
    name: hello
    status: uninstalled
```

Run together, you might see:

```
$ wp doctor check --config=plugin-status.yml --all
+--------------------------+---------+-------------------------------------------------------------------------+
| name                     | status  | message                                                                 |
+--------------------------+---------+-------------------------------------------------------------------------+
| plugin-akismet-active    | success | Plugin 'akismet' is 'active' as expected.                               |
| plugin-hello-uninstalled | error   | Plugin 'hello' is 'inactive' but expected to be 'uninstalled'. |
+--------------------------+---------+-------------------------------------------------------------------------+
```

The `Plugin_Status` check type accepts the following options:

- 'name': Name of the plugin as it appears in `wp plugin list`.
- 'status': Expected plugin status as one of 'active', 'installed', or 'uninstalled'.

---

# Handbook

Here are some helpful guides and resources for using WP-CLI.

Can’t find what you’re looking for? Open an issue (https://github.com/wp-cli/handbook/issues) to request improvements.

## Guides

- **Installing (https://make.wordpress.org/cli/handbook/guides/installing/)** - Recommended and alternative installation mechanisms.
- **Quick start (https://make.wordpress.org/cli/handbook/guides/quick-start/)** - Where to begin after you've installed WP-CLI for the first time.
- **Running commands remotely (https://make.wordpress.org/cli/handbook/guides/running-commands-remotely/)** - Learn how to remotely control multiple servers at once.
- **Commands cookbook (https://make.wordpress.org/cli/handbook/guides/commands-cookbook/)** - The full 101 how commands work, writing your own, and sharing them with the world.
- **Common issues and their fixes (https://make.wordpress.org/cli/handbook/guides/common-issues/)** - In case of fire, break glass.
- **External resources (https://make.wordpress.org/cli/handbook/guides/external-resources/)** - Blog posts, slides and videos from users.
- **Force output to a specific locale (https://make.wordpress.org/cli/handbook/guides/force-output-specific-locale/)** - Localisation issue
- **Identify a Plugin or Theme Conflict (https://make.wordpress.org/cli/handbook/guides/identify-plugin-theme-conflict/)** - Debugging advise
- **Sharing WP-CLI Packages (https://make.wordpress.org/cli/handbook/guides/sharing-wp-cli-packages/)** - Some words about your environment
- **Troubleshooting Guide (https://make.wordpress.org/cli/handbook/guides/troubleshooting/)** - Some help to troubleshoot

## References

- **Global parameters (https://make.wordpress.org/cli/handbook/references/config/)** - Variables defining how a command is executed, including which WordPress user the command is run as and which WordPress instance the command is run against.
- **Built-in commands (https://developer.wordpress.org/cli/commands/)** - Commands included in every copy of WP-CLI.
- **Internal API (https://make.wordpress.org/cli/handbook/references/internal-api/)** - Stable utilities considered safe to use in community commands.

- **Documentation standards (https://make.wordpress.org/cli/handbook/references/documentation-standards/)** - Standards for annotating WP-CLI commands.
- **Hosting companies (https://make.wordpress.org/cli/handbook/references/hosting-companies/)** - List of hosting companies where WP-CLI is installed by default.
- **Shell friends (https://make.wordpress.org/cli/handbook/references/shell-friends/)** - Helpful shortcuts for bash and zsh.
- **Integrated tools (https://make.wordpress.org/cli/handbook/references/tools/)** - Plugins, wrappers, and other projects that integrate with WP-CLI in some form.

## Contributing

- **Bug reports (https://make.wordpress.org/cli/handbook/contributions/bug-reports/)** - Help ensure your issue is resolved in a timely manner.
- **Contributing (https://make.wordpress.org/cli/handbook/contributions/contributing/)** - An introduction to the contributing process.
- **WordCamp Contributor Day (https://make.wordpress.org/cli/handbook/contributions/contributor-day/)** - Quick-start guide for WordCamp Contributor Days.
- **Ideas (https://github.com/wp-cli/ideas)** - Up-vote existing ideas or submit your own.
- **Governance (https://make.wordpress.org/cli/handbook/contributions/governance/)** - Summary of those behind WP-CLI.
- **Implementation details (https://make.wordpress.org/cli/handbook/contributions/implementation-details/)** - Catalog of historical design decisions.
- **Philosophy (https://make.wordpress.org/cli/handbook/contributions/philosophy/)** - Guidelines which inform project scope, command organization, and behavior.
- **Pull requests (https://make.wordpress.org/cli/handbook/contributions/pull-requests/)** - Submit your first bug fix or new feature.
- **Release checklist (https://make.wordpress.org/cli/handbook/contributions/release-checklist/)** - Tasks performed during the process of tagging a release.
- **Roadmap (https://make.wordpress.org/cli/handbook/contributions/roadmap/)** - Where WP-CLI is going in the future.
- **Code Review (https://make.wordpress.org/cli/handbook/contributions/code-review/)** - Quality Assurance on WP-CLI
- **Committers credo (https://make.wordpress.org/cli/handbook/contributions/committers-credo/)** - Product quality, Stellar judgement, Consistent participation
- **Repository Management (https://make.wordpress.org/cli/handbook/contributions/repository-management/)** - Naming, Versions, Milestones, Labels

## Misc

- **Plugin unit tests (https://make.wordpress.org/cli/handbook/misc/plugin-unit-tests/)** - How to set up and run PHPUnit tests for a WordPress plugin.

---

# Write a custom check to perform an arbitrary assertion

Because `wp doctor` checks are built on top of a foundational abstraction, it's relatively straightforward for you to write your own custom check. The basic requirement is that you create a class extending `runcommand\Doctor\Checks\Check` that implements a `run()` method. The `run()` must set a status and message based on whatever procedural logic As an example, here's an annotated custom check to assert Akismet is activated with a valid API key:

