

Running Commands Remotely



- [Using an SSH connection](#)
 - [Scheme](#)
 - [docker](#)
 - [docker-compose](#)
 - [vagrant](#)
 - [Aliases](#)
 - [Running custom commands remotely](#)
 - [Running from Windows Powershell](#)
 - [Making WP-CLI accessible on a remote server](#)
 - [Copy WP-CLI binary to \\$HOME / bin](#)
 - [Specify the \\$PATH in \\$HOME / .ssh/ environment](#)
 - [Remote non-interactive shell resolves aliases and runs wp as alias with php](#)
 - [Configure the remote non-interactive shell to resolve aliases](#)
 - [Alias wp to php which runs the WP-CLI binary](#)
 - [Using before_ssh hook on client machine](#)
-

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](#) 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](#) 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](#).

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
```

Running from Windows Powershell

Windows Powershell has a special meaning for the `@` symbol, meaning it cannot be used without quoting or escaping. To run commands remotely, you need to use one of the following notation.

```
$ wp `@dev core version
$ wp "@dev" core version
```

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](#)).

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

Some web hosts are configured very restrictively:

- They do not allow you to execute your own shell scripts, 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](#) 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 `wp-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
```

📄 #3

📄 cocobook-sync 📄	2026-10-28 10:58:49 UTC
📄 cocobook-sync 📄	2026-10-28 11:46:47 UTC