# COMPOSER

## ORCHESTRATING PHP APPLICATIONS

DAYLE REES

# PHP: Composer

Orchestrating PHP Applications

Dayle Rees

This book is for sale at http://leanpub.com/composer-php

This version was published on 2016-05-16

# Tweet This Book!

Please help Dayle Rees by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading Composer: Orchestrating PHP Applications by @daylerees - https://leanpub.com/composer-php #composer

The suggested hashtag for this book is #composer.

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=#composer

# Contents

# Acknowledgements

First of all, I would like to thank my girlfriend Emma, for not only putting up with all my nerdy ventures but also for taking the amazing red panda shots for the books! Love you, Emma!

Thanks to my parents, who have been supporting my nerdy efforts for close to thirty-two years! Also, thanks for buying a billion or so copies of the first few books for family members!

Thank you to everyone who bought my previous books and to all of the Laravel community. Without your support, futures titles would not have been possible.

# Errata

While this may be my fifth book, and my writing is steadily improving, but I assure you that there will be many, many errors. I don't have a publisher, a review team, or an English degree. I do the best that I can to help others learn about Laravel and PHP. I ask that you, please be patient with my mistakes. You can support the title by sending an email with any errors you might have found to me@daylerees.com[1] along with the section title.

Errors will be fixed as they are discovered. Fixes will be released in future editions of the book.

[1] mailto:me@daylerees.com

# Feedback

Likewise, you can send any feedback you may have about the content of the book or otherwise by sending an email to me@daylerees.com[2]. You can also send a tweet to @daylerees[3]. I will endeavour to reply to all mail that I receive.

[2] mailto:me@daylerees.com
[3] https://twitter.com/daylerees

# Translations

If you would like to translate this book into your language, please send an email to me@daylerees.com[4] with your intentions. I will offer a 50/50 split of the profits from the translated copy, which will be priced at the same as the English copy.

Please note that the book is written in markdown format, and is versioned on Github[5].

[4]mailto:me@daylerees.com

[5]http://github.com

# 1. Introduction

Hi there! Nice to meet you!

My name's Dayle, and I'm a thirty-one-year-old web developer from Cardiff, the capital of Wales in the UK. I've been writing PHP based applications for close to a decade, been heavily involved with the Laravel PHP framework, have written four self-published programming books, and given conference talks worldwide. I promise that you're in good hands as we venture into a new topic with this book.

I'm hoping that the lack of a red panda on the cover, this time, hasn't scared you away. Hopefully, everyone will love the angry looking bird! I'm not even sure what it is. My girlfriend snapped a shot of it one day at the Zoo, and I found it in her photo album. Some kind of puffin, maybe? I should probably check, but not right now!

As you've already gathered. I don't write books like a normal person. In fact, I'm at least 90% sure that I'm not a normal person. The other 10% of my opinion belongs to my split personality.

I write my books with humor and simplicity. It's like we're sitting in the pub together, sharing a drink and learning about programming. I consider my readers to be friends, and I hope that you'll feel the same. The simple language in my books is not just refreshing to new readers, but it's very important to my many foreign-language readers. They find it much more simple to understand my books, than a complex technical book with long unnecessary words. I'm not looking to win any awards with my control of the English language, but I'm sure you're going to learn a bunch of new skills along the way.

I discovered Composer during the development of Laravel four. You see, the framework uses it to piece its components together. Back then, Composer wasn't as widespread as it is now, and I hope that the framework adoptions were partly responsible for its success.

Composer is the most exciting project in the PHP world. It has changed the landscape of the language, and it makes building PHP applications an enjoyable experience.

Right then, I bet you're eager to get started? All that's left to do is thank Jordi Boggiano for creating Composer, and flip that page!

# 2. Concept

Let's start our journey by examining the concept behind Composer. What is this software? What does it do?

I expect that you've all had some experience with building PHP applications. You don't have to have used a framework, but you'll at least need to understand the language. If not, I would suggest starting with my book PHP Pandas[1].

