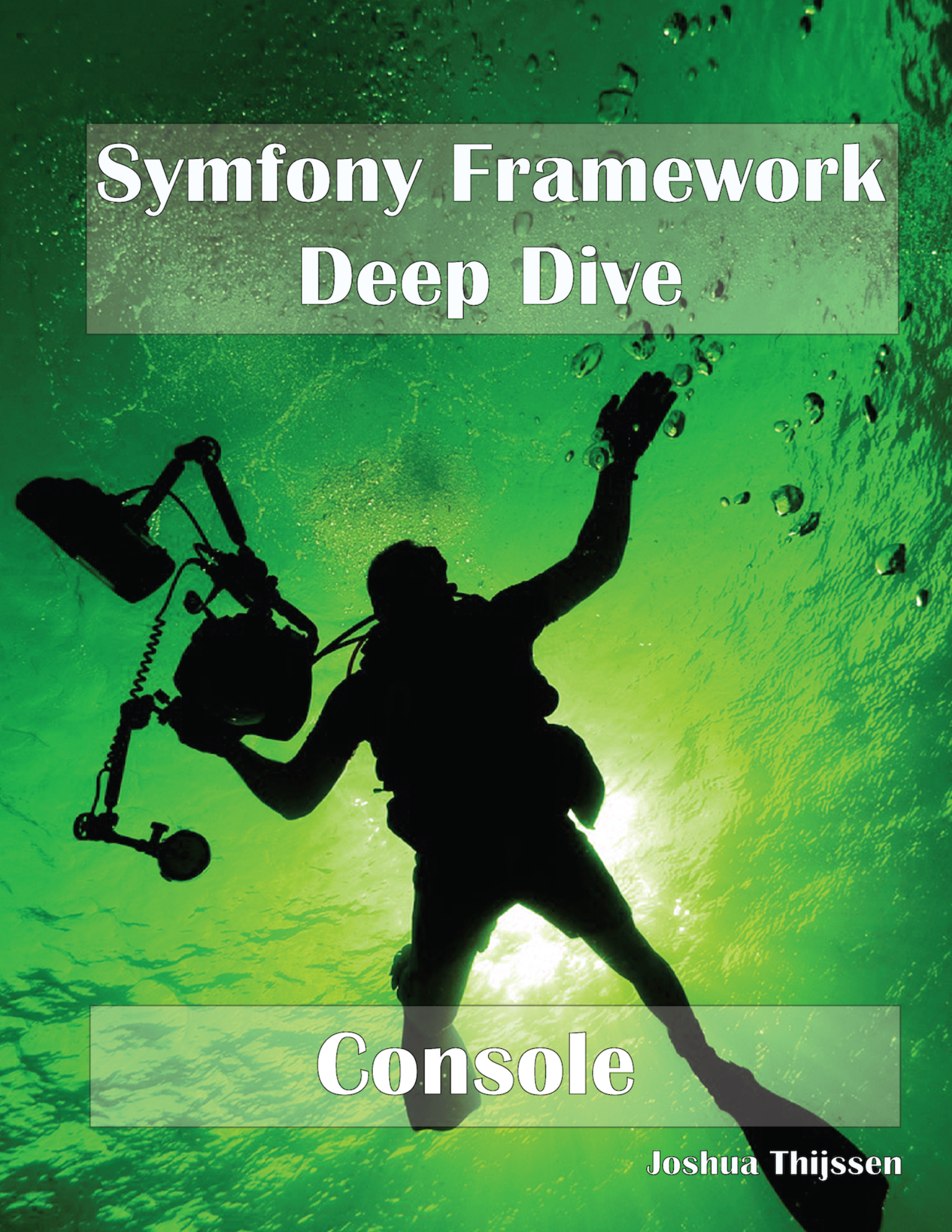
Symfony Framework Deep Dive



Console

Joshua Thijssen

Symfony Framework Deepdive - Console

A deepdive into the Symfony console components

Joshua Thijssen

This book is for sale at http://leanpub.com/symfonyframeworkdeepdive-console

This version was published on 2015-06-15

Tweet This Book!

Please help Joshua Thijssen by spreading the word about this book on Twitter!

The suggested hashtag for this book is #symfonyrainbow.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#symfonyrainbow

Also By Joshua Thijssen

Symfony Framework Deepdive - Security

Contents

Introduction

Sample

This is a sample of the upcoming “Console” edition of the Symfony Deepdive series. It contains a few chapters taken from the different sections that will be found in the full release. If you enjoy reading this sample, you surely will like the full release. For more information about this book, and any other upcoming and previous releases, please take a look at (http://symfony-rainbow.com)[http://symfony-rainbow.com].

About the Symfony rainbow series

The Symfony rainbow series is a collection of books based around the different components that make up the Symfony framework. Every book will concentrate on a single component or a set of smaller components. The books will describe their inner workings in detail and give you some tips and tricks on how to use these components.

