

Writing Behat Tests for WP-CLI



- [Introduction](#)
 - [What are Behat Tests?](#)
 - [Why Behat Tests Matter](#)
 - [Getting Started](#)
- [Writing Your First Test](#)
 - [Basic Test Structure](#)
 - [Your First Test](#)
 - [Common Step Definitions](#)
 - [Given \(Setup\)](#)
 - [When \(Actions\)](#)
 - [Then \(Assertions\)](#)
 - [Running Tests](#)
 - [Setting Up Your Test Environment](#)
 - [Running the Test Suite](#)
 - [Discovering Available Steps](#)
- [Advanced Usage](#)
 - [Writing Effective Tests](#)
 - [Test One Thing at a Time](#)
 - [Use Descriptive Scenario Names](#)
 - [Test Both Success and Failure](#)
 - [Using Variables](#)
 - [Testing Tables and Structured Output](#)
 - [Testing Table Output](#)
 - [Testing JSON Output](#)
 - [Testing CSV Output](#)
 - [Testing Files and Directories](#)

- [Background Sections](#)
 - [Testing Error Messages](#)
 - [Using Scenario Outlines](#)
 - [Writing Custom Steps](#)
 - [Testing with Different PHP and WordPress Versions](#)
 - [Database Credentials](#)
 - [Best Practices](#)
 - [Do's](#)
 - [Don'ts](#)
 - [Scaffolding Tests for New Packages](#)
 - [Troubleshooting](#)
 - [Tests Fail to Connect to Database](#)
 - [Tests Pass Locally But Fail in CI](#)
 - [Command Not Found Errors](#)
 - [Slow Tests](#)
 - [Additional Resources](#)
 - [Examples from WP-CLI Packages](#)
-

WP-CLI uses [Behat](#) for functional testing. This guide will help you understand how to write and run Behat tests for WP-CLI commands and packages.

Introduction

What are Behat Tests?

Behat is a behavior-driven development (BDD) framework for PHP. In WP-CLI, Behat tests are used as functional tests that execute the entire command and verify it works as expected. Unlike unit tests that test individual functions in isolation, functional tests ensure commands work correctly from end to end.

Why Behat Tests Matter

Stability between releases is an important contract WP-CLI makes with its users. Behat tests help ensure:

- Commands work as documented
- Changes don't break existing functionality
- Edge cases are handled properly
- The entire command execution flow is validated

Tests are required for most pull requests. If it's a new feature, tests are needed. If it's fixing a bug, tests are needed to prevent regression.

Getting Started

Every WP-CLI repository with commands has a `features/` directory containing one or more Gherkin-formatted `*.feature` files.

Writing Your First Test

Basic Test Structure

Here's a simple example:

```
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
      """
```

This test follows the Gherkin syntax with three key components:

- **Feature:** - Documents the scope of the file. This should describe the overall functionality being tested.
- **Scenario:** - Describes a specific test case. Each feature file can have multiple scenarios.
- **Given** - Sets up the initial environment for the test (preconditions).
- **When** - Causes an event to occur (the action being tested).
- **Then** - Asserts what's expected after the event is complete (verification).

In a 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.com'.

Your First Test

Let's write a test for a hypothetical command that displays a greeting:

1. Create a file `features/greeting.feature`:

```
Feature: Test greeting command
```

```
  Scenario: Display a greeting
```

```
    Given an empty directory
```

```
    When I run `wp greeting hello`
```

```
    Then STDOUT should contain:
```

```
      """
```

```
      Hello
```

```
      """
```

```
    And the return code should be 0
```

This test:

1. Starts with an empty directory
2. Runs your command
3. Verifies the output contains "Hello"
4. Verifies the command succeeded (exit code 0)

Common Step Definitions

WP-CLI provides many pre-built step definitions. Here are the most commonly used:

Given (Setup)

- `Given an empty directory` - Creates a clean working directory
- `Given a WP install` - Installs a fresh WordPress instance
- `Given a WP multisite install` - Installs WordPress in multisite mode
- `Given a database` - Creates an empty database

When (Actions)

- `When I run 'wp command'` - Execute a command and expect success
- `When I try 'wp command'` - Execute a command (success or failure)
- `When I launch in the background 'wp command'` - Run command in background

Then (Assertions)

- `Then STDOUT should be:` - Exact match of output
- `Then STDOUT should contain:` - Output contains text
- `Then STDOUT should not contain:` - Output doesn't contain text
- `Then the return code should be 0` - Command succeeded
- `Then the return code should be 1` - Command failed
- `Then STDOUT should be empty` - No output produced

See the [Behat Steps reference](#) for a complete list of available steps.

Running Tests

Setting Up Your Test Environment

Before running tests, you need to set up a test database. The `composer behat` script will automatically detect database availability and fall back to SQLite if there is no running database, making it easy to get started without any setup.