When you're creating an application in PHP, you don't want to write all of it yourself. Certain problems have already been solved, and it doesn't make any sense to re-invent the wheel.

Fortunately, we live in a world where open source rules. Software and libraries that are free of charge, and often have shortcuts or utilities to make our jobs as developers that little bit easier. In writing classic PHP applications, you've likely included a bunch of files and classes with your applications. These are your application dependencies.

Here are some examples:

- Libraries for formatting strings and numbers.
- Libraries for working with dates and times.
- Libraries for interacting with a database.

And of course, many, many more. In fact, there are probably millions of pieces of re-usable code out there in the PHP ecosystem. Previously, you'd find some of the more significant libraries on repositories such as PECL, but they weren't easy to install, and would have to be installed for *all* PHP applications on your system, not just the one you're working on. That's really inconvenient, isn't it?

Luckily, this is where Composer comes to the rescue. In fact, I believe that there are three big reasons to use Composer to power your PHP applications. Let's take a look at them now.

## Dependency Management

If we decide to include all of our dependencies ourselves. We'll need to put them in a directory inside our application. We'll need to load their classes or files using `include` or `require`. We might need to bootstrap them with configuration variables, and then,

[1]http://leanpub.com/php-pandas

three years from now they'll be completely out of date, and will explode when you update your version of PHP. In fact, how do we know that our dependencies won't conflict with each other? We have no way of knowing.

> That's a lot of problems!

You're completely right, reader. We don't want to have to deal with all these issues. We'd prefer to focus on building our applications, and not worrying so much about our dependencies.

This is where Composer arrives, wearing it's spandex and cape, to save the day. Composer will take care of your dependencies for you. It will help you install them; they will automatically be available within your applications, and you won't have to `include` them yourself. It will even install version updates of your dependencies for you so that you never go out of date.

Simply put, it's going to make it much easier to install, use, and keep your dependencies up to date. Let's take a look at the next reason to use Composer.

## Class Autoloading

As mentioned in the previous section, Composer will automatically provide the classes and functions available in your dependencies, so that you won't have to use `include` everywhere.

If only we could do that in our applications. Having to `include` every file before we can use a class is a little repetitive, and can litter our code with a bunch of unnecessary `include` and `require_once` statements. We don't want to be messing with that.

> I see where this is going…

I can't trick you, can I reader? You're right. Composer can take care of this for us. It ships with a variety of methods of matching up class names to the files that contain them. So you'll never have to `include` a class again. Isn't that great?

Let's take a look at my third most important reason for adopting composer.

## Team Collaboration

Our final reason for using Composer applies mainly to those working in teams. However, it's a feature that can also be useful to maintainers of libraries and frameworks.

Composer ships with a file that defines the packages in use by your application. In fact, you version it alongside your code. It will detail packages and the exact versions of installed dependencies.

This means that with a quick `composer install` (don't worry; this will arrive in a later chapter) you're running the exact same version of all of your dependencies as the rest of your team. No more cases of "Dude, what do I need to run this?" or "Maybe I need to update my dependencies?". Composer will handle this for you.

Hopefully, you're sold on the benefits of Composer, and I assure you that while these three are the important ones for me, there are much, much more.

In the next chapter, we'll learn about packages. They are super important to Composer. Flip the page; I dare you!

# 3. Packages

Composer lives and breathes packages. Packages are small containers of code that have been created to serve a particular package.

For example, a 'PayPal' Composer package might provide code that will make it more simple to interact with the PayPal payment gateway. A 'Math' package might provide some useful classes and/or functions for performing mathematical calculations and interacting with numbers.

Composer packages are limited only by the author's creativity. We'll soon discover this when we take a closer look at the 'Packagist' Composer package repository.

Before we take a closer look at a package; I think it would be useful to learn about the different types of Composer packages available. I like to separate Composer packages into two distinct categories by their usage. Let me explain.

## Application Packages

Some developers might argue that this shouldn't be called a package at all. In some ways, it may be better to describe these packages as a 'Composer-driven project'. I like to consider any project directory that contains a `composer.json` configuration file (more on this later) a package.