[1] http://pixabay.com/en/sea-ocean-water-light-diver-79606/

[2] http://creativecommons.org/publicdomain/zero/1.0/deed.en

Introduction

Console applications are quite common nowadays in the PHP environment, but this wasn't always the case. PHP itself was written explicitly for serving web server requests and returning dynamic output as a response. This request/ response design of PHP scripts means, as scripts, they have short life cycles. which also means it wasn't really necessary for PHP to have a solid memory management system. But for these short lived scripts, this wasn't really a problem. Any memory that would leak, would be cleared up as soon as the script ended with its response. Even though there is still disagreement in the developer community that PHP is solid enough to be able to handle long-running scripts, such as system daemons, PHP has improved considerably over the years, making it a much more robust system, capable for short term and and even long-running console applications.

Despite some of the disagreement, everyone agrees, writing short lived console applications, like utilities or message queue workers, is something PHP is perfectly capable of doing these days.

But, writing console applications differs from writing web applications. Instead of requests and responses, we have to deal with input/output, argument and option parsing, terminal capabilities, signal handling and much more.

In this book we will look at the different components that Symfony provides us for handling console applications. Not every component described here is purely for the console only. Some of them can be used in your web application too, but their origins or main purpose are bound to the console nevertheless. Since most of these components are not really big (not as big as say the security or the form component), I've bundled four of them inside this book under "Console".

Just like any other component of Symfony, they are not bound to the Symfony2 framework. It's possible, and highly recommended, to use these components in your standalone applications or custom frameworks.

Furthermore, we will take a look at how console commands are integrated into the Symfony2 framework through the `app/console` application.

Symfony versioning

This book is written in the pre 2.7 release of Symfony2. However, all of the book is valid for Symfony 2.7. If there are any big differences between 2.6 and 2.7, they will be emphasized as clearly as possible.

About those who will read this book

This book is written for both beginners and experienced developers, who are curious about the internals of the Symfony2 components that deal with the console, processes and filesystems. Even though everything is explained in great detail, some basic knowledge about using the Symfony framework is assumed. You don't need to be an expert (you will be after reading this book), but you should at least have a basic understanding and experience with writing a (simple) Symfony framework application or some experience with the standalone components.

This book is really about getting to know more about how the components function in-depth, to help you use them in your daily work as a developer. It provides valuable information about understanding the components and allows you to use them more efficiently.

Note that this book is not meant to be light reading material. In fact, it's probably anything but light. There is lots of information about how the components work internally, and almost all of it is explained in great detail, which can make for dry reading. It can be used as a reference, should you run into programming challenges within your Symfony application and need to remember exactly how things work or need a push in the right direction.

A lot of information in this book references to the actual Symfony code. It is wise to have this code close to you, when we are talking about explicit classes or methods. You should be able to follow the text better with the code as a reference.

About those I’d like to thank

This book could not have be written without the help of others. First of all, I’d like to thank *Fabien Potencier* and all the developers, who contributed to the Symfony2 framework. Be it as a core developer, a bundle developer, or even “just” being a Symfony2 framework user.

The members of the *Dutch Web Alliance* and the *PHPNL slack channel*, for providing feedback.

A special thanks to *Scott Molinari* from `skooppa.com` for proofreading and fixing countless spelling errors.

And last but not least, anyone who is reading this book.

The console component

The process component

ProcessUtils

The `ProcessUtils` class is a small utility class, which holds a few static helper methods used by the Process component. It is not meant to be instantiated, which is why the constructor is set to private. All methods below are called statically.

escapeArgument()

The `escapeArgument` wraps the regular `escapeshellarg` from PHP in order to fix some bugs[3][4] on Windows systems. These bugs are very old, but doesn't seem to be fixed correctly yet. It fixes these issues by escaping quotes and backslashes manually. Also, when environment variables are found, they are escaped too, so they don't get automatically expanded.

This manual escaping only occurs on the Windows platform. Other platforms will automatically use the default `escapeshellarg` from PHP, as they do not have these bugs.

validateInput()

The `validateInput` method checks if the `$input` variable is either `null`, a resource, or a string. There is still a deprecated feature, which also allows objects to be cast to a string through the `__toString` magic method. This method is used for checking if a variable is usable as input for a process instance (it has to conform to any of the types mentioned before).

This method has a strange signature, as it also needs a `$caller` argument. This argument is just the name of the method that is calling the `validateInput()` method, so it can display a better error message, when the input is not confirming the standards.

Most likely, you will see that this method is called something like:

[3] https://bugs.php.net/bug.php?id=43784

[4] https://bugs.php.net/bug.php?id=49446

```
$this->input = ProcessUtils::validateInput(
    sprintf('%s::%s', __CLASS__, __FUNCTION__),
    $input
);
```

So when the input fails the validation, it will throw an `InvalidArgumentException` with the calling class and method name. It's only for displaying as a message within the exception, and serves no other purpose.

isSurroundedBy()

Returns true when `$arg` starts and ends with `$char`. This is used to check if a word is surrounded by `%`, to check if it's an environment variable like `%TEMP%`, etc.

The finder component

Directory structure

Adapter
: This directory contains the adapters and interfaces, which will actually do the work of finding files. Their purpose is to return an iterator based on the given constraints and sorting preferences you add to the finder.

