Chapter 18. PHP Command Line

Developers come to PHP from all sorts of backgrounds and with various levels of experience in software development. Regardless of whether you are a new computer science graduate, a seasoned developer, or someone from a noncoding field looking to learn a new skill, the forgiving nature of the language makes it easy to get started. That being said, the largest stumbling block for these noncoder beginners may be PHP's command-line interface.

Noncoder beginners are likely to be comfortable with using a graphical user interface and navigating with a mouse and a graphical display. Give the same user a command-line terminal, and they might struggle with or be intimidated by the interface.

As a backend language, PHP is frequently manipulated at the command line. This potentially makes it an intimidating language for developers not accustomed to text-based interfaces. Fortunately, PHP-based command-line applications are relatively straightforward to build and immensely powerful to use.

An application might expose a command palette similar to its default RESTful interface, thus making interactions from a terminal similar to those over a browser or through an API. Yet another application might bury its administrative tooling in the CLI, protecting less technical end users from accidentally damaging the application.

One of the most popular PHP applications in the market today is WordPress, the open source blogging and web platform. Most users interact with the platform through its graphical web interface, but the WordPress community also maintains a rich command-line interface for the platform: WP-CLI. This tool allows a user to manage everything already exposed by the graphical tool but through a scriptable, text-based terminal interface. It also exposes commands for managing user roles, system configuration, the state of the database, and even the system cache. None of these capabilities exist within the stock web interface!

Any developer building a PHP application today can and should understand the capabilities of the command line, both in terms of what you can do with PHP itself and how your application can expose its functionality through the same interface. A truly rich web application will at some point live on a server that might not expose any sort of graphical interface, so being able to control your application from the terminal is not just a power move—it's a necessity.

The following recipes demystify the intricacies of argument parsing, managing input and output, and even leveraging extensions to build full applications that run in the console.

18.1 Parsing Program Arguments

Problem

You want a user to pass an argument when they invoke your script so it can be parsed from within the application.

Solution

Use the `$argc` integer and the `$argv` array to retrieve the value of an argument directly in the script. For example:

```
<?php
if ($argc !== 2) {
    die('Invalid number of arguments.');
}

$name = htmlspecialchars($argv[1]);

echo "Hello, {$name}!" . PHP_EOL;
```

Discussion

Assuming you named the script in the Solution example *script.php*, it would be invoked in a terminal session with the following command:

```
% php script.php World
```

Internally, the `$argc` variable contains a count of the number of parameters passed to PHP when executing the script. In the Solution example, there are exactly two parameters:

- The name of the script itself (*script.php*)
- Whatever string value you passed after the name of the script

---

NOTE

Both `$argc` and `$argv` can be disabled at runtime by setting the `register_argc_argv` flag to `false` in your *php.ini* file. If enabled, these parameters will contain either the arguments passed to a script or information about a GET request forwarded from a web server.

The first argument will *always* be the name of the script or file being executed. All other arguments are delimited by spaces beyond that. Should you need to pass a compound argument (like a string with spaces), wrap that argument in double quotes. For example:

```
% php script.php "dear reader"
```

More complicated implementations might leverage PHP's `getopt()` function rather than manipulating the argument variables directly. This function will parse both short and long options and pass their contents into arrays your application can then leverage.

Short options are each single characters represented at the command line with a single dash—for example, `-v`. Each option could either merely be present (as in a flag) or be followed by data (as in an option).

Long options are prefixed with double dashes but otherwise act the same way as their short relatives. You can assume either or both styles of options are present and use them however you want in your application.

NOTE

Often, a command-line application will provide both a long option and a single-character shortcut for the same thing. For example, `-v` and `--verbose` are frequently used to control the level of output of a script. With `getopt()`, you can easily have both, but PHP won't link them together. If you support two different methods for providing the same option value or flag, you'll need to reconcile them within your script manually.

The `getopt()` function takes three parameters and returns an array representing the options the PHP interpreter has parsed:

- The first argument is a single string in which each character represents a short option or flag.
- The second argument is an array of strings, and each string is a long option name.
- The final argument, *which is passed by reference*, is an integer representing the index in `$argv` where parsing has stopped when PHP encounters a non-option.

Both short and long options also accept modifiers. If you pass an option by itself, PHP will not accept a value for that option but will treat it as a flag. If you append a colon to an option, PHP will *require* a value. If you append two colons, PHP will treat the value as optional.

As an illustration, Table 18-1 lists out various ways both short and long options can leverage these additional elements.

