Basic Hello World with composer and php

Getting started with composer

I want this to be an introduction to using Composer. If you are looking for more information about using Drupal with Composer then you should checkout the official composer facade doc page on drupal.org.

I will *not* be talking about publishing to Packagist. My php is nearly always for Drupal, and as such, if I publish anything it will most likely be to a project on drupal.org. In the end I will philosophize about Drupal and Composer, but we will start with the composer basics.

Lets get started!

When dealing with any package managed project, I like to start with initializing the project. This goes for npm or composer. True, it isn't 100% necessary, but it is a good way to keep it all organized from the beginning.

Initialize our package

After installing composer, run this command in your project's directory.

```
$ composer init
```

This will run you through a bunch of questions: *name*, *description*, *license*, etc. These are the prompts (at the time of this writing) and my responses to the prompts.

```
  Welcome to the Composer config generator



This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [fanderson/composer]: frob/greetings
Description []: This is a simple hello world example.
Author [Frank Anderson <frob@249517.no-reply.drupal.org>, n to skip]:
Minimum Stability []: dev
Package Type (e.g. library, project, metapackage, composer-plugin) []:
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? n
Would you like to define your dev dependencies (require-dev) interactively [yes]? n
```

Most of these should be easy enough to figure out and I will not go over all of them here. If you want to know what all this stuff really means, then go read the composer docs.

Write some code

I will start coding by creating the file `greetings.php` in the `src/HelloWorld` directory:

```
<?php

namespace HelloWorld;
```

```
class Greetings {
  public static function sayHelloWorld() {
    return 'Hello World';
  }
}
```

Add some basic dependencies

Now we have created our class, but we still have to tell composer about it. To do that we need to make some changes to our `composer.json` file.

```
{
    "name": "frob/greetings",
    "description": "This is a simple hello world example.",
    "authors": [
        {
            "name": "Frank Anderson",
            "email": "frob@249517.no-reply.drupal.org"
        }
    ],
    "minimum-stability": "dev",
    "require": {
        "php": ">=5.5.0"
    },
    "autoload": {
        "psr-0": {
            "HelloWorld": "src/"
        }
    }
}
```

The new stuff starts in the `"require"` part. My changes are some simple ones, first, I have php 5.5.24 installed locally. Thus I add compatibility to anything greater than 5.5.

> *Notice: I am not doing it here, but you need to be careful with open ended requirements. I have no idea if this code will work on php 9, but I don't care so I didn't bother adding the `|| <9.0` or whatever to the required php version.*

The second change is adding PSR-0 autoloading of my HelloWorld namespace. This will become important soon.

Install dependencies

Now we get to run `composer install` and see all the magic happen.

```
$ composer install
```

This does three important things:

Updating dependencies (including require-dev)

This project has no dependencies other than php version 5.5 or higher. But if it did then they would be installed. By default they are installed in the `vendor` directory. Unlike npm there are plugins that can change where vendor code is installed.

Writing lock file

If we look at our directory tree after running `composer install` it looks like this:

```
composer.json
composer.lock
src/
  greetings.php
vendor/
  autoload.php
  composer/
```

Notice the `composer.lock` file. That file needs to be added to the repository.

When executing builds for production the lock file will be used to ensure that the same versions of things that worked for you in development are going to be installed in production. That way you don't have to worry about mysterious dependency versions breaking things in deploy.

Generating autoload files

This is the really cool part of composer. It generates an `autoload.php` file. This file is used by composer to automatically load files based on the PSR-0 spec we stated earlier (see, I told you it would be important).

This is also why composer installs itself into the vendor directory --composer is what handles the autoloading. This is huge, and also a big reason to use composer.

Now so long as we handle the namespaces correctly and we follow the PSR (in this case PSR-0) then we can just load the autoloader and composer will make sure we have what we need when we need it.

Lets use it

Right now we have some potential energy. All this code, but nothing is using it. Kind of like the strength with OOP is also its weakness -- encapsulated code doesn't do anything by itself. Thus, we need to write some *cough* procedural code that uses this object.