Comparator
: The comparator directory holds comparator classes, which are used to compare certain elements. For instance, the `DateComparator` can be used to check if a certain date is equal, earlier or after another date. These comparators can be used to create (and/or test) numbers or ranges.

Exception
: This directory contains exceptions, which can be thrown within the finder component.

Expression
: This class contains expressions, like regular expressions and globbing patterns, which can be used within your finder constraints.

Iterator
: The `Iterator` directory contains iterators, which are used for the PHPAdapter to filter files based on the given constraints.

Shell
: This directory contains, in essence, a mini `process` component. The `Shell` class is able to guess which operating system it is running on, and it has a `testCommand` method, which executes a given command and returns a boolean, if the command succeeded or not. The `Command` class allows you to create commands through a builder like interface, in a simplistic way, and ultimately calls `execute` to run the created command.

Tests
: The `Tests` directory contains PHPUnit tests for this component.

In addition to the directories, there are a few classes defined in the main directory. They are listed here with some small explanation on what they will do. We will go into further detail about these classes in later.

Finder.php
: The `Finder` class is the entry point of the component. Its interface allows you to easily create filters and sorting constraints to find files. It even allows you to change the way files are actually found, by changing the underlying applied adapter as well.

Glob.php
: The `Glob` class defines a `toRegex` method, which allows you to use globbing patterns within text, instead of on files (which is what the standard `glob()` PHP function does). It is a port of a perl version, and it doesn't seem to be used directly by the component itself. However, if you need the actual functionality, it's here for use.

SplFileInfo.php
: This class is an extension of the original `\SplFileInfo` class, where it adds relative path information that is not present in the standard `\SplFileInfo` class. It also provides a `getContents()` method, which will return the contents of a file as a string. If the file could not be read, it will throw a `\RuntimeException`.

Exceptions

AccessDeniedException
: Thrown when unaccessible directories are encountered.

AdapterFailureException
: A base class for adapter failures.

ExceptionInterface
: A global interface for finder exceptions.

OperationNotPermitedException
: Based on the `AdapterFailureException`. This exception seems not to be in use within the finder component.

ShellCommandFailureException
: Based on the `AdapterFailureException`. This exception too seems not to be in use within the finder component.

Even though some specific exceptions are defined, they don't seem to be used. For instance, when an incorrect adapter is being set within the finder, it will throw a regular `\InvalidArgumentException`, not the expected `AdapterFailureException`.

Shell

The shell directory contains two classes, `GnuFindAdapter` and the `BsdFindAdapter`, which can be seen as a "lite" version of the `Process` component. They are used without the finder for running external processes. It might have been possible to create a dependency on the `Process` component, so this directory would not have been needed. But honestly, there are some features found in these classes that aren't available within the `Process` component. It would be nice to see them merged in a later version of Symfony. However, using the Process component, as a dependency, means exactly that: a dependency. A nice thing about the finder is that there are no dependencies at all. And since there is some additional code in this shell directory that will never be part of the Process component itself (as it is very specific for this finder component), we never can get truly rid of the shell directory.

The Shell class

The `shell` class is a small class that can test for the existence of files and can detect the current type of shell the PHP version is running. This is mostly needed for the `GnuFindAdapter` and the `BsdFindAdapter`. The class itself consists of two public methods:

getType()

This will return one of the `TYPE_*` constants defined in this class based on the version of the operating system that this PHP version is running on. Not only will it detect operating systems, but it knows how to also detect `cygwin`, which is a linux type emulator, which can run on top of Windows.

testCommand()

The `testCommand` method tests the availability for commands. It will use the external `which` utility for this. When running on Windows platforms, it will use the `where` utility. These utilities will return the full path of the wanted command, if it can be found. The only thing this method will return, is a boolean depending if the needed command is found or not.

The Command class

The `command` class is a small, but powerful class that allows you to build command lines, which can be executed directly. Sometimes, it's not possible to specify a list of arguments or commands directly, and you have several components that will build up the actual command line (this is exactly what's going on in the finder). With the help of this class, it's easy to build commands by adding and even replacing commands and arguments, before the command is actually executed.

This sounds similar to what the `Process` component does, but this `Command` class is actually much more advanced in managing arguments. We will explain this during our dive into the different methods of this class:

create()

This method creates a new instance of the `Command` class. It can be used directly inside a fluent interface:

```
1 $command = Command::create()->cmd('foo')->arg('--bar');
```

__toString()

Used to cast the command to a string. Internally, it will just call the `join()` method.

escape()

Escapes and returns a command through `escapeshellcmd()`. When adding a command through the `cmd()` method (see below), this is automatically called for you.

quote()

Escapes and returns an argument through `escapeshellarg()`. When adding an argument through the `arg()` method (see below), this is automatically called for you.

cmd()

Appends a command and returns itself for a fluent interface. Commands are automatically escaped by using the `quote()` method.

```
// my\\cmd
$cmd1 = Command::create()->cmd('my\cmd');
```

It's theoretically possible to add arguments to here as well. However, they will be escaped improperly. Use the `arg()` method for adding arguments. Also, when adding arguments this way, they are not counted by `length()`.

```
$cmd1 = Command::create()->cmd('foo --bar --baz');
```

arg()