```php
<?php

/**
 * Ensures Akismet is activated with the appropriate credentials.
 */
class Akismet_Activated extends runcommand\Doctor\Checks\Check {

	public function __construct( $options = array() ) {
		parent::__construct( $options );
		// Every check is to run on 'after_wp_load' by default.
		// You could instead use 'before_wp_load' or 'after_wp_config_load'
		$this->set_when( 'after_wp_load' );
	}

	public function run() {
		// If the Akismet isn't activated, bail early.
		if ( ! class_exists( 'Akismet' ) ) {
			$this->set_status( 'error' );
			$this->set_message( "Akismet doesn't appear to be activated." );
			return;
		}
		// Verify that the API exists.
		$api_key = Akismet::get_api_key();
		if ( empty( $api_key ) ) {
			$this->set_status( 'error' );
			$this->set_message( 'API key is missing.' );
			return;
		}
		// Verify that the API key is valid.
		$verification = Akismet::verify_key( $api_key );
		if ( 'failed' === $verification ) {
			$this->set_status( 'error' );
			$this->set_message( 'API key verification failed.' );
			return;
		}
		// Everything looks good, so report a success.
		$this->set_status( 'success' );
		$this->set_message( 'Akismet is activated with a verified API key.' );
	}

}
```

If the class were placed in an `akismet-activated.php` file, you could register it with:

```
plugin-akismet-activated:
  class: Akismet_Activated
  require: akismet-activated.php
```

Then, run the config file:

```
$ wp doctor check plugin-akismet-activated --config=doctor.yml
+--------------------------+--------+---------------------+
| name                     | status | message             |
+--------------------------+--------+---------------------+
| plugin-akismet-activated | error  | API key is missing. |
+--------------------------+--------+---------------------+
```

---

# Contributing

Welcome and thanks!

We appreciate you taking the initiative to contribute to WP-CLI. It's because of you, and the community around you, that WP-CLI is such a great project.

**Contributing isn't limited to just code.** We encourage you to contribute in the way that best fits your abilities, by writing tutorials, giving a demo at your local meetup, helping other users with their support questions, or revising our documentation.

Please take a moment to read these guidelines in depth. Following the guidelines helps to communicate that you respect the time of the other contributors to the project. In turn, they'll do their best to reciprocate that respect when working with you, across timezones and around the world.

Should you have any questions about contributing, please join the `#cli` channel in the WordPress.org Slack (https://make.wordpress.org/chat/).

## Reporting Security Issues

The WP-CLI team and WordPress community take security bugs seriously. We appreciate your efforts to responsibly disclose your findings, and will make every effort to acknowledge your contributions.

To report a security issue, please visit the WordPress HackerOne (https://hackerone.com/wordpress) program.

## Reporting a bug

Think you've found a bug? We'd love for you to help us get it fixed.

Before you create a new issue, you should search existing issues (https://github.com/search?q=org%3Awp-cli+label%3Abug+is%3Aopen+sort%3Aupdated-desc&type=issues) to see if there's an existing resolution to it, or if it's already been fixed in a newer version of WP-CLI. You should also check our documentation on common issues and their fixes (https://make.wordpress.org/cli/handbook/common-issues/).

Once you've done a bit of searching and discovered there isn't an open or fixed issue for your bug, please follow our guidelines for submitting a bug report (https://make.wordpress.org/cli/handbook/bug-reports/) to make sure it gets addressed in a timely manner.

## Creating a pull request

Want to contribute a new feature? WP-CLI is a mature project, and already chock-full of useful functionality. Please first open an ideas issue to suggest new commands (https://github.com/wp-cli/ideas/issues/new), or open an issue in the appropriate repository to suggest enhancements to existing commands. Opening an issue before submitting a pull request helps us provide architectural and implementation guidance before you spend too much time on the code.

New to the WP-CLI codebase? Check out issues labeled 'good-first-issue' (https://make.wordpress.org/cli/good-first-issues/) for a place to start. These issues are specially earmarked for new contributors.

Once you've decided to commit the time to seeing your pull request through, please follow our guidelines for creating a pull request (https://make.wordpress.org/cli/handbook/pull-requests/) to make sure it's a pleasant experience. See "Setting up (https://make.wordpress.org/cli/handbook/pull-requests/#setting-up)" for details on making local modifications to WP-CLI. Keep in mind pull requests are expected to have tests (https://make.wordpress.org/cli/handbook/pull-requests/#running-and-writing-tests) covering the scope of the change. Read through our code review guidelines (https://make.wordpress.org/cli/handbook/code-review/) for a better understanding of how your pull request will be evaluated.

## Improving our documentation

Is documentation your strength? Take a look at the currently open documentation issues (https://github.com/search?q=org%3Awp-cli+label%3Ascope%3Adocumentation+is%3Aopen+sort%3Aupdated-desc&type=issues) and see if you can tackle any of those.

There are a couple different types of documentation currently part of WP-CLI:

- Documentation for individual WP-CLI commands (anything underneath developer.wordpress.org/commands (https://developer.wordpress.org/cli/commands/)) is contained in the PHPDoc for each command. This means that to edit the documentation for a command, you will need to edit the file that actually provides the functionality for that command. The web documentation is generated from these files at the time of release, so you may not see your changes until the next release.
- Individual documentation pages (anything under make.wordpress.org/cli/handbook (https://make.wordpress.org/cli/handbook/)) can be edited by contributing to the handbook repository on GitHub (https://github.com/wp-cli/handbook/). You don't necessarily need to navigate the GitHub repo though; any page that is part of this repository will have an 'Edit' link in the top right of the page which will take you to the corresponding file on GitHub.

## Contributing in other ways

Feel free to create an issue (https://github.com/wp-cli/wp-cli/issues/new) with your question, and we'll see if we can find an answer for it.

Alternatively, if you have a WordPress.org account, you may also consider joining the `#cli` channel on the WordPress.org Slack organization (https://make.wordpress.org/chat/).

---

# WordCamp Contributor Day

Welcome to WordCamp Contributor Day! We appreciate you sharing your time with WP-CLI.

We'd love to help you submit at least one pull request, so we put together this guide to make it as straightforward as possible. We're here to support you however we can.

When you submit your pull request, add `Related https://github.com/wp-cli/wp-cli/issues/5985` so we can see all pull requests created during the day. We'll give these some special promotion in the next release notes.

Your table leads for the day are: schlessera (https://github.com/schlessera), BrianHenryIE (https://github.com/BrianHenryIE)

A few seasoned WP-CLI contributors are also helping out: TBD

# Getting Started

If you normally use WP-CLI on your web host or via Brew, you're most likely using the Phar executable. The Phar executable is the "built", single-file version of WP-CLI. It's compiled from a couple dozen repositories in the WP-CLI GitHub organization, so modifying WP-CLI requires working amongst those repositories.

Before you can work on WP-CLI, you'll need to first make sure you have PHP and a functioning MySQL or MariaDB server. Once those prerequisites are met, install the `wp-cli-dev` development environment (https://github.com/wp-cli/wp-cli-dev) to start contributing:

```
git clone https://github.com/wp-cli/wp-cli-dev
cd wp-cli-dev
composer install
```

The `wp-cli-dev` installation process clones all of WP-CLI's repositories to your local machine. After it's complete, you'll be able to make changes in whichever repository you'd like. However, you'll need to fork the repository and add it as a remote in order to push your feature branch.

All WP-CLI pull requests are expected to have tests. See running and writing tests (https://make.wordpress.org/cli/handbook/contributions/pull-requests/#running-and-writing-tests) for a complete introduction.

# Suggested Tickets

To help you be successful during Contributor Day, we curated a list of reasonably approachable and actionable issues. Feel free to comment directly on the issue if you plan to work on it. We don't usually assign issues, so no need to worry about that.

**New contributors**

- "wp config create" generates wrong DB_PASSWORD in wp-config.php when db password has " (https://github.com/wp-cli/config-command/issues/180)
- Import into specific directory/location (https://github.com/wp-cli/media-command/issues/146)
- Add --exclude=, argument to skip files (https://github.com/wp-cli/checksum-command/issues/64)
- Not possible to install translations for en_US (https://github.com/wp-cli/language-command/issues/84)
- Support WP-Stash in 'wp cache type' (https://github.com/wp-cli/cache-command/issues/68)
- Add reason for skipping regeneration (https://github.com/wp-cli/media-command/issues/95)
- wp export is not exporting term meta data (https://github.com/wp-cli/export-command/issues/42)

**Seasoned contributors**

- Add commands to export and import SQLite databases (https://github.com/wp-cli/ideas/issues/188)
- SQLite Compatibility (multiple issues) (https://github.com/wp-cli/.github/issues/94)
- Regenerating a single image size (re-)generates auto-scaled big images & auto-rotated images (https://github.com/wp-cli/media-command/issues/196)
- wp cron event run --due-now doesn't respect doing_cron transient (https://github.com/wp-cli/cron-command/issues/27)
- Update command doesn't escape php_binary path, update fails when path has spaces (https://github.com/wp-cli/wp-cli/issues/5815)
- Associative argument contains double quotes if the command is called with WP_CLI::runcommand() (https://github.com/wp-cli/wp-cli/issues/5541)

You're obviously welcome to work on any other issue you'd like too! Contributor Day can be a good opportunity to discuss trickier issues and brainstorm approaches.

---

# Code Review

Code review is a core part of the WP-CLI project's software development workflow. Code review and pairing on code are necessary to maintain quality output, as well as build cohesive styles in our work. Having a well-understood and cohesive style can make all our work easier to approach and maintain for new as well as established developers. If done right, code review can also be a learning process for all involved. Here's a typical code review workflow:

- What code review is
- How to review code
- How to receive code review

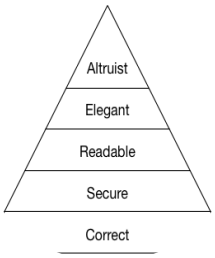
# A typical code review workflow

We're currently using GitHub for everything. Read up on our GitHub workflow (https://make.wordpress.org/cli/handbook/pull-requests/) for details of process, naming, usage of issues and pull requests. A moderately complex issue will typically be addressed across multiple pull requests, each tackling a distinct part of the issue. This makes review simpler because each review stage will be looking at a small code.

- On every commit pushed to a GH branch, our CI automated tests -- linting for all code, unit tests for functional code, and ideally, behavioral and automated acceptance testing -- are run on Travis.
- If the build passes, the PR can be reviewed. If not, the original developer is responsible for getting the build to the point where it passes.
- When the original developer is satisfied with their work, they can request a review from the committers team (https://make.wordpress.org/cli/handbook/committers-credo/) by assigning `@wp-cli/committers` for review.
- Simple pull requests can often be merged by the developer who reviews them. More complex changesets will often require conversations back and forth between reviewer and developer, and should have secondary reviewers.
- GitHub's "Files Changed" tab is a good place to leave inline comments on specific parts of the changeset. More general comments can be left on the pull request's "Conversation" tab.
- The reviewer may suggest changes in the form of a pull request off of the branch being reviewed, or in comments.
- The developer will make changes suggested, discuss the issue for clarity, and may mention the reviewer when they are satisfied with their work.
- If a pull request needs final cleanup before merging or has been abandoned, the reviewer can commit directly to the branch (https://help.github.com/articles/committing-changes-to-a-pull-request-branch-created-from-a-fork/). However, avoid rewriting code without consultation.
- When the reviewer is satisfied with changes, they can either merge or assign the pull request to a second reviewer for merge. The original developer (and ideally the reviewer) should both be available for a couple of days post-merge to address any issues that arise.

# What code review is

A good way to visualize the objectives of code review is this analogy (http://blog.d3in.org/post/111338685456/maslows-pyramid-of-code-review) to Maslow's "Hierarchy of Needs" pyramid. From most basic to the highest level, a reviewer is checking that code is **Correct**, **Secure**, **Readable**, **Elegant**, and **Altruistic**. It's important to keep this sense of priorities: if a change introduces unhandled edge cases, bugs, or security vulnerabilities, those issues need to be addressed before coding style guidelines or beautification practices preferences will matter.




## How to review code

*As a reviewer*, your first job is to get an understanding of what the proposed change does and why it's essential. There's no point in critiquing anything until you understand what it does, why it's necessary, and what decisions went into the way this was built.