Application packages contain a `composer.json` only to list other packages as dependencies, or take advantage of Composer's fantastic class autoloading capabilities. They aren't intended to be re-distributable. It's your project. Your baby. It might be a confidential codebase, or something experimental, but it's not meant for sharing.

Just because your package isn't shared, it doesn't mean that you can't use the other packages as dependencies.

In the good ol' days of PHP, if you wanted to take advantage of an open source calculation class then you would have copied it into your project, or implemented this functionality yourself. You wouldn't receive updates from the original author unless you update the files by hand. You were missing out on hotfixes and new features. These were sad times.

By listing another package as a dependency of your Composer driven application you can use all that great code without any drawbacks. If the author decides to update his or her package, then you can update your local version by running `composer update`. If you are worried about hidden updates or changes making their way into the dependency package, then you can fix it to a specific version. This way you will always receive the same codebase as a dependency.

You can even ensure that a dependency version exists within a certain range of version numbers, is stable, is a development version and many, many more options! We'll learn more about this later.

You can list as many packages as you like as dependencies. A typical Composer configuration file may contain the following code snippet.

**Example 01: Require block.**

```
"require": {
    "laravel/framework": "4.1.*",
    "solarium/solarium": "3.2.0",
    "omnipay/omnipay": "~2.0",
    "mockery/mockery": "0.8.0"
},
```

Don't worry too much about the implementation, but it should be clear that in the above example we are defining four dependencies for our application. These packages contain code and classes that we wish to use.

The first package `laravel/framework` is an entire PHP framework, and has many dependencies of its own. This doesn't phase Composer at all. It will quite happily retrieve the entire dependency tree for us.

Retrieving dependencies is the most important Composer feature within an application package contest, but it's not the only one! Composer is also able to manage the mapping of file paths to class definitions, and will load PHP classes within your own application or dependency packages whenever you need them.

You no longer have to make a thousand calls to `include()` the files containing the classes that you wish to use. Let Composer do the hard work! We'll take a closer look at this feature within the 'autoloading' chapter of the book.

One other interesting feature of Composer is the ability to move binary or executable files to a specific location. This makes it possible to include useful tools such as PHPUnit or PHP Mess Detector as a development dependency of your application and then execute them right within your project directory. You no longer need to install these applications globally! (Actually, composer makes it easier to install them globally too, if that's what you prefer!)

I don't want to cover every feature here. We have a whole book for that! Why don't we take a look at the other type of package available for use with Composer?

## Dependency Packages

A dependency or ‘redistributable’ packages are packages of code which are *intended* for use as a dependency of another package. They aren’t our website, blog or CSS showcase. They are libraries and components. Perhaps even frameworks built of multiple components.

The format of a dependency package is no different to that of an application package. The configuration is nearly identical. It can list dependencies, autoloading mechanisms and all of that fun stuff!

Dependency packages are likely to be software development kits for a particular service. Libraries or classes that provide specific and re-usable functionality. I’ve even seen some dependency packages that don’t provide any PHP code at all! I’ve seen some that are used to include CSS frameworks, but I don’t wish to overcomplicate this chapter by explaining how that works.

To summarize, a Composer package may simply be a consumer of other Composer packages and tools to empower an application, or it may be a container for redistributable open source code that is intended to make the lives of other developers easier!

In the next chapter, we’ll learn how to install Composer. Don’t worry; in most situations you’ll only have to do it once, and it’s quite a simple process.

# 4. Installation

Before we can begin to use Composer, we must first install it. That's right; it doesn't ship with your computer. Shame on Apple or some other makes I guess!

Composer exists as a PHP script wrapped in a `.phar` file. This `phar` is a sort of archive, much like a `.zip` file, except that it is executable. The method of installing Composer is simply retrieving this binary.

Before we attempt to retrieve the Composer `phar` file. Let's check to make sure we meet the requirements to use the application. You will need all of the following.