Appends an argument and returns itself for a fluent interface. Arguments are automatically escaped and quoted by the `escape()` method.

```
// my\\cmd '--args' 'foo bar'
$cmd1 = Command::create()->cmd('my\cmd')->arg('--args')->arg('foo bar');
```

add()

Appends a string or command to the current command.

```
$cmd1 = Command::create()->cmd('foo')->arg('--arg1');
$cmd2 = Command::create()->cmd('bar')->arg('--arg2');
$cmd1->add($cmd2);

// foo '--arg1' bar '--arg2'
print $cmd1."\n";

$cmd1->add('| sort');

// foo '--arg1' bar '--arg2' | sort
print $cmd1."\n";
```

top()

Adds a command or string at the beginning of the command.

```php
$cmd1 = Command::create()->cmd('foo')->arg('--arg1');
$cmd1->top("bar");

// bar foo '--arg1'
print $cmd1."\n";
```

ins()

One of the great things about this class is that you can add labels to certain points in your command line. Suppose your command consists of two different pipes commands: a `cat` and a `grep`, which are piped together in the following structure: `cat <args> | sort <args>`.

Now, at some point you want to add arguments to the `cat` part, and some arguments to the `sort` part. You can create labels to specific points in your command, which allows you to use that specific point in your command for adding or changing arguments, without needing to worry about other changing arguments earlier in your command line.

```php
    // Result: find '.'
    $cmd = Command::create()->cmd("find")->arg(".");

    // Result: find '.' | sort
    $cmd->ins('sort')->add('| sort');

    // Result: find '.' | sort -R
    $cmd->get('sort')->add('-R');

    // Result: find '.' | sort -R | uniq
    $cmd->ins('uniq')->add('| uniq');

    // Result: find '.' | sort -R -n | uniq
    $cmd->get('sort')->add('-n');
```

However, you must realize that when adding a label with `ins()`, it will return a new command. This is your "label" point, which you can use to insert items later on. This means that you cannot use `ins()` inside a fluent interface.

```
// Wrong
$grep = Command::create()
    ->cmd("grep")
    ->ins('grepargs')
    ->arg("a")
    ->arg(".")
;
```

```
$grep = Command::create()
    ->cmd("grep")
;
$grep->ins('grepargs');
$grep
    ->arg("a")
    ->arg(".")
;

$grep->get('grepargs')->arg('-R');
$grep->get('grepargs')->arg('-i');
```

get()

The `get()` method will return a previously created command through `ins()` by its label name. If the label does not exist, it will throw a `RuntimeException`.

end()

This seems like a very badly named method. It returns the parent command, provided that the command has been constructed with a parent command. If no parent command is given during constructing of this command, this method will throw a `\RuntimeException`.

Parent commands aren't used in any way: they will not be added to the current command or anything similar. If you want to "chain" commands this way, it might be a great way to start (think about piping multiple commands to each other). However this functionality is not present (yet) in this system. And most likely, it never will be, as its goal is to only serve the `Finder` component.

Don't use this method, or parent commands in general, unless you have a very good reason.

length()

Returns the number of elements (commands and arguments) stored in the current command. Note that when adding multiple commands and/or arguments to the `cmd()` and `arg()` methods are not counted.

```
Command::create()->cmd("echo")->arg("bar")->length();   // 2
Command::create()->cmd("echo 'bar'")->length();   // 1
```

setErrorHandler()

Sets an error handler to capture any output from `STDERR`, when executing the command.

```
// echo 'normal output' ; echo 'error output' >&2
$cmd1 = Command::create()->cmd('echo')->arg("normal output");
$cmd1->add(";");
$cmd2 = Command::create()->cmd('echo')->arg("error output")->add('>&2');
$cmd1->add($cmd2);

$cmd1->setErrorHandler(function ($error_output) {
    print "E> ".$error_output."\n";
});
$output = $cmd1->execute();

var_dump($output);

/*
 E> error output

 array(1) {
   [0] =>
   string(13) "normal output"
 }
*/
```

getErrorHandler()

Returns the current installed error handler, or `NULL`, when no handler is installed.

execute()

Executes the current command. When no error handler is set, it will be simply executed through `exec()` and the output is returned. If an error handler is set, the command is run through `proc_open()`, in order to capture any output from `STDERR`. If such output exists, the error handler will be called before returning the actual output.

The return value of this method is an array of strings from the output.

join()

Concatenates all parts of the command together into a single string.

addAtIndex()

Inserts a string or even previously created command at the specified index. This index is zero based, meaning that adding at index `0` will prepend the command or string. Adding at an index larger than the number of elements will just append it to the end of the command.

Currently, adding a command with the method does not work. As a workaround, wrap the command inside an `array` first. There is an issue and pull request opened[5], but not yet merged as of writing.

```
// echo --bar '--baz'
$cmd1 = Command::create()->cmd("echo")->arg("--bar");
$cmd1->addAtIndex('--baz', 1);


// echo '--bar' '--baz'
$cmd1 = Command::create()->cmd("echo")->arg("--bar");
$cmd2 = Command::create()->arg("--baz");
$cmd1->addAtIndex(array($cmd2), 1);
```

Also note that when adding strings directly, they are not quoted automatically.