Next, look it over for correctness. Are all functions which take parameters and produce output covered by functional unit tests? Do they actually do what they're supposed to? Can you picture an edge case where a function would error unexpectedly or return something other than the expected result? If this command has output, does it render properly in all situations? Are all global flags handled? Does it have unsurprising fallbacks for uncommon situations? Does it handle errors with clear output messages?

If it addresses the business and UX requirements, the next thing to check for is security. Does it follow basic sanitization and escaping practices for all untrusted input? If it interacts with other aspects of the codebase, is it liberal in the inputs it accepts and conservative in its output, making sure to only pass expected values? Think like an attacker. If there's any way a malicious agent could exploit this code, or an unlucky user could trigger a bug that fatals or looks bad, it's your job to find it.

Next, check for readability. Functions, variables, and files should be named clearly according to their meaning. Everything should adhere to the surrounding code style.

> *All code in any code-base should look like a single person typed it, no matter how many people contributed.*
>
> -- *Principles of writing Idiomatic JavaScript (https://github.com/rwaldron/idiomatic.js/)*

As readability is inherently subjective, this requires the ability to look back from the immediate code changes to the bigger picture. Think of someone months down the road trying to trace a given code path through the current changeset. Are there unnecessary steps that could be simplified? Are code comments and inline documentation robust enough to recreate the thought process behind the code?

Finally, check for elegance and overall quality. Code should follow existing and known patterns so that others can understand it at a glance. If a change introduces a chance to refactor surrounding functionality, to abstract and standardize old code into new patterns, suggest those opportunities.

## How to receive code review

*As the person receiving the code review*, your job is to learn from suggestions. Defensiveness, stubbornness, or impatience can prevent you from getting the most out of suggestions. It's OK to explain yourself if you feel the reviewer seemed to misunderstand your intent. Arguing semantics or insisting on the correctness of one approach is almost always a bad habit and distracts from the team process. If something isn't clear to someone familiar with the codebase who's reviewing your work today, it will definitely be unclear to a new developer being on-boarded six months from now.

It's best to respond to the issues addressed by the reviewer as quickly as possible. Keep in mind that the review process requires context switching on the part of the reviewer as well as on your part, and the more immediate that process is, the less disruptive that context switch will be.

## Additional readings

- Glen Sanford: On code review (http://glen.nu/ramblings/oncodereview.php) - "Pending code reviews represent blocked threads of execution[, code review should always be your top priority]"
- The Ten Commandments of Egoless Programming (http://www.techrepublic.com/article/the-ten-commandments-of-egoless-programming/) - "Understand and accept that you will make mistakes. The point is to find them early before they make it into production."

---

# Release Checklist

There are two different release checklists available from within the `wp-cli/wp-cli` (https://github.com/wp-cli/wp-cli) repository:

- Generate a  Regular Release Checklist (https://github.com/wp-cli/wp-cli/issues/new?assignees=schlessera&labels=i%3A+scope%3Adistribution&template=4-REGULAR_RELEASE_CHECKLIST.md&title=Release+checklist+for+v2.x.x) - applies to major (2.x.x) and minor (x.1.x) releases.
- Generate a  Patch Release Checklist (https://github.com/wp-cli/wp-cli/issues/new?assignees=schlessera&labels=i%3A+scope%3Adistribution&template=5-PATCH_RELEASE_CHECKLIST.md&title=Release+checklist+for+v2.x.x) - applies to patch (x.x.1) releases only.

As a first step for initiating a release process, the corresponding checklist should be generated and added to the milestone it relates to.

---

# Default doctor diagnostic checks

Although it's power comes from its ability to be customized (https://make.wordpress.org/cli/handbook/doctor-customize-config/), `wp doctor` includes a number of default diagnostic checks considered to be recommendations for production websites.

Use `wp doctor list` to view these default checks:

| name | description |
|---|---|
| autoload-options-size | Warns when autoloaded options size exceeds threshold of 900 kb. |
| constant-savequeries-falsy | Confirms expected state of the SAVEQUERIES constant. |
| constant-wp-debug-falsy | Confirms expected state of the WP_DEBUG constant. |
| core-update | Errors when new WordPress minor release is available; warns for major release. |
| core-verify-checksums | Verifies WordPress files against published checksums; errors on failure. |
| cron-count | Errors when there's an excess of 50 total cron jobs registered. |
| cron-duplicates | Errors when there's an excess of 10 duplicate cron jobs registered. |
| file-eval | Checks files on the filesystem for regex pattern `eval\(.*base64_decode\(.*`. |
| option-blog-public | Confirms the expected value of the 'blog_public' option. |
| plugin-active-count | Warns when there are greater than 80 plugins activated. |
| plugin-deactivated | Warns when greater than 40% of plugins are deactivated. |
| plugin-update | Warns when there are plugin updates available. |
| theme-update | Warns when there are theme updates available. |

To explain these further:

- Autoloaded options are options that are automatically loaded in every request to WordPress. A size exceeding the recommended threshold could be a symptom of a larger problem.
- Because `SAVEQUERIES` causes WordPress to save a backtrace for every SQL query, which is an expensive operation, using `SAVEQUERIES` in production is discouraged.
- WordPress minor versions are typically security releases that should be applied immediately.

If you create a custom `doctor.yml` config file (https://make.wordpress.org/cli/handbook/doctor-customize-config/), you can use `wp doctor list --config=<file>` to view the diagnostic checks listed in the file.

---

# Force output to a specific locale

WP-CLI always outputs English because it doesn't support localization. But, because WordPress supports localization, you may see non-English output when performing specific commands.

For instance:

```
$ wp theme update --all
מעבר למצב תחזוקה...
מוריד עדכונים מ-https://downloads.wordpress.org/theme/hueman.3.3.25.zip...
Using cached file '/home/xxx/.wp-cli/cache/theme/hueman-3.3.25.zip'...
פתיחת עדכון...
התקנת גרסה חדשה...
הסרת הגרסה הקודמת של התבנית...
התבנית עודכנה בהצלחה.
ביטול מצב תחזוקה...
+--------+-------------+-------------+---------+
| name   | old_version | new_version | status  |
+--------+-------------+-------------+---------+
| hueman | 3.3.24      | 3.3.25      | Updated |
+--------+-------------+-------------+---------+
Success: Updated 1 of 1 themes.
```

To force WordPress to always output English at the command line, you need to filter the active locale.

Given a `force-locale.php` file:

```
<?php
WP_CLI::add_wp_hook( 'pre_option_WPLANG', function() {
    return 'en_US';
});
```

You can force the locale to `en_US` with:

```
wp --require=force-locale.php
```

One nice thing about this approach is that you can easily apply it across multiple WP installs using a config file (https://make.wordpress.org/cli/handbook/config/#config-files).

---

# Write a check for asserting the value of a given option

One of the check types included in `wp doctor` is `Option_Value`, or the ability to assert that a given option is a specific value. The check type is in use by a couple of the default diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-default-checks/), and you can use the `Option_Value` check type in your custom `doctor.yml` configuration file (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-customize-config/).

As an example, here are two checks using `Option_Value`, one which ensures the blog is public and the other which ensures users can't register:

```
option-blog-public:
  check: Option_Value
  options:
    option: blog_public
    value: 1
option-users-can-register:
  check: Option_Value
  options:
    option: users_can_register
    value: 0
```

Run together, you might see:

```
$ wp doctor check --config=option-value.yml --all
+---------------------------+---------+-------------------------------------------------+
| name                      | status  | message                                         |
+---------------------------+---------+-------------------------------------------------+
| option-blog-public        | error   | Site is private but expected to be public.      |
| option-users-can-register | success | Option 'users_can_register' is '0' as expected. |
+---------------------------+---------+-------------------------------------------------+
```

The `Option_Value` check type accepts the following options:

- 'option': Name of the option.
- 'value': Option is expected to be a specific value.

---

# Sharing WP-CLI Packages

By default, WP-CLI places installed packages (https://developer.wordpress.org/cli/commands/package/) in `~/.wp-cli/packages/`, a hidden subdirectory for the user's home directory.

Because the home directory is different for each user, this naturally means each system user will have a separate directory of installed packages. If you have multiple active shell users on a server, and want to share installed WP-CLI packages between them, there are a couple of supported ways to do this.

## `WP_CLI_PACKAGES_DIR` environment variable

To override the directory WP-CLI uses for installed packages, provide a `WP_CLI_PACKAGES_DIR` environment variable. If you wish for multiple users to share the same packages directory, you can simply provide the same `WP_CLI_PACKAGES_DIR` environment variable for each user.

```
vim /etc/environment
export WP_CLI_PACKAGES_DIR=/usr/local/lib/wp-cli-packages
```

Similarly, you can make sure the directory is only writable by a specific user to make packages available to all users, but only installable by the specific user.

It's worth noting the `WP_CLI_PACKAGES_DIR` environment variable *overrides* WP-CLI's default behavior of loading packages installed in the user's home directory. If you want to support both, you'll need to take the second approach.

### Composer project in a shared directory

WP-CLI's installed packages directory is simply a Composer project under the hood. Given this architecture, you can create your own Composer project in an arbitrary directory, and load it into scope using WP-CLI's `--require=<path>` flag.

First, create your Composer project.

```
$ mkdir /usr/local/lib/wp-cli-packages
$ cd /usr/local/lib/wp-cli-packages
$ composer init -n --name=runcommand/wp-cli-packages -s=dev --repository=https://wp-cli.org/package-index/
$ composer require runcommand/hook
Using version dev-master for runcommand/hook
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing runcommand/hook (dev-master 7a7beae)
    Cloning 7a7beae2013eeea243cc44524a7c5c21da11979e

Writing lock file
Generating autoload files
```

Now, once your Composer project has a dependency or two, you can use `wp --require=<path/to/autoload>` (or the equivalent `config.yml` statement) to load the packages into WP-CLI.

---

# How to create a custom plugin:

If you want to create your plugins, WP-CLI has a powerful scaffold command that allows us to generate starter code. In this guide we will see how to generate starter code for a basic plugin.

## Step 1 - Scaffold the plugin files

The following command uses several options to lets us specify the plugin slug, its name, description, author name and uri as well as the plugin uri. You can replace the values passed to the options below to customize the plugin based on your needs.

```
$ wp scaffold plugin wpcli-demo-plugin --plugin_name="WP-CLI Demo Plugin" --plugin_description="This is a wp-cli demo plugin" --
Success: Created test files.
```

The above command generates a new folder called `wpcli-demo-plugin` in the plugins directory, with the following files structure.

```
| - bin/
| - tests/
| - .gitignore
| - .editorconfig
| - .phpcs.xml.dist
| - .travis.yml
| - Gruntfile.js
| - package.json
| - phpunit.xml.dist
| - readme.txt
| - wpcli-demo-plugin.php
```

Unless you use the --skip-tests flag the following files are always generated:

- `phpunit.xml.dist` is the configuration file for PHPUnit.
- `.travis.yml` is the configuration file for Travis CI. Use `--ci=<provider>` to select a different service.
- `bin/install-wp-tests.sh` configures the WordPress test suite and a test database.
- `tests/bootstrap.php` is the file that makes the current plugin active when running the test suite.
- `tests/test-sample.php` is a sample file containing test cases.
- `.phpcs.xml.dist` is a collection of PHP_CodeSniffer rules.

## Step 2 - create a custom post type:

We can now use the scaffold command again to add a custom post type inside our new plugin using the `wp scaffold post-type` command.

```
$ wp scaffold post-type books --label=Book --textdomain=wpcli-demo-plugin --dashicon=dashicons-book-alt --plugin=wpcli-demo-plug
Success: Created '/wpcli-demo-plugin/post-types/books.php'.
```

## Step 3 - Write code inside the main file:

The main plugin file `wpcli-demo-plugin.php` is the starting point that we can use to write our plugin logic.

Inside the main plugin file lets now reference the new post type we just created.

Open in your favourite text editor the file `wpcli-demo-plugin.php` and under the line saying "your code starts here" add the following:

```
\\Your code starts here.
require('post-types/books.php');
```

## Step 4 - Activate the plugin

You can now use two wp-cli commands to check the list of plugins and activate your newly created plugin. `wp plugin list` and `wp plugin activate`. The first command lists all plugins installed while the second activates a given plugin.