> *Fun fact, I really did cough when I wrote that.*

So then, lets write some tests. Create a `tests` directory and add a file `test.php` to it with the following contents.

```
<?php

// Autoload files using the Composer autoloader.
require_once __DIR__ . '/../vendor/autoload.php';

use HelloWorld\Greetings;

echo Greetings::sayHelloWorld();
```

Like I said before, autoloading and composer give us the handy `use` and `namespace` keywords. It is beyond the scope of this article to discus how the autoloader works, just that it does and how to use it.

Remember the directory stucture `src/HelloWorld` and the require statement in the `composer.json` file? In our `composer.json` we told the autoloader to expect a psr-0 directory structure and that our code would be in `src`. Then we declared the `namespace HelloWorld;` in our `greetings.php` file.

Now in our `test.php` file we are *using* (with the `use` keyword) the `HelloWorld\Greetings` namespace and object in our test to call the `Greetings::sayHelloWorld();` static method.

Lets commit

At this point lets commit our code. The begs the question, what is our code? For now that is only the src and tests directory and the composer.php file. Lets initialize and commit those.

```
git init
git add src tests composer.json
git commit -m "initial commit"
```

We should also setup a `.gitignore` file so git stops barking at us about the vendor and `composer.lock` file.

> *If this was a project I would be leaving the composer.lock in file the repository. We have set this up more like utility library than an actual project. It is only valuable for deployment and it be ignored by composer when this is used as a library anyway.*

So run this from the command line:

```
$ echo "vendor" >> .gitignore
$ echo "composer.lock" >> .gitignore
```

This will append both `vendor` and `composer.lock` to the `.gitignore` file (or create it if it doesn't exist) thus it's contents will look like this.

```
vendor
composer.lock
```

Bonus points, something borrowed

I was going to stop here, but after reading that I think it is missing something. The other real power from composer. Using someone else's code.

I still want to keep this simple so I am adding something simple and *maybe* useful: a timer **phpunit/php-timer**.

To add 3rd party packages to your package you run the `composer require` command. In order to add the php-timer we run:

```
$ composer require phpunit/php-timer
```

Unlike npm we do not need to specify the `--save` argument. Composer assumes the save. Here is another awesome thing that composer does for us. Composer will **"solve"** our dependency tree for us. What that means is that the packages will tell composer what versions of dependencies they support and composer will figure out what the latest version that fits that requirement.

For example if dependency **foo** can use version **5** and above of the **php-timer** package, but dependency **bar** doesn't work with version **7** or version **5**, but does work with version **6**. Then composer knows that only the latest release of version **6** is supported by your application.

> *If there are lots of dependencies and sub-dependencies then solving this issue takes time. Also, composer will tell you if it cannot be solved. If two packages are not compatible then composer will tell you right away.*

After we run that command we will see `"phpunit/php-timer": "^1.0"` was added as a requirement. We can tell composer to only use some versions, but I will let you read the docs on that.

There is one more thing that we need to do before we can use the *PHP_Timer* class in our project. We will need to edit the `composer.json` file to tell our autoloader about our new class.

```
"autoload": {
    "psr-0": {
        "HelloWorld": "src/"
    },
    "classname": {
      "PHP_Timer": "src/"
    }
}
```

Here we are adding the `classname` keyword and then listing the "PHP*Timer" class as member with "src/" as the value. I got this value because inside the* `vendor/php-timer` *directory is the "PHP*Timer" class, without any `namespace` keyword. Due to the lack of a namespace we just have to tell the autoloader that when we declare `use PHP_Timer;` in our php file that we need the *PHP_Timer* class loaded.

> *I fingured out what to do by looking at packages that used PHP_Timer. It can be confusing, Read composer namespaces in five minutes to learn more about how composer uses namespaces.*

Now that we have added the classname to the composer file we can start to use it in our greetings.php file. Lets add the some use of the php-timer class to the *Greetings* class.

```
<?php
namespace HelloWorld;

use PHP_Timer;

class Greetings {
  public static function sayHelloWorld() {
    $timer = new PHP_Timer();
    $timer->start();
    return "Hello World\n" . $timer->resourceUsage() . "\n";
  }
}
```

Now when we run the test it will print:

```
$ php tests/test.php
Hello World
Time: 1 ms, Memory: 0.50MB
```

Scripts, scripts, and scripts

Currently, to test our command we need to run `php tests/test.php` from the command line. That is about as simple as it gets. But what about if we want to run a linter or some real tests? This is where the *scripts* member in the `composer.json` file comes into play.