[5] https://github.com/symfony/symfony/issues/14384

The filesystem component

Introduction

Another small yet very effective component is the `FileSystem` component. This component is a simple class with some basic and more advanced file system methods that are standard in PHP, but with some more features and/or better handling.

The filesystem component does not have any dependencies on other components or classes.

Directory structure

Since this is a small component, not many directories and files are present.

Exception
: This directory contains exceptions, which are thrown by the component.

Tests
: The `Tests` directory contains PHPUnit tests for this component.

In addition to the directories, there are a few classes defined in the main directory. They are listed here with some small explanations on what they will do, but further in-depth discussion of these classes will occur later in other chapters.

Filesystem.php
: This is the main class of the component which contains the file system abstraction.

LockHandler.php
: This class allows you to create locks through (local) files. A lock can be used for anything: making sure only one process will be running at the same time, that only one process will be able to read or write to a certain file or anything else that needs to be done in a "locked" fashion. There are multiple ways of creating and maintaining locks on operating systems, and this class uses a fairly generic "file locking" system for this purpose.

Exceptions

ExceptionInterface
: This is a base interface, which defines no methods itself.

IOExceptionInterface
: This interface defines the `getPath()` method, which should be implemented by the `IOException`.

IOException
: Whenever an input/output error occurs at file level, this exception will be thrown. Since it implements the `IOExceptionInterface`, the `getPath()` will return the path of the file which triggered the exception.

FileNotFoundException
: When a file is not found, this exception can be thrown. This exception is a extension of the `IOException` class, so here too the `getPath()` can return the path that triggered the exception (ie. the path that was not found).

The FileSystem class

The `FileSystem` class is a collection of file utility methods. Most of them are wrappers around the standard PHP functionality, providing a more generic and platform-agnostic way of dealing with files.

copy()

The `copy` method copies a file to another destination. However, when the file already exists at the destination AND its modification time is older than the source file, only then will the file will be copied. In other words, a file will only be copied, when the source file is newer than the destination file, if it exist. This is often a great way to ensure that you are not overwriting "newer" files with older files.

It's possible to overrule this constraint by setting the third parameter to `true`. In that case, the destination will always be overwritten, even when the destination file is newer than the source file.

```
1  $fs = new Filesystem();
2  $fs->copy('/dir/original.txt', '/anotherdir/result.txt');
3
4  // Always overwrite, even when destination exists and is newer
5  $fs->copy('/dir/original.txt', '/anotherdir/result.txt', true);
```

The `copy()` method will create all directories recursively to the destination, if they do not exist.

Copying streams

The `copy()` copies files through PHP's stream functionality. This means that you are not directly bound to copying files from the filesystem alone, but also from an HTTP url to a local file, an HTTP url to an FTP site or even a file in a compressed tar file to an amazon S3 bucket (provided you register a stream wrapper that allows to do this). For more information about stream protocols and streams in general, take a look at the PHP manual about wrappers[6] and streams[7].

[6] http://php.net/manual/en/wrappers.php
[7] http://php.net/manual/en/book.stream.php

mkdir()

`mkdir()` allows you to create a directory or multiple directories easily. Ultimately, it isn't a lot more than a simple wrapper around the standard `mkdir` functionality of PHP. However, there are some additional checks inside. For instance, it will check to see if the directory is actually created. This can actually be quite useful. For instance, it might be possible that somebody else just created the directory, so your `mkdir` was just a fraction too late. In that case, your `mkdir` will have failed, however, the directory actually does exist. A recipe for long hours of debugging fun!

Even though `mkdir()` catches "directories-created-by-somebody-else" errors, it can also be disadvantageous. For instance, it will not check to see if the directory actually has the correct permissions, which we initially wanted. So theoretically, it is possible that you wanted to create a directory `dir` with very strict permissions, and somebody else created the same directory with very loose permissions. Your `mkdir()` will not complain in that situation.

The `mkdir()` will skip any directories you specify, if they are in fact directories already. However, when you specify a directory that is already present as an actual file, it will try (and fail) to create the directory and throws an `IOException`.

```
1 $fs = new Filesystem();
2 $fs->mkdir('dir', 0777);
3
4 // Create a few different directories
5 $fs->mkdir(array('dir1', 'dir2', 'dir3'));
6
7 // Traversable elements work too
8 $it = new ArrayIterator(array('dir4', 'dir5', 'dir6'));
9 $fs->mkdir($it);
```

About umasks

If you have tried the example from the `mkdir` section, you might have noticed that your directory permissions aren't exactly the same as you have specified.

```
1 $fs = new Filesystem();
2 $fs->mkdir('dir', 0777);
```

Your directory will be created, though when displaying a detailed listing, you will see something like this:

```
$ ls -la
total 1076
drwxr-xr-x 4 jthijssen jthijssen    4096 Mar 25 22:10 .
drwxr-xr-x 4 jthijssen jthijssen    4096 Mar 25 22:05 ..
drwxr-xr-x 2 jthijssen jthijssen    4096 Mar 25 22:10 dir
```

The `drwxr-xr-x` in front of our `dir` entry actually maps back to permissions `0755`, and not to our `0777` permission we specified when creating the directory. This is because we have to deal with something called the `umask`.

