

How to WP-CLI

- [How to WP-CLI](#)
- [Figure out why WordPress is slow](#)
- [How to create a custom plugin](#)
- [How to install WordPress](#)
- [How to put the site in maintenance mode](#)
- [How to start the webserver](#)
- [Identify a Plugin or Theme Conflict](#)
- [Plugin Integration Tests](#)

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](#)
- [How to put the website in maintenance mode](#)
- [How-to start the webserver](#)
- [How to create a custom plugin](#)
- [Figure out why WordPress is slow](#)

Figure out why WordPress is slow

`wp profile` is a WP-CLI command to help you quickly identify what's slow with WordPress. It's designed to work alongside Xdebug and New Relic because it's easy to deploy to any server that has WP-CLI. With `wp profile`, you gain quick visibility into key performance metrics (execution time, query count, cache hit/miss ratio, etc.) to guide further debugging.

Installation

First, install the `wp profile` package if you haven't already:

```
$ wp package install wp-cli/profile-command
```

Step 1 - Profile the WordPress load process

Dealing with a slow WordPress install you've never worked with before? Run `wp profile stage` to see metrics for each stage of the WordPress load process. Include the `--url=<url>` argument to mock the request as a specific URL.

```
$ wp profile stage --url=runcommand.io
```

stage	time	query_time	query_count	cache_ratio	cache_hits	cache_misses	hook_time	hook_count	request_time	request_count
bootstrap	0.7597s	0.0052s	14	93.21%	357	26	0.3328s	2717	0s	0
main_query	0.0131s	0.0004s	3	94.29%	33	2	0.0065s	78	0s	0

template	0.7041s	0.0192s	147	92.16%	2350	200	
0.6982s	6130	0s	0				
+-----+-----+-----+-----+-----+-----+-----+-----							
+-----+-----+-----+-----+-----+-----+-----+-----							
total (3)	1.477s	0.0248s	164	93.22%	2740	228	
1.0375s	8925	0s	0				
+-----+-----+-----+-----+-----+-----+-----+-----							
+-----+-----+-----+-----+-----+-----+-----+-----							

When WordPress handles a request from a browser, it's essentially executing as one long PHP script. `wp profile stage` breaks the script into three stages:

- **bootstrap** is where WordPress is setting itself up, loading plugins and the main theme, and firing the `init` hook.
- **main_query** is how WordPress transforms the request (e.g. `/2016/10/21/moms-birthday/`) into the primary `WP_Query`.
- **template** is where WordPress determines which theme template to render based on the main query, and renders it.

Step 2 - Dive into a specific stage

In the example from above, bootstrap seems a bit slow, so let's dive into it further. Run `wp profile stage bootstrap` to dive into higher fidelity mode of a given stage. Use the `--spotlight` flag to filter out the zero-ish results.

```
$ wp profile stage bootstrap --url=runcommand.io --spotlight
```

+-----+-----+-----+-----+-----+-----+-----+-----							
+-----+-----+-----+-----+-----+-----+-----+-----							
hook		callback_count	time	query_time	query_count	cache_ratio	
cache_hits	cache_misses	request_time	request_count				
+-----+-----+-----+-----+-----+-----+-----+-----							
+-----+-----+-----+-----+-----+-----+-----+-----							
muplugins_loaded:before			0.1644s	0.0017s	1		
40%	2	3	0s	0			
muplugins_loaded	2		0.0005s	0s	0		
50%	1	1	0s	0			
plugins_loaded:before			0.1771s	0.0008s	6		
77.63%	59	17	0s	0			
plugins_loaded	14		0.0887s	0s	0		
100%	104	0	0s	0			
after_setup_theme:before			0.043s	0s	0		

100%	26	0	0s	0	
init		82	0.1569s	0.0018s	7
96.88%	155	5	0s	0	
wp_loaded:after			0.027s	0s	0
	0	0	0s	0	
+-----+-----+-----+-----+-----+					
+-----+-----+-----+-----+-----+					
total (7)		98	0.6575s	0.0043s	14
77.42%	347	26	0s	0	
+-----+-----+-----+-----+-----+					
+-----+-----+-----+-----+-----+					

Each stage is further segmented by `wp profile` based on its primary actions. For the bootstrap stage, the primary actions include `muplugins_loaded`, `plugins_loaded`, and `init`. You can also see some intermediate actions like `plugins_loaded:before` and `wp_loaded:after`. These intermediate actions correspond to script execution before (or after) actual WordPress hooks. They're pseudo hooks in a sense.