- PHP 5.3.2+ CLI
- PHP CURL extension.
- PHP LibSSL extension.
- PHP Zip extension.

As I mentioned earlier, Composer is a PHP application, and as such, requires PHP installed to function. It also needs the extensions mentioned above. If you try to install composer without first satisfying these requirements, you will receive an error.

Right, Windows users, are you here? Go and wait in the corner for a little while. First, we are going to cover the Unix (Mac/Linux) installation instructions.

## Unix based OS

First, we need to retrieve the Composer installer. We can use the `CURL` application to achieve this goal. On a recent Mac, you will find that CURL is already installed, however, on certain Linux distributions, including Ubuntu, you will need to install CURL using your package manager. For example:

**Example 01: Install CURL.**

```
$ sudo apt-get install curl
```

Once CURL is installed, we can continue to install the software. We will use CURL to retrieve the installation script; then we will use the pipe `|` modifier to execute the script with PHP. Let's give it a go.

**Example 02: Download Composer.**

```
$ curl -sS https://getcomposer.org/installer | php
#!/usr/bin/env php
All settings correct for using Composer
Downloading...

Composer successfully installed to: /Users/dayle/Projects/composer.phar
Use it: php composer.phar
```

As we can see, all our settings are correct. This is Composer's way of saying that we meet all of the requirements for using Composer. It then procedes to download the `.phar` file, and lets us know where it has been downloaded to.

Let's execute Composer to see if it's working. We can use the `php` CLI interface to execute the `.phar` file and see the result.

**Example 03: Run Composer.**

```
$ php composer.phar
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version fb72d26def99d08ee32e5be7b553817f585f164e 2013-12-20 10:53:52

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for normal ou\
tput, 2 for more verbose output and 3 for debug
  --version        -V Display this application version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --profile           Display timing and memory usage information
  --working-dir    -d If specified, use the given directory as working directory.

```

```
Available commands:
  about            Short information about Composer
  archive          Create an archive of this composer package
  config           Set config options
  create-project   Create new project from a package into given directory.
  depends          Shows which packages depend on the given package
  diagnose         Diagnoses the system to identify common errors.
  dump-autoload    Dumps the autoloader
  dumpautoload     Dumps the autoloader
  global           Allows running commands in the global composer dir ($COMPOSER\
_HOME).
  help             Displays help for a command
  init             Creates a basic composer.json file in current directory.
  install          Installs the project dependencies from the composer.lock file\
 if present, or falls back on the composer.json.
  licenses         Show information about licenses of dependencies
  list             Lists commands
  require          Adds required packages to your composer.json and installs them
  run-script       Run the scripts defined in composer.json.
  search           Search for packages
  self-update      Updates composer.phar to the latest version.
  selfupdate       Updates composer.phar to the latest version.
  show             Show information about packages
  status           Show a list of locally modified packages
  update           Updates your dependencies to the latest version according to \
composer.json, and updates the composer.lock file.
  validate         Validates a composer.json
```

Well, that sure is a lot of commands! In fact, it will be a useful reference for when you begin using Composer within your own applications.

It’s clear that Composer is functioning correctly, and we could continue to use it in this fashion. We will have to install Composer for all of our projects, though, won’t we? Why don’t we install it globally? Now that’s DRY (Don’t Repeat Yourself) thinking!

To install Composer globally, we need only move it to a location within our `PATH` environmental variable. This will allow us to execute the command from anywhere within our system. Normally the `/usr/local/bin` directory is part of everyone’s path, but feel free to substitute it for another location within the command below.

**Example 04: Global install of Composer.**

```
$ mv composer.phar /usr/local/bin/composer
```

You will notice that we strapped 'composer' onto the end of the target destination. This is because we don't want to be typing `composer.phar` to execute it all the time. Let's just use `composer` instead by renaming the file.

Let's test to make sure that the relocated command functions as expected.

**Example 05: Run Composer once again.**

```
$ composer
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version fb72d26def99d08ee32e5be7b553817f585f164e 2013-12-20 10:53:52

Usage:
  [options] command [arguments]
```

Brilliant! I hid a few of the commands to save some space, but as we can see, Composer is executing as expected. It is now installed!