The `umask`, or `file mode creation mask`, is a mask that will be added through all file creation functions on operating system level. So it is not something you can "change" from a PHP or Symfony point of view, however, you are able to change this umask.

As the default, the `umask` will be set to `0022`. This setting means that when creating files (or directories, which are in fact files from a unix point of view as well), the system will mask off the group write permission bit and the other write permission bit. This in turn means that only the creator of the file is able to write to the file.

Let's change the default umask and try again with another directory:

```
// Only mask the "other" permission write bit
umask(0002);
$fs = new Filesystem();
$fs->mkdir('dir1', 0777);
```

```
$ ls -la
total 1076
drwxr-xr-x 4 jthijssen jthijssen    4096 Mar 25 22:10 .
drwxr-xr-x 4 jthijssen jthijssen    4096 Mar 25 22:05 ..
drwxr-xr-x 2 jthijssen jthijssen    4096 Mar 25 22:10 dir
drwxrwxr-x 2 jthijssen jthijssen    4096 Mar 25 22:16 dir1
```

Notice that our `dir1` now has their group write permission bit set, but not the other write permission bit. If we would change our umask to `0000`, even the other write permission bit will be set, in effect giving the directory the permissions you've actually asked for in `mkdir()`.

Do not rely on the `umask()` to be `0022`. If you do not want others except your group and yourself to be able to read/write or browse a a directory, set the permissions in `mkdir()` to `0770`. This way, no matter what the `umask` is set to, working with the file system will always result in the correct permissions.

exists()

`Exists` allows you to check if a file or collection of files exists or not. When a collection (through an array or traversable object) is passed, this method will return `false` as soon as at least one of the given files is missing.

Note that internally, this method uses the `file_exists` function of PHP. This function will return true if a file (or directory, device etc) exists. When a file is a symbolic link AND the target does NOT exist, `file_exists` will return false. To check if a symbolic link actually exists - without worrying if its target does - use the `is_link` PHP function instead.

touch()

The `touch` method will "touch" a file or collection of files. This means that their modification time (`mtime`) and/or access time (`atime`) will be changed. However, when a file does not exist yet, it will automatically be created, as an empty file.

By default, `touch()` will change the `mtime` and `atime` to the current time, but it's possible to specify both times explicitly through this function.

Just like a lot of functionality, when a file cannot be processed (touched, in our case), the method will throw an `IOException`. This means that when specifying a collection of files, it will stop at the first failure. It will not continue to touch other files. In essence none of the methods of this class are atomic (either everything succeeds or nothing does).

remove()

This method is used to remove a file, directory or collection of files / directories. When a directory is specified, the method will recursively remove all files in this directory.

> This method will do a recursive remove. Be careful when removing directories recursively as it might be disastrous, when you recursively remove the wrong directory.

The method will remove all files in reversed order, mostly to make sure that recursive deletions will start with the deepest entries first. When a directory has been found (and it's not a symlink to a directory), it will call this `remove()` method recursively with a collection of files taken from the `FilesystemIterator` [8]. This iterator will iterate a directory recursively, so it will pick up all files and directories within the starting directory.

There is an bug/issue[9] within PHP that seems to differ in behavior based on which platform you are running. For instance, on a Windows platform, a symlink to a directory must be removed `rmdir()`, while on other platforms, this can be done with the more sensible `unlink()` PHP function. The `remove()` method of this class will automatically detect and calls the correct functionality.

Just like all other methods, when a file cannot be removed, it will throw an `IOException` and leaves the remaining files in place.

chmod()

The `chmod` can manually change the permissions of files. It even allows the changing of directory permissions recursively.

> Note that when dealing with file permission masks, arguments are expected to be passed as `octal` values. These might seem similar to normal numbers, but instead only uses the number `0` to `7`. The number `10` in octal is actually the number `8` in decimal. This means that larger numbers are not what you might think:

```
$fs = new Filesystem();
$fs->touch(array('file1', 'file2'));

$fs->chmod('file1',  777);  // Not octal
$fs->chmod('file2', 0777);  // Octal
```

And the result:

```
-r----x--t 1 jthijssen jthijssen       0 Mar 26 09:45 file1
-rwxrwxrwx 1 jthijssen jthijssen       0 Mar 26 09:45 file2
```

[8] http://www.phparch.com/books/mastering-the-spl-library/

[9] https://bugs.php.net/bug.php?id=52176

As you can see, the first file, using decimal permissions, have strange permissions, while the second file is correct.

Do not forget the leading `0` to indicate an octal number!

chown()

This method allows you to (recursively) change the name of a file or collection of files. When a symlink is found AND the `lchown()` PHP function exists (it is not available on the Windows platform), it will try and change the owner of the symlink, not the target of the symlink. Note that this will only work when running the script as the root user.