You have two options:

Option 1: SQLite (Recommended for simplicity)

The easiest way to run tests is with SQLite, which requires no database setup:

```
WP_CLI_TEST_DBTYPE=sqlite composer behat
```

Or simply run `composer behat` and let it automatically use SQLite if no database is configured.

Option 2: MySQL/MariaDB

If you prefer to use MySQL or MariaDB, run the setup script to create the test database:

```
composer prepare-tests
```

This creates a MySQL user `wp_cli_test` with password `password1` that has full privileges on the `wp_cli_test` database.

Note: MySQL 8.0 changed the default authentication plugin. If you have connection issues, see [this blog post](#) for solutions.

Running the Test Suite

Run all tests:

```
composer behat
```

Run tests for a specific feature file:

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

Run a specific scenario (by line number):

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

Run with verbose output:

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

Re-run only failed tests:

```
composer behat-rerun
```

Discovering Available Steps

To see all available step definitions:

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

This is helpful when writing tests to find the right step for your needs.

Advanced Usage

Writing Effective Tests

Test One Thing at a Time

Each scenario should test a single piece of functionality:

```
# Good - tests one specific behavior
Scenario: Delete a post
  Given a WP install
  And I run `wp post create --post_title='Test' --porcelain`
  And save STDOUT as {POST_ID}

  When I run `wp post delete {POST_ID}`
  Then STDOUT should contain:
    """
    Success: Trashed post
    """

# Bad - tests multiple unrelated behaviors
Scenario: Create, update, and delete a post
  # Too much in one test...
```

Use Descriptive Scenario Names

Make it clear what's being tested:

```
# Good
Scenario: Fail gracefully when post doesn't exist

# Bad
Scenario: Test delete
```

Test Both Success and Failure

Don't just test the happy path:

```
Scenario: Successfully create a post
  Given a WP install
  When I run `wp post create --post_title='Test'`
  Then the return code should be 0

Scenario: Fail when required argument is missing
  Given a WP install
  When I try `wp post create`
```

```
Then the return code should be 1
```

```
And STDERR should contain:
```

```
"""
```

```
Error
```

```
"""
```

Using Variables

Store command output in variables for later use:

```
Scenario: Create and then fetch a post
```

```
Given a WP install
```

```
When I run `wp post create --post_title='Test Post' --porcelain`
```

```
Then save STDOUT as {POST_ID}
```

```
When I run `wp post get {POST_ID} --field=title`
```

```
Then STDOUT should be:
```

```
"""
```

```
Test Post
```

```
"""
```

Testing Tables and Structured Output

Testing Table Output

```
Scenario: List posts in table format
```

```
Given a WP install
```

```
And I run `wp post create --post_title='First'`
```

```
And I run `wp post create --post_title='Second'`
```

```
When I run `wp post list --fields=ID,post_title`
```

```
Then STDOUT should be a table containing rows:
```

```
| ID | post_title |
```

```
| 1  | First      |
```

```
| 2  | Second     |
```

Testing JSON Output

Scenario: List posts in JSON format

Given a WP install

And I run `wp post create --post\_title='Test' --porcelain`

When I run `wp post list --format=json`

Then STDOUT should be JSON containing:

```
"""
```

```
[{"post_title": "Test"}]
```

```
"""
```

Testing CSV Output

Scenario: Export posts as CSV

Given a WP install

When I run `wp post list --format=csv`

Then STDOUT should be CSV containing:

ID	post_title	post_status
----	------------	-------------

1	Test	publish
---	------	---------

Testing Files and Directories

Scenario: Create a plugin file

Given a WP install

When I run `wp scaffold plugin my-plugin`

Then the my-plugin/my-plugin.php file should exist

And the my-plugin/my-plugin.php file should contain:

```
"""
```

```
Plugin Name: My Plugin
```

```
"""
```

Background Sections

Use `Background:` to run common setup for all scenarios:

Feature: Post management

Background:

Given a WP install

And I run `wp post create --post\_title='Test'`

Scenario: Update post title

When I run `wp post update 1 --post\_title='Updated'`

Then STDOUT should contain:

"""

Success

"""

Scenario: Delete post

When I run `wp post delete 1`

Then STDOUT should contain:

"""

Success

"""

Testing Error Messages

Use `When I try` instead of `When I run` to allow commands to fail:

Scenario: Error when plugin doesn't exist

Given a WP install

When I try `wp plugin activate non-existent-plugin`

Then the return code should be 1

And STDERR should contain:

"""

Error: The 'non-existent-plugin' plugin could not be found.

"""

Using Scenario Outlines

Test multiple inputs with a single scenario:

Scenario Outline: Create posts with different titles

Given a WP install

When I run `wp post create --post\_title='<title>`

Then STDOUT should contain:

```
"""
```

```
Success
```

```
"""
```

Examples:

```
| title          |
```

```
| Simple Title   |
```

```
| Title's Quotes |
```

```
| UTF-8: ☐☐☐   |
```

Writing Custom Steps

For advanced use cases, you might need custom step definitions. These are defined in PHP in your package's test bootstrap file.