Lets add a *scripts* definition to the end of our `composer.json` file:

```
"scripts": {
  "test": "php tests/test.php"
}
```

Now we only need to run the command `composer test` or `composer run test` from the command line in order to run our tests. The value of the *scripts* member is a list of key value pairs where the key is the name of the command and the value is the command that is run. The value can even be an array of commands that composer will run.

If we change that to:

```
"scripts": {
  "test": ["php tests/test.php", "echo Hello"]
}
```

Then composer will run both the commands.

> *`composer run` is shorthand for `composer run-script`*

There are a whole slew of pre-defined commands that we can add to our `composer.json` file. For example, using the key `post-update-cmd` will run the scripts after the `update` and before the `install` commands.

Download my Hello World Greetings composer project from github.

Composer and Drupal

As my professional life goes, 99% of my php work is Drupal -- and it has been since release of Drupal 6. I have done work in other languages and I have used other frameworks; but when it comes to me getting paid to write php, it is almost always Drupal.

As I said, I got started with Drupal seriously with Drupal 6, but now, as is the Drupal way, Drupal 6 is dead and Drupal 8 is new. Drupal 8 was a nearly complete re-architecture of Drupal.

Drupal was special, it was special because when we (Drupal Developers) needed something we built it. Our issue was that we built it for us. Now the needs of Drupal aren't unique. We build drush, but every project seems to have a cli now-a-days.

Drush has make functionality. It can handle bundling our projects together with a build process. It also gives us the ability to download and install modules or change/modify our configuration. We can start temporary web-servers with Drush and do simple development work or test patches. The issue is that Drush was our tool and no one else's.

With Drupal 8 we are leaving the island and seeing how others have solved problems. We don't have to do it all ourselves and sometimes it is better not to even try. There is a whole slew of functionality out there that we can now incorporate into our modules. Composer is the key in all of that.

Written on August 16, 2016 by Frank Robert (frob) Anderson

fahri rizki • a year ago

Uncaught Error: Class 'helloworld\Greetings' not found, i get this error, why?? ijust clone your project and run 'composer install', may i miss something?

^ | v • Reply • Share ›

Frank Robert Anderson Mod ➚ fahri rizki • 3 months ago • edited

Okay, so I have gotten around to testing this and it should all still work as an example.

1. Clone repo from github
2. run `composer install`
3. run `composer run test`

The above three step should still work as of the end of 2019. Not sure what else you are doing that could be causing failure.

^ | v • Reply • Share ›

이상민 ➚ fahri rizki • 3 months ago

Check This.

DIRECTORY : src/HelloWorld
FILENAME : Greetings.php

Reply • Share ›

Frank Robert Anderson Mod ➚ fahri rizki • a year ago

Probably missing that this example is 3 years old and might not work anymore. When I have time I will update it as this was never tested with php7.

All the text should still be correct, but the specific code may no longer work, sorry.

Reply • Share ›

Stop Medicaid Theft • 2 years ago • edited

Great job. I am working though the exercises now. I will link to this.

Reply • Share ›

Andrew Bell • 4 years ago

Great tutorial, thanks! I'm new to PHP and wanted to just get a general idea of how Composer worked, so this really was focused to my needs (and up to date). As a fellow blogger, wanted to give a little feedback, but overall, it was very clear and had the right amount of detail!

I was using IDEA, and it threw me off because it saw PHP_Timer as undefined, but running the test was fine!

I got a little mixed up, I didn't have src/helloworld directory structure at first, I might've missed it at first in the tutorial (the Writing lock file section shows a directory structure that looks different than the end result)

Also, as someone new to PHP, go ahead and show me how to run the test before we import a 3rd party library.

Reply • Share ›

Frank Robert Anderson Mod ➚ Andrew Bell • 4 years ago • edited

Thanks for the feedback. I will have to double check the final directory structure to make sure it is the same. I write these in a live blog style (coding as I go), so it is likely that something changed from the first section to the last.

Everyone's IDE will be different so I didn't want to focus on just one. Normally I use phpStorm, but in this case I wanted no IDE interference so I used atom. IMO there isn't a really good IDE for php yet, but phpStorm is good enough.

 • Reply • Share ›

Social Networks

Blogroll

BTMash's blob of contradictions
LA Grafitti
Justin Biard's icodealot