```
$ wp plugin list
+-------------------+----------+-----------+---------+
| name              | status   | update    | version |
+-------------------+----------+-----------+---------+
| akismet           | inactive | available | 4.1.5   |
| hello             | inactive | none      | 1.7.2   |
| wpcli-demo-plugin | inactive | none      | 0.1.0   |
+-------------------+----------+-----------+---------+
```

From the list above we can see that our plugin wpcli-demo-plugin is inactive. Let's enable it using the other command.

```
$ wp plugin activate wpcli-demo-plugin
Plugin 'wpcli-demo-plugin' activated.
Success: Activated 1 of 1 plugins.
```

Our plugin is now active. We can visit the WordPress admin panel and start using our books custom post type.

---

# Philosophy

WP-CLI is the command-line interface for WordPress. It provides commands for actions that can be performed in the WordPress backend (e.g. `wp theme activate` and `wp user create`), as well as commands for actions that don't have an equivalent web UI (e.g. `wp cron event run`, `wp search-replace`, and `wp scaffold child-theme`).

WP-CLI's guiding principle is to be the fastest and canonical way to manage WordPress:

- Speed is inherently present in the simplicity of the command-line interface, and the ability to chain multiple commands together into a more complex operation.
- When functionality ends up in WP-CLI, it should be good enough to become the defacto solution to the problem.
- "Managing WordPress" is the ever-evolving problem space in which we operate. WP-CLI should always focus on making *existing management patterns* more efficient.

This page contains a list of guidelines that should inform decisions related to scope, command organization and behavior:

## Good commands are simple, interact with WordPress, and solve one unique problem.

When a command implements existing WordPress functionality (e.g. `wp export`), it should mirror and re-use existing WordPress behavior as much as possible. Doing so ensures it meets the user's assumed expectations of behavior.

When a command adds functionality not present in the WordPress admin (e.g. `wp search-replace`), it should solve one unique problem and interact with WordPress in some way. Good commands are born of true user need; for a new proposed WP-CLI command, there probably should be existing prior art in the ecosystem.

If a command provides useful functionality, but it doesn't have anything to do with WordPress, it doesn't belong in WP-CLI. Similarly, an idea for a command is sometimes better implemented as a change to documentation instead.

## Consistency makes for an enjoyable user experience.

Whenever possible, commands should emulate one another with consistent usage behavior, argument names, and documentation.

## Decisions, not options.

WordPress' philosophy states (https://wordpress.org/about/philosophy/):

> *[The end user is considered first when making implementation decisions]. A great example of this consideration is software options. Every time you give a user an option, you are asking them to make a decision. When a user doesn't care or understand the option this ultimately leads to frustration. As developers we sometimes feel that providing options for everything is a good thing, you can never have too many choices, right? Ultimately these choices end up being technical ones, choices that the average end user has no interest in. It's our duty as developers to make smart design decisions and avoid putting the weight of technical choices on our end users.*

The same principle applies to WP-CLI. In every case, we should first try to improve the default behavior. Options should be added sparingly, only when a solution is required and no other solutions are available.

As a rule of thumb, once a command has a half-dozen options or more, it becomes difficult to understand how the command will operate under each condition.

## Composability is always a good idea.

One of the most useful ideas in UNIX was that of pipelines (http://en.wikipedia.org/wiki/Pipeline_%28Unix%29).

WP-CLI commands should be composable, i.e. the output from one command should be easily pipe-able to another command.

A corollary of this is that commands should be *orthogonal*, which means that there should be no overlapping functionality between commands. (`wp plugin install --activate` is an exception).

See Shell Friends (https://make.wordpress.org/cli/handbook/references/shell-friends/) for specific examples of how this philosophy is applied.

# Readability trumps number of keystrokes.

Most of the commands in WP-CLI will be used non-interactively, so make the parameter names self-documenting.

# Don't assume anything.

Bundled commands MUST work on any given WordPress install (provided that it's a new enough version), no matter how it's configured and no matter how much data it has.

As a corollary, bundled commands should not assume the presence of any plugin or theme.

---

# How to install WordPress

Downloading and installing WordPress using WP-CLI is straight forward. It takes four steps. First, you will need to download WordPress using the `wp core download` command.

## Step 1 - Download WordPress

The syntax of the command to download WordPress is the following: `wp core download [--path=<path>] [--locale=<locale>] [--version=<version>] [--skip-content] [--force]`

```
$ wp core download --path=wpdemo.test --locale=it_IT
Creating directory '/wpdemo.test/'.
Downloading WordPress 5.4.1 (it_IT)...
md5 hash verified: 3fa03967b47cdfbf263462d451cdcdb8
Success: WordPress downloaded.
```

The command above creates a `wpdemo.test/` folder inside your current working directory and downloads the latest WordPress version. You can replace the `--path=wpdemo.test` with your desired folder name and the `--locale=it_IT` with your desired locale. You can omit the `--locale` option and, that will download by default WordPress in American English using the locale en_US.

## Step 2 - Generate a config file

In this step, we will generate a config file and set up the database credentials for our installation. The basic syntax of the command is the following: `wp config create --dbname=<dbname> --dbuser=<dbuser> [--dbpass=<dbpass>]`

```
$ wp config create --dbname=your_db_name_here --dbuser=your_db_user_here --prompt=dbpass
1/10 [--dbpass=<dbpass>]: type_your_password
Success: Generated 'wp-config.php' file.
```

The command above generates the `wp-config.php` file and adds to it the database credentials that you passed. Make sure to replace `your_db_name_here` with the name you want to assign to the database, replace `your_db_user_here` with your database user and type the database password when prompted with the following: `1/10 [--dbpass=<dbpass>]:`

## Step 3 - Create the database

In this step, we are going to create the database based on the information we passed to the `wp-config.php` file in step 2.

```
$ wp db create
Success: Database created.
```

Now we are ready to move to the final step where we install WordPress.

## Step 4 - Install WordPress

To install WordPress now, we need to run one last command.