Example custom step definition:

```
/**
 * @Given a custom configuration file
 */
public function aCustomConfigurationFile() {
    $config = <<<EOT
custom_setting: value
another_setting: 123
EOT;
    $this->proc( "echo '{$config}' > wp-cli.yml" )->run_check();
}
```

Custom steps should:

- Have descriptive docblock annotations
- Follow Gherkin conventions (Given/When/Then)
- Be reusable across multiple tests
- Use existing WP-CLI test framework methods

For most packages, the built-in step definitions in [wp-cli/wp-cli-tests](#) are sufficient. Only create custom steps when necessary.

Testing with Different PHP and WordPress Versions

WP-CLI tests run against multiple PHP and WordPress versions in CI. Your tests should work across these versions.

Keep in mind:

- WordPress behavior may differ between versions
- Use version-appropriate expectations
- Avoid testing WordPress core bugs (test your command, not WordPress)

Database Credentials

By default, MySQL/MariaDB tests use:

- Database: `wp_cli_test`
- Username: `wp_cli_test`
- Password: `password1`
- Host: `localhost`

Override these with environment variables from [wp-cli/wp-cli-tests](#):

- `WP_CLI_TEST_DBNAME`
- `WP_CLI_TEST_DBUSER`
- `WP_CLI_TEST_DBPASS`
- `WP_CLI_TEST_DBHOST`

Alternatively, use SQLite by setting `WP_CLI_TEST_DBTYPE=sqlite` to avoid database configuration entirely.

Best Practices

Do's

- **Test command behavior, not implementation** - Tests should verify what the command does, not how it does it
- **Make tests independent** - Each test should run successfully in isolation
- **Use meaningful test data** - Use realistic post titles, user names, etc.
- **Test edge cases** - Empty strings, special characters, large numbers
- **Keep scenarios focused** - One logical test per scenario
- **Use Given steps for setup** - Don't mix setup and testing
- **Test error conditions** - Verify your command fails appropriately

Don'ts

- **Don't test WordPress core functionality** - Test your command, not WordPress
- **Don't make tests depend on each other** - Each scenario should stand alone
- **Don't use hard-coded IDs** - Use variables or porcelain output
- **Don't skip error testing** - Failed commands are important to test
- **Don't test implementation details** - Focus on user-facing behavior
- **Don't forget to test help text** - Include scenarios for `wp help your-command`

Scaffolding Tests for New Packages

When creating a new WP-CLI package, use `wp scaffold package-tests` to generate the testing infrastructure:

```
wp scaffold package my-package
cd my-package
wp scaffold package-tests .
```

This creates:

- `.github/workflows/testing.yml` - GitHub Actions workflow
- `features/` - Directory for your feature files
- `features/load-wp-cli.feature` - Basic test to verify WP-CLI loads
- Other testing infrastructure files

See the [scaffold-package-command documentation](#) for more details.

Troubleshooting

Tests Fail to Connect to Database

If you see database connection errors:

1. Try using SQLite instead: `WP_CLI_TEST_DBTYPE=sqlite composer behat`
2. If using MySQL, verify it's running: `mysql -u wp_cli_test -ppassword1 wp_cli_test`
3. Check your MySQL version (MySQL 8.0 has authentication changes)
4. Verify the database was created: `composer prepare-tests`

Tests Pass Locally But Fail in CI

Common causes:

- Different PHP versions (check your CI matrix)
- Different WordPress versions (check your test matrix)
- Timing issues (some commands need longer to complete)
- Database collation differences

Command Not Found Errors

If your command isn't found during tests:

1. Verify your `composer.json` properly declares the command
2. Check that `composer install` ran successfully
3. Ensure your command class is properly autoloaded

Slow Tests

If tests are slow:

- Minimize database operations
- Use `Given an empty directory` instead of `Given a WP install` when possible
- Avoid unnecessary plugin installations
- Use `--porcelain` to reduce output parsing

Additional Resources

- [Behat Steps Reference](#) - Complete list of available steps
- [WP-CLI Test Framework](#) - The underlying test framework
- [Behat Documentation](#) - Official Behat documentation
- [Writing Functional Tests for WP-CLI Packages](#) - Detailed tutorial
- [Pull Request Guidelines](#) - Contributing guidelines

Examples from WP-CLI Packages

Real-world examples from WP-CLI core packages:

- [Core Command Tests](#) - Examples of testing installation and updates
- [Search-Replace Command Tests](#) - Testing database operations
- [Scaffold Command Tests](#) - Testing file generation

Study these to see how different types of commands are tested in practice.

Now you're ready to write comprehensive Behat tests for your WP-CLI commands! Remember: good tests make WP-CLI more reliable for everyone.

📄 #4

📄 cocobook-sync 📄 2026-10-28 11:01:10 UTC

📄 cocobook-sync 📄 2026-10-29 01:42:31 UTC