For the `user` parameter, you can either specify an existing user on your system OR specify a valid user id (`uid`).

```
1  $fs = new Filesystem();
2  $fs->touch(array('file1', 'file2'));
3
4  $fs->chown('file1', 'games:games');
5  $fs->chown('file2', 65534); // uid 65534 equals 'nobody' on some distributions
```

Unlike the linux `chown()` tool, this method cannot set both the owner and group at the same time (like `chown user.group file`)

chgrp()

The `chgrp()` does exactly the same as the `chown()`, only for the group of a file.

rename()

This method will rename an existing file (it cannot use an array or collection of files!) to something else. This could be used for instance to actually rename a file, or to change the file from one directory to another one. The `$overwrite` argument will specify if the target must be overwritten, if it already exists (it will not do so by default).

```php
$fs = new Filesystem();

// Rename a file
$fs->rename('foo.txt', 'bar.txt');

// Move the file to another directory
$fs->rename('foo.txt', 'another/dir');
```

Note that when renaming a file, the directory structure must be present. It will not automatically create the directory structure needed.

symlink()

This method creates a symbolic link to a directory. When the target directory does not exist, it will be created (recursively) first. When the target already exists as a symlink, the symlink will be removed first.

On systems where symbolic links are not available (Windows platform), you can set a third parameter `$copyOnWindows` to `true`. That will actually copy all the contents of the directory to the target directory. However, it's not a symlink: any changes in one directory will not be seen in the other!

makePathRelative()

The `makePathRelative()` method returns the relative path based on a starting path. Basically, it answers the question: "what is the path that leads from `$startPath` to `$endPath`?".

```php
$fs = new Filesystem();

// bar/baz/
$fs->makePathRelative('/foo/bar/baz', '/foo');

// ../foo/bar/baz/
$fs->makePathRelative('/foo/bar/baz', '/dir');
```

In essence, it strips down both paths to directories, throws away the directories they have in common, and adds enough times `../` to get to the common base directory part.

mirror()

This method will 'mirror' a directory to another directory. It behaves a bit like the `rsync` utility found on unix platforms.

There are four arguments to be passed: the `$origininDir` and the `$targetDir`, the directories in question. An `$iterator` option, which must be a `Traversable` instance, and finally `$options`.

Iterating

One of the nice features of this `mirror()` method is that you can specify which files should be mirrored by supplying a custom iterator. When you don't pass one, it will internally use the `RecursiveDirectoryIterator` to recursively iterate over all the elements in the directory. However, you could pass your own iterator (for instance, through a filter iterator), to only copy certain files.

```
1  $fs = new Filesystem();
2
3  // Set a recursive iterator that filters filenames ending on .mp3
4  $it = new \RecursiveDirectoryIterator('src', \FilesystemIterator::SKIP_DOTS);
5  $it = new \RecursiveRegexIterator($it, '/\.mp3$/i');
6  $it = new \RecursiveIteratorIterator($it,
7                 \RecursiveIteratorIterator::CHILD_FIRST);
8
9  // Mirror only .mp3 files from src to dst
10 $fs->mirror('src', 'dst', $it);
```

Options

There are a few options that can be passed to this method, all boolean values:

copy_on_windows
: When on windows, must we copy files over when a symlink is found, instead of creating a symlink (which isn't implemented under windows).

delete
: Use this, when files that are available in the target directory, but not in the source directory, should be removed. If set to `true`, both directories are really mirrored. When set to `false`, additional files in the target directory might be available.

Because of some bugs, you cannot use a custom iterator together with the 'delete' option.

override
: If a file must be overwritten, even when the timestamp of the destination is newer than the original one (see `copy()`), use override.

isAbsolutepath()

The `isAbsolutePath()` method will check if the path given is an absolute file path or not. It will check both for unix-type systems (starting with a `/`), Windows schemes (starting with `X:\`) or checks if the path is a url.

dumpFile()

`dumpFile()` is a method that allows you to atomically dump data into a file. It does this by creating a temporary file, fills it with your contents, and finally "changes" the name of the temporary file into the actual destination.

The reason this process is atomic is because it calls the PHP `rename()` function internally, which in turn calls the `rename()` system call. This call is atomic: a file will be renamed, and no other processes will be able to see a missing file during this process.

toIterator()

The `toIterator()` method is a private method, which ensures that anything passed to it will be returned, as something that is iterable. When the `$files` argument passed is `Traversable`, it can be traversed (ie. iterated) by `foreach`. However, when the argument passed is not a `Traversable`, this method will return it into something traversable by wrapping the argument inside an `ArrayObject`. This method is used in multiple other methods to make sure that files provided is something that can be traversed, without worrying if a user provided an array, an iterator, or just a single filename as a string.

