Wonderful World of Composer

Chapter 1: Composer

COMPOSER: BECAUSE USING EXTERNAL LIBRARIES SHOULD BE FUN!¶

Welcome to the brand new world of PHP with Composer!

One of the great things about PHP is that we're huge! But how big are we really? PHP consists of a lot of successful, but extremely isolated libraries, like Wordpress, Drupal, Joomla and all of the PHP frameworks like Symfony and Zend. Historically, these different groups share almost nothing, which means that in reality, we're all pretty small islands. That's right, even though we're the biggest group in the world, we've somehow turned ourselves into the underdogs.

Well, it's time for a whole new beautiful era where your community is *all* of the PHP world.

PHP developers haven't typically shared code or used outside libraries because, well, it sucked! To use just one outside library, you'd need to tackle at least three major issues:

1) First, how do I autoload the PHP classes in the library? Autoloading is the background machine that makes it possible to reference PHP classes without using `require` or `include` statements. When I bring in outside code, I either need to figure out which files to include or how to configure that library's autoloader.

2) Second, we need to know if this library depends on any other libraries. And if it does, that's yet another library I need to download and configure.

3) Finally, how should I store the library in my project? Should I use SVN externals? Git submodules? Just commit the whole darn thing into my project?

The answer to these 3 problems is Composer. Want to do something crazy like bring Symfony's Finder component into a Drupal project? We're about to learn just how easy that is. But wait! Before you non-Drupal people run off, the process and code you'll see have nothing to do with Drupal. This is one of the first great things about using Composer: the process for using external libraries is the same now in *any* PHP project.

Our Project¶

Move into the directory where your project lives. Our first goal is to use a standalone Symfony component called "Finder" to list all the files in a directory and print them onto the page. Just to make things really exciting, we're going to do this inside a custom Drupal module.

To start, I've created a function called `get_current_files` deep inside my project, which returns an array of filenames:

```
// sites/all/modules/list_files/list_files.module
// ...

function get_current_files()
{
```

```
    return array('foo');
}
```

Tip

I've already created a module called `list_files` and enabled it before starting the tutorial. I've also gone far enough to print these onto the screen. Ok great, now let's get to work!

Installation¶

The first thing you'll need to do is download the Composer executable. Go to GetComposer.org, click "Download", then copy one of the two code blocks depending if you have `curl` installed.

```
$ curl -s https://getcomposer.org/installer | php
```

Composer itself is just an executable file, and this fancy bit of code downloads the file and makes sure your system is setup to use Composer. If you see any errors or warnings during this step, you may need to tweak your PHP configuration.

If everything went ok, you'll have a brand new `composer.phar` sitting at the root of your project. Ok, let's put this guy to work! Execute the composer script by typing `php composer.phar`:

```
$ php composer.phar
```

This shows you a list of all the available composer commands: we'll get to know some of these in the next few minutes.

Creating the composer.json File¶

So far, we have a `composer.phar`, but that's it! Composer's main job is to download third-party libraries into a `vendor/` directory in your project. To tell Composer which libraries you need, your project needs to have a `composer.json` configuration file. Instead of creating this file by hand, let's use the first Composer command: `init`.

```
$ php composer.phar init
```

The `init` command will ask you several questions about your project, but unless you're planning to open-source it, don't worry too much about your answers. Finally, it'll ask you to interactively define your dependencies. By "dependencies", Composer is asking you which third-party libraries you want to include in your project. To start, we're going to add a Symfony2 component called Finder - and you can find its documentation at *symfony.com*.

At the prompt, simply type "finder" and wait for the results. Behind the scenes, Composer is searching against a giant central repository of packages called "Packagist", which you can search directly at packagist.org. In the language of Composer, a package is just an individual directory that you want to download into your project. A typical package contains PHP classes, but it can really contain anything.

Each package has a unique name, and your first job is to find the name of the one you need. Ideally, the name is included in the documentation for the library, but even if its not, you can often find it just by searching. On Packagist, searching for "finder" reveals a package called `symfony/finder`, which is definitely the right one!

