How To Install and Use Composer on Ubuntu 18.04

9

Updated October 16, 2018 95.6k PHP UBUNTU 18.04

By: Brian Hogan

Not using **Ubuntu 18.04**? Choose a different version:

A previous version of this tutorial was written by Brennen Bearnes.

Introduction

Composer is a popular *dependency management* tool for PHP, created mainly to facilitate installation and updates for project dependencies. It will check which other packages a specific project depends on and install them for you, using the appropriate versions according to the project requirements.

In this tutorial, you'll install and get started with Composer on an Ubuntu 18.04 system.

Prerequisites

To complete this tutorial, you will need:

- One Ubuntu 18.04 server set up by following the Ubuntu 18.04 initial server setup guide, including a sudo non-root user and a firewall.

Step 1 — Installing the Dependencies

Before you download and install Composer, you'll want to make sure your server has all dependencies installed.

First, update the package manager cache by running:

```
$ sudo apt update
```

Now, let's install the dependencies. We'll need `curl` in order to download Composer and `php-cli` for installing and running it. The `php-mbstring` package is necessary to provide functions for a library we'll be using. `git` is used by Composer for downloading project dependencies, and `unzip` for extracting zipped packages. Everything can be installed with the following command:

```
$ sudo apt install curl php-cli php-mbstring git unzip
```

With the prerequisites installed, we can install Composer itself.

Step 2 — Downloading and Installing Composer

Composer provides an installer, written in PHP. We'll download it, verify that it's not corrupted, and then use it to install Composer.

Make sure you're in your home directory, then retrieve the installer using `curl`:

```
$ cd ~
$ curl -sS https://getcomposer.org/installer -o composer-setup.php
```

Next, verify that the installer matches the SHA-384 hash for the latest installer found on the Composer Public Keys / Signatures page. Copy the hash from that page and store it as a shell variable:

```
$ HASH=544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdfdd586475ca9813a858088ffbc1f233e9b180f
```

Make sure that you substitute the latest hash for the highlighted value.

Now execute the following PHP script to verify that the installation script is safe to run:

```
$ php -r "if (hash_file('SHA384', 'composer-setup.php') === '$HASH') { echo 'Installer verified'; } e
```

You'll see the following output.

Output

```
Installer verified
```

If you see `Installer corrupt`, then you'll need to redownload the installation script again and double check that you're using the correct hash. Then run the command to verify the installer again. Once you have a verified installer, you can continue.

To install `composer` globally, use the following command which will download and install Composer as a system-wide command named `composer`, under `/usr/local/bin`:

```
$ sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer
```

You'll see the following output:

```
Output
All settings correct for using Composer
Downloading...

Composer (version 1.6.5) successfully installed to: /usr/local/bin/composer
Use it: php /usr/local/bin/composer
```