Table 18-1. PHP `getopt()` arguments

Argument	Argument type	Description
`a`	Short option	A single flag with no value: `-a`
`b:`	Short option	A single flag with a required value: `-b value`
`c::`	Short option	A single flag with an optional value: `-c value` or just `-c`

Argument	Argument type	Description
`ab:c`	Short option	Composite three flags where `a` and `c` have no value but `b` requires a value: `-a -b value -c`
`verbose`	Long option	Option string with no value: `--verbose`
`name:`	Long option	Option string with a required value: `--name Alice`
`output::`	Long option	Option string with an optional value: `--output file.txt` or just `--output`

To illustrate the utility of option parsing, define a program as in Example 18-1 that exposes both short and long options but also leverages free-form (non-option) input after the flags. The following script will expect the following:

- A flag to control whether output should be capitalized (`-c`)
- A username (`--name`)
- Some extra, arbitrary text after the options

Example 18-1. Direct illustration of `getopt()` with multiple options

```
<?php
$optionIndex = 0;

$options = getopt('c', ['name:'], $optionIndex);
                                  ❶


$firstLine = "Hello, {$options['name']}!" . PHP_EOL;
                                  ❷


$rest = implode(' ', array_slice($argv, $optionIndex));
                                  ❸
```

```
if (array_key_exists('c', $options)) {
    $firstLine = strtoupper($firstLine);
    $rest = strtoupper($rest);
}

echo $firstLine;
echo $rest . PHP_EOL;
```

❶ Use `getopt()` to define both the short and long options your script expects. The third, optional parameter is passed by reference and will be overwritten by the index at which the interpreter runs out of options to parse.

❷ Options with values are easy to extract from the resultant associative array.

❸ The resultant index from `getopt()` can be used to quickly extract any additional data from the command by pulling unparsed values out of the `$argv` array.

❹ Options without values will still set a key in the associative array, but the value will be a Boolean `false`. Check that the key exists, but don't rely on its value because of the counterintuitive nature of the result.

Assuming you name the script defined by Example 18-1 *getopt.php*, you can expect to see a result like the following:

```
% php getopt.php -c --name Reader This is fun
HELLO, READER!
THIS IS FUN
%
```

See Also

Documentation on `$argc`, `$argv`, and the `getopt()` function.

18.2 Reading Interactive User Input

Problem

You want to prompt the user for input and read their response into a variable.

Solution

Read data from the standard input stream by using the `STDIN` file handle constant. For example:

```
echo 'Enter your name: ';

$name = trim(fgets(STDIN, 1024));

echo "Welcome, {$name}!" . PHP_EOL;
```

Discussion

The standard input stream makes it easy for you to read any data that comes in with a request. Reading data directly from the stream in a program using `fgets()` will pause the execution of your program until the end user provides that input to you.

The Solution example leverages the shorthand constant `STDIN` to reference the input stream. You could just as easily use the stream's fully qualified name (along with an explicit `fopen()`), as demonstrated in Example 18-2.

Example 18-2. Reading user input from `stdin`

```
echo 'Enter your name: ';

$name = trim(fgets(fopen('php://stdin', 'r'), 1024));

echo "Welcome, {$name}!" . PHP_EOL;
```

NOTE

The special `STDIN` and `STDOUT` shorthand names are only accessible in an application. If using the interactive terminal REPL as in Recipe 18.5, these constants will not be defined nor will they be accessible.

An alternative approach is to use the GNU Readline extension with PHP, which may or may not be available in your installation. This extension wraps much of the manual work to prompt for, retrieve, and trim user input. The entire Solution example could be rewritten as shown in Example 18-3.

Example 18-3. Reading input from the GNU Readline extension

```
$name = readline('Enter your name: ');

echo "Welcome, {$name}!" . PHP_EOL;
```

Additional functions provided by the Readline extension, like `readline_add_history()`, allow for efficient manipulation of the system's command-line history. If the extension is available, it's a powerful way to work with user input.

NOTE

Some distributions of PHP, like those for Windows, will come with Readline support enabled by default. In other situations, you might need to compile PHP explicitly to include this support. For more on native PHP extensions, review Recipe 15.4.

See Also

Further discussion of standard input in Recipe 11.2.

18.3 Colorizing Console Output

Problem

You want to display text in the console in different colors.

Solution

Use properly escaped console color codes. For example, print the string `Happy Independence Day` in blue text on a red background as follows:

```
echo "\e[0;34;41mHappy Independence Day!\e[0m" . PHP_EC
```

Discussion

Unix-like terminals support ANSI escape sequences that grant programs fine-grained control over things like cursor location and font styling. In particular, you can define the color used by the terminal for all following text with this escape sequence:

```
\e[{foreground};{background}m
```

Foreground colors come in two variants—regular and bold (determined by an extra Boolean flag in the color definition). Background colors lack this differentiation. All of the colors are identified by these codes in Table 18-2.