Unix folk, feel free to skip to the updating section. It's time for the Windows folk to be educated.

## Windows

Bear with me Windows folk. It has been some time since I last developed on Windows. Still, let's give this a go.

Before installing Composer, you must first find a directory that exists within your `PATH` environment variable, as shown within the Windows `Environmental Variables` system settings dialog. Let's imagine that the directory `c:\bin\` exists within your PATH.

First, let's open a command prompt, by running `cmd` after clicking the `Start` button, and the `Run...`. We will first move to the directory located within our PATH.

**Example 06: Move to location within our path.**

```
C:\>cd bin
C:\bin>
```

Next, we will use PHP to retrieve the file, and to evaluate it. It's a complicated command so, please make note of all the special characters.

**Example 07: Fetch Composer.**

```
C:\bin>php -r "eval('?>'.file_get_contents('https://getcomposer.org/installer'))\
;"
```

The Composer `.phar` file will be downloaded to the current directory, but isn't conveniently executable in its current form. Let's use the Windows `echo` command to output an execution script into a new `composer.bat` file. Here is the command required.

**Example 08: Create a Composer script.**

```
C:\bin>echo @php "%~dp0composer.phar" %*>composer.bat
```

Wonderful! We're done.

We should probably test that it's working, though, right? Let's open a fresh command prompt. You remember how, right? Next, we will attempt to execute Composer.

**Example 09: Run Composer.**

```
C:\>composer
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version fb72d26def99d08ee32e5be7b553817f585f164e 2013-12-20 10:53:52

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for normal ou\
tput, 2 for more verbose output and 3 for debug
  --version        -V Display this application version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --profile           Display timing and memory usage information
  --working-dir    -d If specified, use the given directory as working directory.

Available commands:
  about            Short information about Composer
  archive          Create an archive of this composer package
  config           Set config options
  create-project   Create new project from a package into given directory.
  depends          Shows which packages depend on the given package
  diagnose         Diagnoses the system to identify common errors.
  dump-autoload    Dumps the autoloader
  dumpautoload     Dumps the autoloader
  global           Allows running commands in the global composer dir ($COMPOSER\
_HOME).
  help             Displays help for a command
  init             Creates a basic composer.json file in current directory.
  install          Installs the project dependencies from the composer.lock file\
 if present, or falls back on the composer.json.
  licenses         Show information about licenses of dependencies
  list             Lists commands
  require          Adds required packages to your composer.json and installs them
  run-script       Run the scripts defined in composer.json.
  search           Search for packages
  self-update      Updates composer.phar to the latest version.
  selfupdate       Updates composer.phar to the latest version.
  show             Show information about packages
  status           Show a list of locally modified packages
  update           Updates your dependencies to the latest version according to \
composer.json, and updates the composer.lock file.
  validate         Validates a composer.json
```

Wonderful! That wasn't so hard, was it?

## Updating

Every now and again you will want to update Composer to take advantage of the latest bug fixes and features. In fact, Composer will let you know when your current version is 30 days out of date.

**Example 10: Composer update notification.**

```
$ composer install
Warning: This development build of composer is over 30 days old. It is recommend\
ed to update it by running "/usr/local/bin/composer self-update" to get the late\
st version.
```

As the comment suggests, we can use the `self-update` command to instruct Composer that it should update itself. For example:

**Example 11: Update Composer.**

```
$ composer self-update
```

On unix based systems you may need to use the `sudo` command to allow Composer to execute with administrative privileges in order to update itself. Just like this:

**Example 12: Update Composer with privileges.**

```
$ sudo composer self-update
```

Be sure to keep your Composer installation up to date. You never know what new features you might be missing out on. Also, if you happen to encounter a bug with Composer, it's always worth updating to see if it has been fixed in a later release.

In the next chapter, we'll learn how to initialise a Composer powered project, so that we can begin orchestrating!