The LockHandler class

The `lockhandler` class is a simple class that allows you to easily create locks based on files. With the help of locks, you can ensure that certain parts of your code are only run by you, without worrying about other processes that might interfere. This could be vital, for instance, when you are writing data to a file and you don't want others to write to the same file at the same time.

```
$lock = new LockHandler("test", "lockdir");

// Wait (block) until we get the lock
if ($lock->lock(true)) {

    // Do stuff here that only we are allowed to do when holding a lock

    // Release the lock again
    $lock->release();
}
```

Constructing a `LockHandler` is done by supplying a name and additionally a path. If no path has been given, the lock handler will automatically use your temporary directory (through `sys_get_temp_dir()`). If the path does not exist, it will create the directory first. This will be done in the `__constructor`, and even when you don't actually use the lock.

It will mangle the name of the lock file by using the actual name provided and the `sha256` hash of the name. This is mostly to ensure that the file will be unique and you can easily use more common and descriptive names for your locks:

```
$lock = new LockHandler('critical event #1');
if ($lock->lock()) {
    // do stuff
    $lock->release();
}


$lock = new LockHandler('critical event #2');
if ($lock->lock()) {
```

```
10      // do something else
11      $lock->release();
12  }
```

lock()

The `lock()` method actually creates and locks the file. When you call the `lock()` method twice, it will not "double lock" the file, but returns directly (as you are the one, who already holds the lock in the first place).

```
1  $lock = new LockHandler('critical event #1');
2  $lock->lock();  // Creates the lock
3  $lock->lock();  // Does nothing
4  $lock->lock();  // Does nothing
5
6  $lock->release();  // Will release the lock
```

Atomic reads and writes

Stick around for some complexity when dealing with locking, race-conditions and multi-process code:

First, the `lock()` method will try and open the actual file in read (`r`) mode. This works if the locked file is already present (and readable).

If a file could not be found, or was not readable, it will try and create the file in `"open+write"` mode. This is a "special" mode, as it will ensure that the file will only be created, should it not yet exist. Even though we just checked this in the previous line, it might be possible that some other process just created the file after we checked if the file was present, but before we could create the file ourself (this a so-called `race condition`). This "open+write" mode of opening a file will ensure that either the file will be created, or it will fail, if somebody else has already created it just before us.

If the creation of the file succeeded, we change the permissions of the file to `0444`, making it readable for others (and ourselves). If we couldn't open the file, it might be because some other process already created the file, and in that case, we can try and open the file for reading again, because it will probably exist at this point.

There is a catch here: remember that a file was created as write-only (call this event A), and we change the permissions to `0444` through `chmod` (call this event B). Suppose process 1 has just finished event A. Process 2 at this point is running event A and finds that it could not create the file (since it's just been created by process 1). Process 2 will continue and try to open the file for reading, but finds that it cannot. This is because process 1 still hasn't finished event B (changing the permissions). At

that point, process 2 will `usleep` for a short while, giving process 1 the opportunity to finish event B. Process 2 will finally try and open the file again for reading, in the hope that process 1 has finished on time. There is no guarantee that it has finished (the process might be paused, or things just takes a long time), but it's a good check to catch in most situations.

If after this point we still can't open the file for reading, we give up with an IOException. Otherwise, we have a file handle from a file and we can actually create a lock on it.

This complexity is needed to ensure that everything will work without race-conditions, no matter how processes are executed through the code. Realize that this part of the code is written with an execution flow or two (or more!) separate processes in mind, and thus, might not all make sense in the "normal" way of imperative thinking.

Locking through flock()

The locking process itself is not difficult from a user point of view: we only have to call the `flock()` PHP function with the `LOCK_EX` and `LOCK_NB` flag. This will return either `true`, when the file is exclusively locked by us, or `false`, when another process has already locked the file. It is NOT POSSIBLE that two processes can hold the same lock, because underwater, this locking will be done atomically (which is MUCH harder than it sounds).

But, often our code just wants a lock, and when we can't have it yet, it should wait for it to become available. Instead of writing user land loops for this (which are very inefficient), we can supply the boolean parameter `$blocking` to the `lock()` method. This will call `flock()` without the `LOCK_NB` flag, meaning the function will block until a lock becomes available. In effect it will halt your code until a lock is free, without you having to worry about it, and it does so in a efficient way by literally pausing your program until the lock becomes available.

However, you do have to be careful with this, as there is no guarantee that you will get the lock within a certain amount of time, or even at all! Always check if you acquired the lock or not by checking the result of the `lock()` method.

Always check the result of `lock()` to see if you really got the lock.

release()

The `release()` method must be called, when it's time to release a lock. By calling this method, and if the lock file was opened (and locked), it will call the `flock()` method to unlock the file and directly closes the file.

There are cases, when it's possible to omit this call. For instance, when your PHP script terminates, all remaining open files will be automatically closed, including the lock file, which will also be

automatically unlocked. This means it's not possible to have a "dangling" lock file created by a terminated process, even if you forgot to unlock a file.

Your operating system will automatically unlock lock files when a process ends or crashes. It's not possible to lock a file that cannot be unlocked anymore, because your PHP script has crashed for instance.

After releasing a lock, the lock file is unlocked, but it isn't automatically removed. There is no option available for removing lock files, and it's hard to do this manually, as you don't have any real say about the name of the actual lock file, nor will you have any methods for finding the name of the used lock file. A good way to deal with this is to keep your lock files inside a separate directory and use an external process to clean up files if needed.