Table 18-2. ANSI color codes

Color	Normal foreground	Bright foreground	Background
Black	0;30	1;30	40
Red	0;31	1;31	41
Green	0;32	1;32	42
Yellow	0;33	1;33	43
Blue	0;34	1;34	44
Magenta	0;35	1;35	45
Cyan	0;36	1;36	46

Color	Normal foreground	Bright foreground	Background
White	`0;37` (really light gray)	`1;37`	`47`

To reset the terminal colors back to normal, use a simple `0` in place of any color definitions. The code `\e[0m` will reset all attributes.

See Also

Wikipedia coverage of ANSI escape codes.

18.4 Creating a Command-Line Application with Symfony Console

Problem

You want to create a full command-line application without manually writing all of the argument parsing and handling code yourself.

Solution

Use the Symfony Console component to define your application and its commands. Example 18-4, for example, defines a Symfony command for greeting a user by name with `Hello world` at the console. Example 18-5 then uses that command object to create an application that greets the user within the terminal.

Example 18-4. A basic hello world command

```
namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
```

```
#[AsCommand(name: 'app:hello-world')]
class HelloWorldCommand extends Command
{
    protected static $defaultDescription = 'Greets the

    // ...
    protected function configure(): void
    {
        $this
            ->setHelp('This command greets a user...')
            ->addArgument('name', InputArgument::REQUIF
    }

    protected function execute(InputInterface $input, C
    {
        $output->writeln("Hello, {$input->getArgument('
        return Command::SUCCESS;
    }
}
```

Example 18-5. Creating the actual console application

```
#!/usr/bin/env php
<?php
// application.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\Console\Application;

$application = new Application();

$application->add(new App\Command\HelloWorldCommand());

$application->run();
```

Then run the command as follows:

```
% ./application.php app:hello-world User
```

Discussion

The Symfony project provides a robust collection of reusable components for PHP. It acts as a framework to simplify and greatly increase the speed of development for web applications as well. It's remarkably well documented, powerful, and best of all, free and entirely open source.

NOTE

The open source Laravel framework, the data modules of which were covered in Recipe 16.9, is itself a meta package of individual Symfony components. Its own Artisan console tool is built atop the Symfony Console component. It provides rich command-line control over Laravel projects, their configuration, and even their runtime environments.

Like any other PHP extension, Symfony components are installed via Composer.[1] The Console component itself can be installed as follows:

```
% composer require symfony/console
```

The preceding `require` command will update your project's *composer.json* file to include the Console component, and it also installs this component (and its dependencies) in your project's *vendor/* directory.

NOTE

If your project is not already using Composer, installing any package will create a new *composer.json* file for you automatically. You should take time to update it to autoload any classes or files your project requires so everything works together seamlessly. For more on Composer, extensions, and autoloading, review Chapter 15.

Once you have the library installed, you can start leveraging it immediately. Business logic for various commands can live elsewhere within your application (e.g., behind a RESTful API) but can also be imported into and exposed via the command-line interface.

By default, every class that descends from `Command` gives you the ability to work with user-provided arguments and to display content back to the

terminal. Options and arguments are created with the `addArgument()` and `addOption()` methods on the class and can be manipulated within its `configure()` method directly.

Output is highly flexible. You can print content directly to the screen with any of the methods of the `ConsoleOutputInterface` class listed in Table 18-3.

Table 18-3. Symfony console output methods

Method	Description
`writeln()`	Writes a single line to the console. Equivalent to using `echo` on some text followed by an explicit `PHP_EOL` newline.
`write()`	Writes text to the console without appending a newline character.
`section()`	Creates a new output section that can be atomically controlled as if it were an independent output buffer.
`overwrite()`	Only valid on a section—overwrites content in a section with the given content.
`clear()`	Only valid on a section—clears all contents of a section.

In addition to the text methods introduced in Table 18-3, Symfony Console empowers you to create dynamic tables in the terminal. Every `Table` instance is bound to an output interface and can have as many rows, columns, and separators as you need. Example 18-6 demonstrates how a simple table can be built and populated with content from an array before itself being rendered to the console.

Example 18-6. Rendering tables in the console with Symfony

```
// ...

#[AsCommand(name: 'app:book')]
```

```
class BookCommand extends Command
{
    public function execute(InputInterface $input, Outp
    {
        $table = new Table($output);
        $table
            ->setHeaders(['ISBN', 'Title', 'Author'])
            ->setRows([
                [
                    '978-1-940111-61-2',
                    'Security Principles for PHP Applic
                    'Eric Mann'
                ],
                ['978-1-098-12132-7', 'PHP Cookbook', '
            ])
        ;
        $table->render();

        return Command::SUCCESS;
    }
}
```

Symfony Console automatically parses the content passed into a `Table` object and renders the table for you complete with grid lines. The preceding command produces the following output in the console:

```
+-------------------+---------------------------------
| ISBN              | Title
+-------------------+---------------------------------
| 978-1-940111-61-2 | Security Principles for PHP Appli
| 978-1-098-12132-7 | PHP Cookbook
+-------------------+---------------------------------
```

Further modules within the component aid in the control and rendering of dynamic progress bars and interactive user prompts and questions.