Step 3 - Profile specific hooks

When you've found a specific hook you'd like to assess, run `wp profile hook <hook>`. Include the `--fields=<fields>` argument to limit output to certain fields.

```
$ wp profile hook plugins_loaded --url=runcommand.io --fields=callback,time,location
```

+-----+-----+		
+-----+-----+		
callback	time	
location		
+-----+-----+		
+-----+-----+		
wp_maybe_load_widgets()	0.0046s	wp-
includes/functions.php:3501		
wp_maybe_load_embeds()	0.0003s	wp-
includes/embed.php:162		
VaultPress_Hotfixes->protect_jetpack_402_from_oembed_xss()	0s	
vaultpress/class.vaultpress-hotfixes.php:124		
_wp_customize_include()	0s	wp-
includes/theme.php:2052		
EasyRecipePlus->pluginsLoaded()	0.0013s	
easyrecipeplus/lib/EasyRecipePlus.php:125		

Gamajo\GenesisHeaderNav\genesis_header_nav_i18n()	0.0007s	genesis-header-
nav/genesis-header-nav.php:61		
DS_Public_Post_Preview::init()	0.0001s	public-post-
preview/public-post-preview.php:52		
wpseo_load_textdomain()	0.0004s	wordpress-seo-
premium/wp-seo-main.php:222		
load_yoast_notifications()	0.0016s	wordpress-seo-
premium/wp-seo-main.php:381		
wpseo_init()	0.0329s	wordpress-seo-
premium/wp-seo-main.php:240		
wpseo_premium_init()	0.0019s	wordpress-seo-
premium/wp-seo-premium.php:79		
wpseo_frontend_init()	0.0007s	wordpress-seo-
premium/wp-seo-main.php:274		
Black_Studio_TinyMCE_Plugin->load_compatibility()	0.0016s	black-studio-tinymce-
widget/black-studio-tinymce-widget.php:206		
Jetpack::load_modules()	0.0564s	
jetpack/class.jetpack.php:1672		
+-----+-----+		
+-----+		
total (14)	0.1026s	
+-----+-----+		
+-----+		

There you have it! We've discovered that `wpseo_init()` and `Jetpack::load_modules()` are collectively contributing ~100ms to every page load.

With `wp profile`, discovering why WordPress is slow becomes the easy part.

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 let 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" --plugin_author=wp-cli --  
plugin_author_uri="https://wp-cli.org" --plugin_uri="https://plugins.wp-cli.org/demo-plugin"  
Success: Created test files.
```

The above command generates a new folder called `wpcli-demo-plugin` in the plugins directory, with files such as:

- `wpcli-demo-plugin.php`
- `readme.txt`
- `package.json`
- `.gitignore`
- `.editorconfig`

Unless you use the `--skip-tests` flag, test-related files are generated too. These include:

- `phpunit.xml.dist` is the configuration file for PHPUnit.
- A CI configuration file. Use `--ci=<provider>` to select a supported CI provider.
- `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-plugin
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.

How to install WordPress

Downloading and installing WordPress using WP-CLI is straightforward. 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`.

Next, navigate into the newly created folder (`cd wpdemo.test`) before running the subsequent commands.

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.

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.
```

How to start the webserver

You can use the command `wp server` to launch 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>. You can open the browser and visit the page to access your WordPress installation.

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
```

Plugin Integration Tests

This guide demonstrates how to run integration tests both in continuous integration 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 in continuous integration

The `wp scaffold plugin-tests` command generates a CircleCI configuration by default. Use `--ci=<provider>` to generate a configuration for a different supported provider, such as GitHub Actions:

```
wp scaffold plugin-tests my-plugin --ci=github
```

All you need to do to enable this is copy and then edit the following files from the [WP-CLI sample plugin](#):

- `.circleci/config.yml`, `phpunit.xml.dist` and `.phpcs.xml.dist` files
- `tests` folder

See the docs for an [explanation of what each file does](#). 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 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](#))

So, let's get started:

1. Install PHPUnit

The version of PHPUnit to install depends on both PHP and WordPress versions. See: [PHPUnit Compatibility and WordPress Versions – Make WordPress Core](#)

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 `.circleci/config.yml` file by default. Use `--ci=<provider>` to generate a configuration for a different supported CI provider.

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 the [common issues guide](#) 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](#).

Second, the `bin/install-wp-tests.sh` script doesn't work directly in Windows. Windows 10 [introduced a Windows Subsystem for Linux](#) 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 writable directory, and then manually create `wp-tests-config.php`.