```
$ wp core install --url=wpclidemo.dev --title="WP-CLI" --admin_user=wpcli --admin_password=wpcli --admin_email=info@wp-cli.org
Success: WordPress installed successfully.
```

Remember to replace the values passed to each of the following options with your details:

- `--url=wpclidemo.dev` replace wpclidemo.dev with your website url,
- `--title="WP-CLI"` replace WP-CLI with the name you want to assign to the website,

- `--admin_user=wpcli` replace wpcli with the username you want to assign to the website administrator
- `--admin_password=wpcli` replace wpcli with the password you want to use to access the WordPress administrator panel.

Congratulations! You have successfully installed WordPress using WP-CLI.

---

# Write a check for verifying contents of WordPress files

One of the check types included in `wp doctor` is `File_Contents`, or the ability to check all or a selection of WordPress files for a given regex pattern. The check type is in use by a couple of the default diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-default-checks/), and you can use the `File_Contents` check type in your custom `doctor.yml` configuration file (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-customize-config/).

The `File_Contents` check type is the most efficient way to check WordPress files because it only iterates the filesystem once, regardless of how many file checks you've registered.

As an example, here are two checks using `File_Contents`, one which ensures sessions aren't used by plugins and the other which ensures `$_SERVER['SERVER_NAME']` isn't used in `wp-config.php`:

```
file-sessions:
  check: File_Contents
  options:
    regex: .*(session_start|\$_SESSION).*
    only_wp_content: true
file-server-name-wp-config:
  check: File_Contents
  options:
    regex: define\(.+WP_(HOME|SITEURL).+\$_SERVER.+SERVER_NAME
    path: wp-config.php
```

Run together, you might see:

```
$ wp doctor check --config=file-contents.yml --all
+----------------------------+---------+-----------------------------------------------------------------------------------------------+
| name                       | status  | message                                                                                       |
+----------------------------+---------+-----------------------------------------------------------------------------------------------+
| file-sessions              | success | All 'php' files passed check for '.*(session_start|\$_SESSION).*'.                            |
| file-server-name-wp-config | success | All 'php' files passed check for 'define\(.+WP_(HOME|SITEURL).+\$_SERVER.+SERVER_NAME'. |
+----------------------------+---------+-----------------------------------------------------------------------------------------------+
```

The `File_Contents` check type accepts the following options:

- 'regex': Regex pattern to check against each file's contents.
- 'extension'. File extension to check. Defaults to 'php'. Separate multiple file extensions with a '|'.
- 'path'. Check a specific file path. Value should be relative to `ABSPATH` (e.g. 'wp-content' or 'wp-config.php').
- 'only_wp_content': Only check the wp-content directory.

---

# Write a check for asserting the value of a given constant

One of the check types included in `wp doctor` is `Constant_Definition`, or the ability to assert that a given constant is either defined, a specific value, or falsy. The check type is in use by a couple of the default diagnostic checks (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-default-checks/), and you can use the `Constant_Definition` check type in your custom `doctor.yml` configuration file (https://make.wordpress.org/cli/handbook/guides/doctor/doctor-customize-config/). As an example, here are two checks using `Constant_Definition`, one which ensures `SAVEQUERIES` isn't defined and the other which ensures `DISALLOW_FILE_MODS` is true:

```
constant-savequeries-falsy:
  check: Constant_Definition
  options:
    constant: SAVEQUERIES
    falsy: true
constant-disallow-file-mods-true:
  check: Constant_Definition
  options:
    constant: DISALLOW_FILE_MODS
    value: true
```

Run together, you might see:

```
$ wp doctor check --config=constant-definition.yml --all
+---------------------------------+---------+-------------------------------------------------+
| name                            | status  | message                                         |
+---------------------------------+---------+-------------------------------------------------+
| constant-savequeries-falsy      | success | Constant 'SAVEQUERIES' is undefined.            |
| constant-disallow-file-mods-true | success | Constant 'DISALLOW_FILE_MODS' is defined 'true'. |
+---------------------------------+---------+-------------------------------------------------+
```

The `Constant_Definition` check type accepts the following options:

- 'constant': Name of the constant.
- 'defined': Constant is expected to be defined.
- 'value': Constant is expected to be a specific value.
- 'falsy': Constant is expected to be undefined or falsy.

---

# Tools

The following is a list of projects that integrate with WP-CLI in some form. For installable WP-CLI packages, please see the package index (https://wp-cli.org/package-index/).

# Plugins

The following table is an alphabetical list of known commands defined in WordPress plugins:

| Command | WordPress plugin |
|---|---|
| acf | Advanced Custom Fields wp-cli (https://github.com/hoppinger/advanced-custom-fields-wpcli) |
| amt | Add-Meta-Tags command line interface (http://www.codetrax.org/projects/wp-add-meta-tags/wiki/Command_Line_Interface) |
| any-ipsum | Any Ipsum (https://wordpress.org/plugins/any-ipsum/) |
| backup | BackUpWordPress (https://wordpress.org/plugins/backupwordpress/) |
| backwpup | BackWPup (https://wordpress.org/plugins/backwpup/) |
| blog-dupe | blog-duplicator (https://github.com/trepmal/blog-duplicator) |
| camptix | CampTix (https://github.com/Automattic/camptix) |
| check-content | CheckContent (https://github.com/mattclegg/wp-cli_check-content) |
| co-authors-plus | Co-Authors Plus (https://github.com/automattic/co-authors-plus) |
| composer | Composer (http://wordpress.org/plugins/composer) |
| config | WP-CFM (https://wordpress.org/plugins/wp-cfm/) |
| crontrol | WP-Crontrol (http://wordpress.org/plugins/wp-crontrol/) |
| csv | Advanced CSV Importer (https://wordpress.org/plugins/advanced-csv-importer/) |
| deploy | wp-deploy-flow (https://github.com/demental/wp-deploy-flow) |
| developer | Developer (http://wordpress.org/plugins/developer/) |
| edd | Easy Digital Downloads (https://easydigitaldownloads.com/docs/wp-cli-commands/) |
| elasticpress | ElasticPress (https://github.com/10up/ElasticPress) |
| github-updater | GitHub Updater (https://github.com/afragen/github-updater) |
| google-sitemap | Google Sitemap Generator CLI (https://github.com/wp-cli/google-sitemap-generator-cli) |
| image-gen | Image Gen (https://github.com/trepmal/image-gen/) |
| import_sidebar | Widget Import Export (https://github.com/cftp/widget-importer) |
| itelic | Licensing for Exchange (https://github.com/iron-bound-designs/exchange-addon-licensing) |
| jekyll-export | WordPress to Jekyll Exporter (https://github.com/benbalter/wordpress-to-jekyll-exporter/) |
| jetpack | Jetpack by WordPress.com (http://wordpress.org/plugins/jetpack/) |
| liveblog | Liveblog (http://wordpress.org/plugins/liveblog/) |

| Command | WordPress plugin |
|---|---|
| mainwp | MainWP (https://mainwp.com/) |
| migrate | WP Migrate DB CLI Interface (https://github.com/duncanjbrown/WP-CLI-Migrate/) |
| migratedb | WP Migrate DB Pro (https://deliciousbrains.com/wp-migrate-db-pro/doc/cli-addon/) |
| more-plugin-info | More Plugin Info (http://wordpress.org/plugins/more-plugin-info/) |
| multi-device | Multi Device Switcher (https://wordpress.org/plugins/multi-device-switcher/) |
| nginx | Nginx Cache Controller (http://wordpress.org/plugins/nginx-champuru/) |
| optimize | WP-Optimize (https://getwpo.com/) |
| orderweight | Order Weight for WooCommerce (https://wordpress.org/plugins/woo-order-weight/) |
| p2-by-email | P2 By Email (https://github.com/humanmade/P2-By-Email) |
| p2-resolved-posts | P2 Resolved Posts (http://wordpress.org/plugins/p2-resolved-posts/) |
| p2p | Posts 2 Posts (http://wordpress.org/plugins/posts-to-posts/) |
| pdf-light-viewer | PDF Light Viewer (https://github.com/antongorodezkiy/pdf-light-viewer) |
| phpcompat | PHP Compatibility Checker (https://wordpress.org/plugins/php-compatibility-checker/) |
| post-gen | Post Gen (https://github.com/trepmal/post-gen/) |
| quick-mail | Quick Mail (https://wordpress.org/plugins/quick-mail/) |
| rest-api-toolbox | REST API Toolbox (https://github.com/petenelson/wp-rest-api-toolbox) |
| redis-cache | Redis Object Cache (https://wordpress.org/plugins/redis-cache/) |
| revisions | wp-revisions-cli (https://github.com/trepmal/wp-revisions-cli/) |
| revision-strike | Revision Strike (https://wordpress.org/plugins/revision-strike/) |
| safe-redirect-manager | Safe Redirect Manager (https://github.com/10up/safe-redirect-manager) |
| scheduled-unsticky | Scheduled Unsticky (http://wordpress.org/plugins/scheduled-unsticky/) |
| simple-history | Simple History (https://simple-history.com/) |
| site duplicate | MultiSite Clone Duplicator (http://wordpress.org/plugins/multisite-clone-duplicator/) |
| stream | WP Stream (http://wordpress.org/plugins/stream/) |
| super-cache | WP Super Cache CLI (https://github.com/wp-cli/wp-super-cache-cli) |
| supportpress | SupportFlow (https://github.com/SupportFlow/supportflow) |
| themecheck | wp-cli-themecheck (https://github.com/anhskohbo/wp-cli-themecheck) |
| thinkup | Thinkup Import WP CLI Commands (https://github.com/taras/wp-cli-thinkup-import) |
| total-cache | W3 Total Cache (https://wordpress.org/plugins/w3-total-cache/) |
| updraftplus | UpdraftPlus (https://updraftplus.com/) |
| Unsplash | Import images from Unsplash into your Media Library (https://github.com/A5hleyRich/wp-cli-unsplash-command) |
| wp2static | Generate & deploy a static HTML version of your site (https://github.com/elementor/wp2static) |
| yoast | Reindex Indexables on Yoast (https://developer.yoast.com/features/wp-cli/reindex-indexables/) |

If you implement a WP-CLI command in one of your plugins, please list it here.

# Wrappers

- Chef WP-CLI (https://github.com/francescolaffi/chef-wp-cli) - WP provisioning using Chef
- node-wp-cli (https://github.com/gtg092x/node-wp-cli) - Node JS wrapper for WP-CLI
- Puppet WP-CLI (https://github.com/rmccue/puppet-wp) - WP provisioning using Puppet
- wpcli gem (https://github.com/hasanen/wpcli) - run WP-CLI commands from Ruby
- wplib (https://github.com/szepeviktor/wplib) - shell scripts for managing multiple sites
- wpcli helpers (https://github.com/charleslouis/wp_cli_helpers) - a bundle of time saver aliases and functions for the shell
- cPanel-wp-management (https://github.com/MarioKnight/cPanel-wp-management) - shell scripts designed to loop through all accounts on cPanel servers
- Plesk WordPress Toolkit (https://www.plesk.com/wp-toolkit/) - WordPress installations management UI backed by WP-CLI

# Editor plugins

- Vim - https://github.com/dsawardekar/wordpress.vim (https://github.com/dsawardekar/wordpress.vim)
- Netbeans - https://github.com/junichi11/netbeans-wordpress-plugin (https://github.com/junichi11/netbeans-wordpress-plugin)

# Vagrant boxes

- Trellis (https://github.com/roots/trellis)
- Varying Vagrant Vagrants (https://github.com/Varying-Vagrant-Vagrants/VVV)

# Misc

- AnsiPress (https://github.com/AnsiPress/AnsiPress) - Setup NGINX/WordPress Stack
- Bedrock (https://github.com/roots/bedrock) - WP base stack
- EasyEngine (https://github.com/rtCamp/easyengine/) - Hosting control panel

- WordPress Development Flow (https://github.com/cityindex/wordpress-development-flow) - retired
- WP-CLI GUI (http://wpcligui.com/) - a GUI to assist with installing WP using WP-CLI
- WP-API Oauth (https://github.com/WP-API/OAuth1) - WP REST API - OAuth 1.0a Server

---