The Console component even aids in coloring terminal output directly. Unlike the complicated ANSI escape sequences discussed in Recipe 18.3, Console allows you to use named tags and styles directly to control content.

WARNING

At the time of this writing, the Console component disables output coloring on Windows systems by default. There are various, free terminal applications (like Cmder) available for Windows as alternatives to the standard terminal that do support output coloring.

The terminal is an incredibly powerful interface for your users. Symfony Console makes it easy to target this interface within your application without resorting to hand-parsing arguments or manually crafting rich output.

See Also

Full documentation of Symfony's Console component.

18.5 Using PHP's Native Read-Eval-Print-Loop

Problem

You want to test some PHP logic without creating a full application to house it.

Solution

Leverage PHP's interactive shell as follows:

```
% php -a
```

Discussion

The PHP interactive shell provides a read-eval-print loop (REPL) that effectively tests single statements in PHP and, where possible, prints directly to the terminal. Within the shell, you can define functions and classes or even directly execute imperative code without creating a script file on disk.

This shell is an efficient way to test a particular line of code or piece of logic outside the context of a full application.

The interactive shell also enables full tab-completion for all PHP functions or variables as well as any functions or variables that you have defined while the shell session is running. Merely type the first few characters of an otherwise long name, press Tab, and the shell will automatically complete the name for you. If there are multiple possible completions, press the Tab key twice to see a list of all possibilities.

You can control two particular settings for the shell in your *php.ini* configuration file: `cli.pager` allows for an external program to handle output rather than displaying directly to the console, and `cli.prompt` allows you to control the default `php >` prompt.

For example, you can replace the prompt itself by passing an arbitrary string to `#cli.prompt` within the shell session as follows:

```
% php -a
                                ❶

php > #cli.prompt=repl ~>
                                ❷

repl ~>
                                ❸
```

❶ The initial invocation of PHP launches the interactive shell.

❷ Setting the `cli.prompt` configuration directly will override the default until this session closes.

❸ Once you've overridden the default prompt, you will see your new version until you exit.

WARNING

Backticks can be used to execute arbitrary PHP code within the prompt itself. Some examples in the PHP documentation use this method to prepend the current time to the prompt. However, this might not work consistently between systems and could introduce unnecessary instability when executing your PHP code.

You can even colorize your output by using the ANSI escape sequences defined in Table 18-2. This presents a more pleasant interface in many situations and empowers you to provide additional information if desired. The CLI prompt itself introduces four additional escape sequences, as defined in Table 18-4.

Table 18-4. CLI prompt escape sequences

Sequence	Description
`\e`	Adds colors to the prompt by using the ANSI codes introduced in Recipe 18.3.
`\v`	Prints the PHP version.
`\b`	Indicates which logical block contains the interpreter. By default, this will be `php` but could be `/*` to represent a multiline comment.
`\>`	Represents the prompt character, which is `>` by default. When the interpreter is inside another unterminated block or string, this will change to indicate where the shell is. Possible characters are `' " { ( >`.

By using both ANSI escape sequences to define colors and the special sequences defined for the prompt itself, you can define a prompt that exposes the version of PHP and the location of the interpreter and that uses a friendly foreground color as follows:

```
php > #cli.prompt=\e[032m\v \e[031m\b \e[34m\> \e[0m
```

The preceding setting results in the display in Figure 18-1.

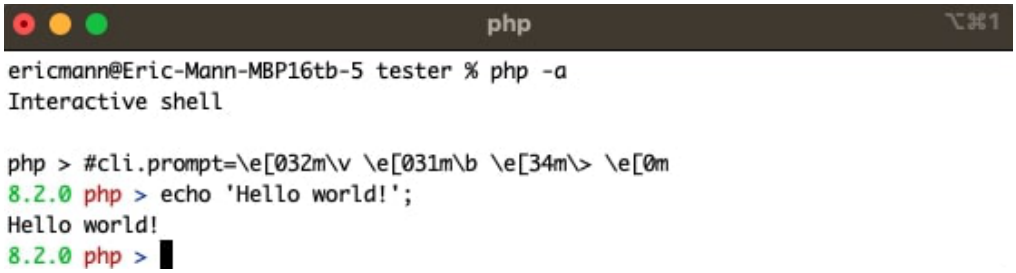

Figure 18-1. The PHP console updated with colorization

WARNING

Not every console will support colorization via ANSI control sequences. If this is a pattern you intend to use, take care to test your sequences thoroughly prior to asking anyone else to use the system. While a properly rendered console is attractive and easy to use, unrendered escape sequences can make a console nearly impossible to work with.

See Also

Documentation on PHP's interactive command shell.

[1] For more on Composer, review Recipe 15.3.