Back at the terminal, our search for "finder" has returned a bunch of results including many versions of `symfony/finder`. The second thing we need to figure out is which version we want. The safest choice is to choose the latest stable release, which should follow the X.X.X

format. In our case, this is `v2.1.2`. If a version ends in `-dev`, it's a development branch, which may be stable or unstable based on the library. The `dev-master` version is special, and it always means the latest, bleeding-edge code. If you use a lesser-known package, `dev-master` may be your only option.

Let's choose to install `v2.1.2`. When it asks you to install "dev dependencies" interactively, choose no. You probably won't need to worry about dev dependencies, but if you're curious about them, check out Composer's documentation. Finally, confirm generation and add `vendor/` to your `.gitignore` file if you're storing your project with git:

```
# .gitignore
/vendor/
```

The end-product of the `init` command is the new `composer.json` file that's now in your project. Open it and check out the `require` key: this is really the only important part of this file right now and it simply tells Composer which packages your project needs. We could have created this file by hand - the `init` task is just there for convenience.

```
{
    "name": "weaverryan/drupal",
    "require": {
        "symfony/finder": "v2.1.2"
    },
    "authors": [
        {
            "name": "Ryan Weaver",
            "email": "ryan@thatsquality.com"
        }
    ]
}
```

Using the "install" command¶

At this point, we've downloaded the Composer executable and created the `composer.json` config file. To actually put Composer to work, run `php composer.phar install`. This is the most important command: it reads the `composer.json` file and downloads all the needed libraries into the vendor directory:

```
$ php composer.phar install
```

And look, a vendor directory!

```
vendor/
    composer/
    symfony/
        finder/
            Symfony/...
```

It contains a symfony directory that holds the Finder library and a few other things that help with autoloading - which is one of the most powerful features of Composer.

Autoloading¶

Now that Composer has downloaded the Finder library, let's use it! To keep things simple, I'll paste in some Finder code that looks for all gif files that have been modified within the past day:

```
// sites/all/modules/list_files/list_files.module
// ...

/**
 * A utility function to return the array of current SplFileInfo objects
 */
function get_current_files()
{
    // the "files" directory
    $dir = drupal_realpath(file_default_scheme() . '://');

    $finder = new \Symfony\Component\Finder\Finder();
    $finder->in($dir)
        ->name('*.gif')
        ->date('since 1 day ago')
    ;
    $files = array();
    foreach ($finder as $file) {
        $files[] = $file->getFilename();
    }

    return $files;
}
```

This code should work, but when we refresh the page, we get a class not found error!

Class SymfonyComponentFinderFinder not found.

Of course! Even though Composer downloaded the Finder library for us, we can't use any of its PHP classes without including them.
Fortunately, Composer solves this for us - through autoloading. The exact details of how autoloading works goes beyond this screencast, but the important thing is that Composer helps us out. To use Composer's autoloader, simply include the `vendor/autoload.php` file somewhere in your project. For now, let's put it right inside this function:

```
// sites/all/modules/list_files/list_files.module
// ...

function get_current_files()
{
    require __DIR__.'/../../../../vendor/autoload.php';

    // ...
}
```

Refresh the page again. It works! By including Composer's autoloader, the Finder library - as well as the PHP classes for any other libraries we included via Composer - are made available

to us automatically.

To make our third-party classes available anywhere, it would be even better to include the autoload file in some central, bootstrap file in your project. For Drupal, this might be the `settings.php` file:

```
// sites/default/settings.php
require __DIR__.'/../../vendor/autoload.php';

// ... the rest of the file
```

When we refresh, everything still works.