To test your installation, run:

```
$ composer
```

And you'll see this output displaying Composer's version and arguments.

```
Output
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version 1.6.5 2018-05-04 11:44:59

Usage:
  command [options] [arguments]

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

This verifies that Composer installed successfully on your system and is available system-wide.

Note: If you prefer to have separate Composer executables for each project you host on this server, you can install it locally, on a per-project basis. Users of NPM will be familiar with this approach. This method is also useful when your system user doesn't have permission to install software system-wide.

To do this, use the command `php composer-setup.php`. This will generate a `composer.phar` file in your current directory, which can be executed with `./composer.phar command`.

Now let's look at using Composer to manage dependencies.

Step 3 — Using Composer in a PHP Project

PHP projects often depend on external libraries, and managing those dependencies and their versions can be tricky. Composer solves that by tracking your dependencies and making it easy for others to install them.

In order to use Composer in your project, you'll need a `composer.json` file. The `composer.json` file tells Composer which dependencies it needs to download for your project, and which versions of each package are allowed to be installed. This is extremely important to keep your project consistent and avoid installing unstable versions that could potentially cause backwards compatibility issues.

You don't need to create this file manually - it's easy to run into syntax errors when you do so. Composer auto-generates the `composer.json` file when you add a dependency to your project using the `require` command. You can add additional dependencies in the same way, without the need to manually edit this file.

The process of using Composer to install a package as dependency in a project involves the following steps:

- Identify what kind of library the application needs.
- Research a suitable open source library on Packagist.org, the official package repository for Composer.
- Choose the package you want to depend on.
- Run `composer require` to include the dependency in the `composer.json` file and install the package.

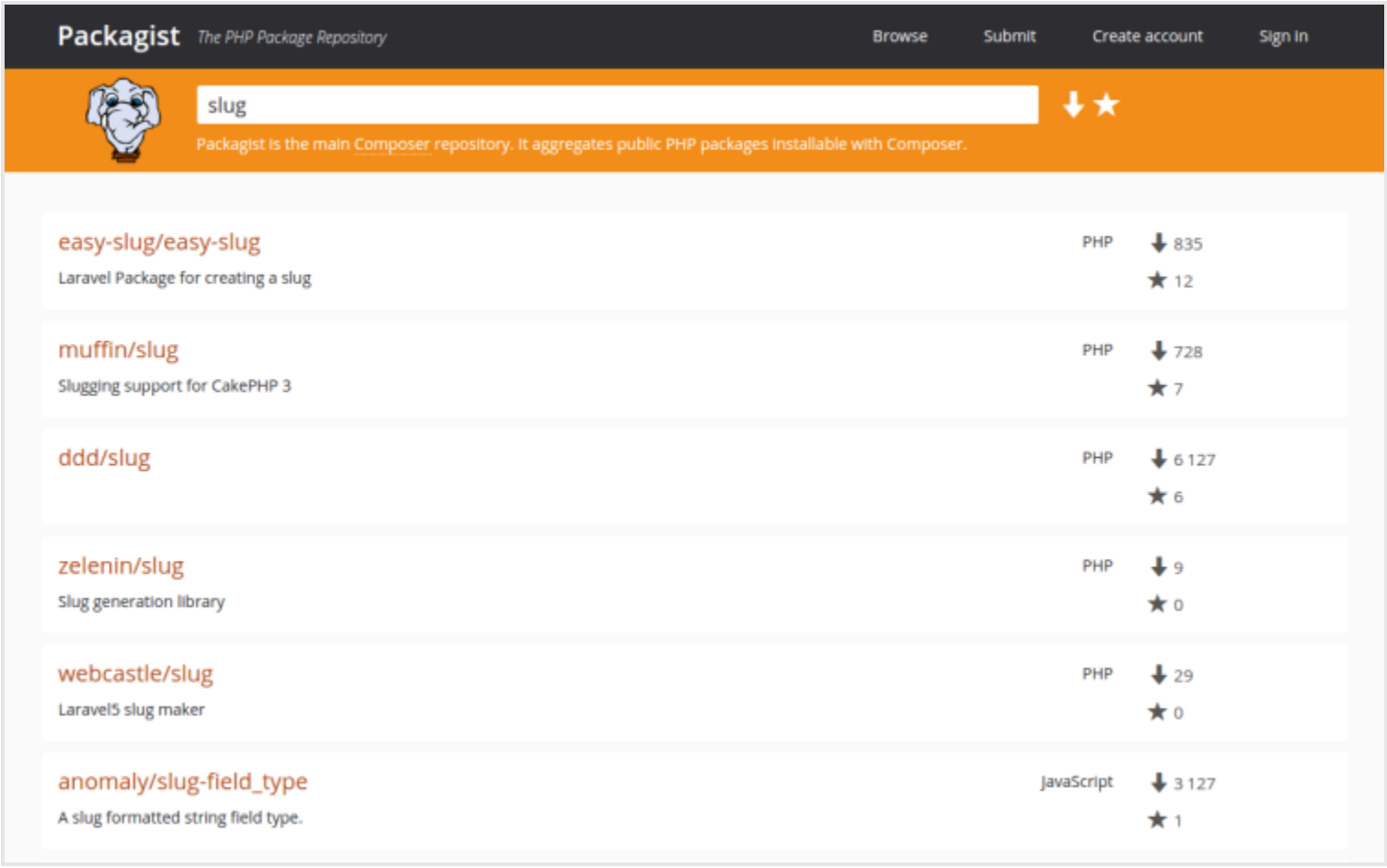
Let's try this out with a demo application.

The goal of this application is to transform a given sentence into a URL-friendly string - a *slug*. This is commonly used to convert page titles to URL paths (like the final portion of the URL for this tutorial).

Let's start by creating a directory for our project. We'll call it **slugify**:

```
$ cd ~
$ mkdir slugify
$ cd slugify
```

Now it's time to search Packagist.org for a package that can help us generate *slugs*. If you search for the term "slug" on Packagist, you'll get a result similar to this:



You'll see two numbers on the right side of each package in the list. The number on the top represents how many times the package was installed, and the number on the bottom shows how many times a package was starred on GitHub. You can reorder the search results based on these numbers (look for the two icons on the right side of the search bar). Generally speaking, packages with more installations and more stars tend to be more stable, since so many people are using them. It's also important to check the package description for relevance to make sure it's what you need.

We need a simple string-to-slug converter. From the search results, the package `cocur/slugify` seems to be a good match, with a reasonable amount of installations and stars. (The package is a bit further down the page than the screenshot shows.)

Packages on Packagist have a **vendor** name and a **package** name. Each package has a unique identifier (a namespace) in the same format GitHub uses for its repositories, in the form `vendor/package`. The library we want to install uses the namespace `cocur/slugif`. You need the namespace in order to require the package in your project.

Now that you know exactly which package you want to install, run `composer require` to include it as a dependency and also generate the `composer.json` file for the project:

```
$ composer require cocur/slugify
```

You'll see this output as Composer downloads the dependency:

```
Output
Using version ^3.1 for cocur/slugify
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing cocur/slugify (v3.1): Downloading (100%)
Writing lock file
Generating autoload files
```

As you can see from the output, Composer automatically decided which version of the package to use. If you check your project's directory now, it will contain two new files: `composer.json` and `composer.lock`, and a `vendor` directory:

```
$ ls -l
```

```
Output
total 12
-rw-rw-r-- 1 sammy sammy   59 Jul 11 16:40 composer.json
-rw-rw-r-- 1 sammy sammy 2934 Jul 11 16:40 composer.lock
drwxrwxr-x 4 sammy sammy 4096 Jul 11 16:40 vendor
```

The `composer.lock` file is used to store information about which versions of each package are installed, and ensure the same versions are used if someone else clones your project and installs its dependencies. The `vendor` directory is where the project dependencies are located. The `vendor` folder doesn't need to be committed into version control - you only need to include the **composer.json** and **composer.lock** files.

> When installing a project that already contains a `composer.json` file, run `composer install` in order to download the project's dependencies.

Let's take a quick look at version constraints. If you check the contents of your `composer.json` file, you'll see something like this:

```
$ cat composer.json
```

```
Output
{
    "require": {
        "cocur/slugify": "^3.1"
    }
}
sam
```

You might notice the special character `^` before the version number in `composer.json`. Composer supports several different constraints and formats for defining the required package version, in order to provide flexibility while also keeping your project stable. The caret (`^`) operator used by the auto-generated `composer.json` file is the recommended operator for maximum interoperability, following semantic versioning. In this case, it defines **3.1** as the minimum compatible version, and allows updates to any future version below **4.0**.

Generally speaking, you won't need to tamper with version constraints in your `composer.json` file. However, some situations might require that you manually edit the constraints–for instance, when a major new version of your required library is released and you want to upgrade, or when the library you want to use doesn't follow semantic versioning.

Here are some examples to give you a better understanding of how Composer version constraints work:

Constraint	Meaning	Example Versions Allowed
^1.0	>= 1.0 < 2.0	1.0, 1.2.3, 1.9.9
^1.1.0	>= 1.1.0 < 2.0	1.1.0, 1.5.6, 1.9.9
~1.0	>= 1.0 < 2.0.0	1.0, 1.4.1, 1.9.9
~1.0.0	>= 1.0.0 < 1.1	1.0.0, 1.0.4, 1.0.9
1.2.1	1.2.1	1.2.1
1.*	>= 1.0 < 2.0	1.0.0, 1.4.5, 1.9.9
1.2.*	>= 1.2 < 1.3	1.2.0, 1.2.3, 1.2.9

For a more in-depth view of Composer version constraints, see the official documentation.

Next, let's look at how to load dependencies automatically with Composer.

Step 4 — Including the Autoload Script

Since PHP itself doesn't automatically load classes, Composer provides an autoload script that you can include in your project to get autoloading for free. This makes it much easier to work with your dependencies.

The only thing you need to do is include the `vendor/autoload.php` file in your PHP scripts before any class instantiation. This file is automatically generated by Composer when you add your first dependency.

Let's try it out in our application. Create the file `test.php` and open it in your text editor:

```
$ nano test.php
```

Add the following code which brings in the `vendor/autoload.php` file, loads the `cocur/slugify` dependency, and uses it to create a slug:

test.php

```
<?php
require __DIR__ . '/vendor/autoload.php';

use Cocur\Slugify\Slugify;

$slugify = new Slugify();

echo $slugify->slugify('Hello World, this is a long sentence and I need to make a slug from it!');
```

Save the file and exit your editor.

Now run the script:

```
$ php test.php
```

This produces the output `hello-world-this-is-a-long-sentence-and-i-need-to-make-a-slug-from-it`.

Dependencies need updates when new versions come out, so let's look at how to handle that.

Step 5 — Updating Project Dependencies

Whenever you want to update your project dependencies to more recent versions, run the `update` command:

```
$ composer update
```

This will check for newer versions of the libraries you required in your project. If a newer version is found and it's compatible with the version constraint defined in the `composer.json` file, Composer will replace the previous version installed. The `composer.lock` file will be updated to reflect these changes.

You can also update one or more specific libraries by specifying them like this:

```
$ composer update vendor/package vendor2/package2
```

Be sure to check in your `composer.json` and `composer.lock` files after you update your dependencies so that others can install these newer versions.

Conclusion

Composer is a powerful tool every PHP developer should have in their utility belt. In this tutorial you installed Composer and used it in a simple project. You now know how to install and update dependencies.

Beyond providing an easy and reliable way for managing project dependencies, it also establishes a new de facto standard for sharing and discovering PHP packages created by the community.

By: Brian Hogan

Upvote (9) Subscribe Share

We just made it easier for you to deploy faster.

TRY FREE

Related Tutorials

How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04

How To Set Up Laravel, Nginx, and MySQL with Docker Compose

How to Deploy a Symfony 4 Application to Production with LEMP on Ubuntu 18.04

How To Install and Use Composer on Debian 9

How To Install and Secure phpMyAdmin on Debian 9

5 Comments

Leave a comment...

Log In to Comment

edrd *September 23, 2018*

0 If you are following this tutorial on a personal machine/account, make sure to run `sudo chown -R $USER $HOME/.composer` if you have any permissions problem.

corumcrm *October 18, 2018*

0 $ sudo apt install curl php-cli php-mbstring git unzip

> E: Unable to locate package php-mbstring

corumcrm *October 19, 2018*

0 solved editing /etc/apt/sources.list

gerardreches *December 23, 2018*

0 [deleted]

jchaven *January 6, 2019*

0 sudo nano /etc/apt/sources.list
---------- copy ----------
deb http://archive.ubuntu.com/ubuntu bionic main multiverse restricted universe
deb http://archive.ubuntu.com/ubuntu bionic-security main multiverse restricted universe
deb http://archive.ubuntu.com/ubuntu bionic-updates main multiverse restricted universe
---------- copy ----------

Source: https://askubuntu.com/questions/1062160/package-php7-2-mbstring-unmet-dependencies-in-ubuntu-18-04

gerardreches *December 23, 2018*

0 I saw this composer installation method everywhere, but I also tried just with `sudo apt-get install composer` and it is working fine!

Why would we want to take the long way? Has something wrong the `apt-get` method?

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Copyright © 2019 DigitalOcean™ Inc.