The composer.lock file and “install” versus “update”¶

Things are going so well that I think we should add another library! So let’s get crazy! Head back to packagist.org and find a library `symfony/filesystem`. To tell Composer that we want this package, just edit the `composer.json` by hand, add a second entry under the `require` with the name of the library. To make things more interesting, let’s use the `2.1.x-dev` version, which will give us the latest commit on the 2.1 branch:

```
{
    "name": "weaverryan/drupal",
    "require": {
        "symfony/finder": "v2.1.2",
        "symfony/filesystem": "2.1.x-dev"
    },
    "authors": [
        {
            "name": "Ryan Weaver",
            "email": "ryan@thatsquality.com"
        }
    ]
}
```

Next, we need to tell Composer to re-read this file and download the new library.

Before, we used the `install` command to do this. But if you try that command now, it prints out a few lines, but doesn’t actually do anything. Why not?

```
# does not download the new library :/
$ php composer.phar install
```

When it comes to downloading the libraries we need, composer actually has two different commands: `install` and `update`.

When we ran the `install` command earlier, one of the things it did was create a `composer.lock` file that recorded the exact versions of all libraries that it downloaded at that exact moment.

Normally, the `install` command actually ignores the `composer.json` file and reads all of the information from this lock file instead. If you make a change to `composer.json` and run `php composer.phar install`, that change won’t be used. The lock file is important, because if multiple developers are using a project, each one can run `php composer.phar install` and receive identical versions of all libraries, even if new commits have been added to them.

In fact, the *only* time that the `install` command reads the `composer.json` file is when you first start the project, because the lock file doesn't exist yet.

In this one case, `install` acts exactly like the `update` command, which always ignores the lock file and reads the `composer.json` file instead. This checks and potentially upgrades all the libraries in `composer.json` and updates `composer.lock` when it finishes.

What this ultimately means is that you should use a simple workflow. Unless you're adding a new library or intentionally upgrading something, always use `composer.phar install`.

Using the update Command¶

When you *do* need to add a new library or upgrade something. Use `composer.phar update`. You can be even more precise by calling `update` and passing it the name of the library you're updating. By doing this, Composer will only update *that* library, instead of all of them.

```
$ php composer.phar update symfony/filesystem
```

Icing in the Cake: The require Command¶

Also, Composer has a cool shortcut command for adding new libraries into your project:

```
$ php composer.phar require
```

Tip

You can use the require command to search for a library. In this example, I searched for the `doctrine/dbal` package and added it.

With the `require` command, you can search for the package you need and Composer will automatically update your `composer.json` for you *and* run the `update` command to download the library. In this case, when I included the `doctrine/dbal` package, an extra packaged called `doctrine/common` was downloaded. This is dependency management in action. Composer is smart enough to know that `doctrine/dbal` depends on `doctrine/common` and it downloads it for you. Woo!

Version Control¶

The lock file is especially important if you have multiple developers so that you can be sure that each person has identical vendor libraries. To make this possible, commit both your composer.json file *and* your `composer.lock` file:

```
$ git add composer.json
$ git add composer.lock
$ git add .gitignore
$ git add sites
```

Typically `composer.phar` is ignored, since each developer can download it individually.

Now, let's pretend like we're a new developer that's pulling down the codebase.

```
$ cd ..
$ git clone drupal drupal2
$ cd drupal2
```

Notice that the project doesn't have a `vendor/` directory yet, because we didn't commit the vendor files. In fact, we ignored the `vendor` directory in git, because Composer can populate it for us.

I'll copy in the `composer.phar` file from the previous directory and then run `php composer.phar install`.

```
$ cp ../drupal/composer.phar .
$ php composer.phar install
```

This reads the `composer.lock` file and downloads everything we need into the `vendor/` directory. And just like that, your new developer has a functional project!

Conclusion¶

There's a lot more that Composer can do, but you already understand how to find libraries, manage your `composer.json` file, use Composer's autoloader and download the external libraries with the `update` or `install` commands.

If you'd like to learn more, check out the documentation at GetComposer.org. One interesting topic is scripts - which are callbacks that are executed before or after packages are installed. Other important topics include the dump-autoload command, "dev dependencies", minimum stability, and installing Composer globally. If you want to start your Symfony project using Composer checkout our latest version of Getting Started in Symfony. Good luck and See ya next time!

With <3 from SymfonyCasts