Managing Dependencies with Composer

Sunshine PHP
February 8th, 2014

Beau Simensen

@beausimensen

beau.io

#composer

look for **simensen** and say "hi"

freenode IRC

Dragonfly Development

@dflydev

dflydev.com

reflxlabsinc.com

@reflxlabs

REFLX Labs

boogio™

boogio.com

@wearboogio

COMPOSER

Dependency Management

Dependencies are external requirements

Managing dependencies for PHP projects has not always been trivial

Composer makes dependency management easier

`composer.json`

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

vendor-name/project-name

A package's name cannot change and must be all lowercase

Vendor name should be unique to the developer, project, or company

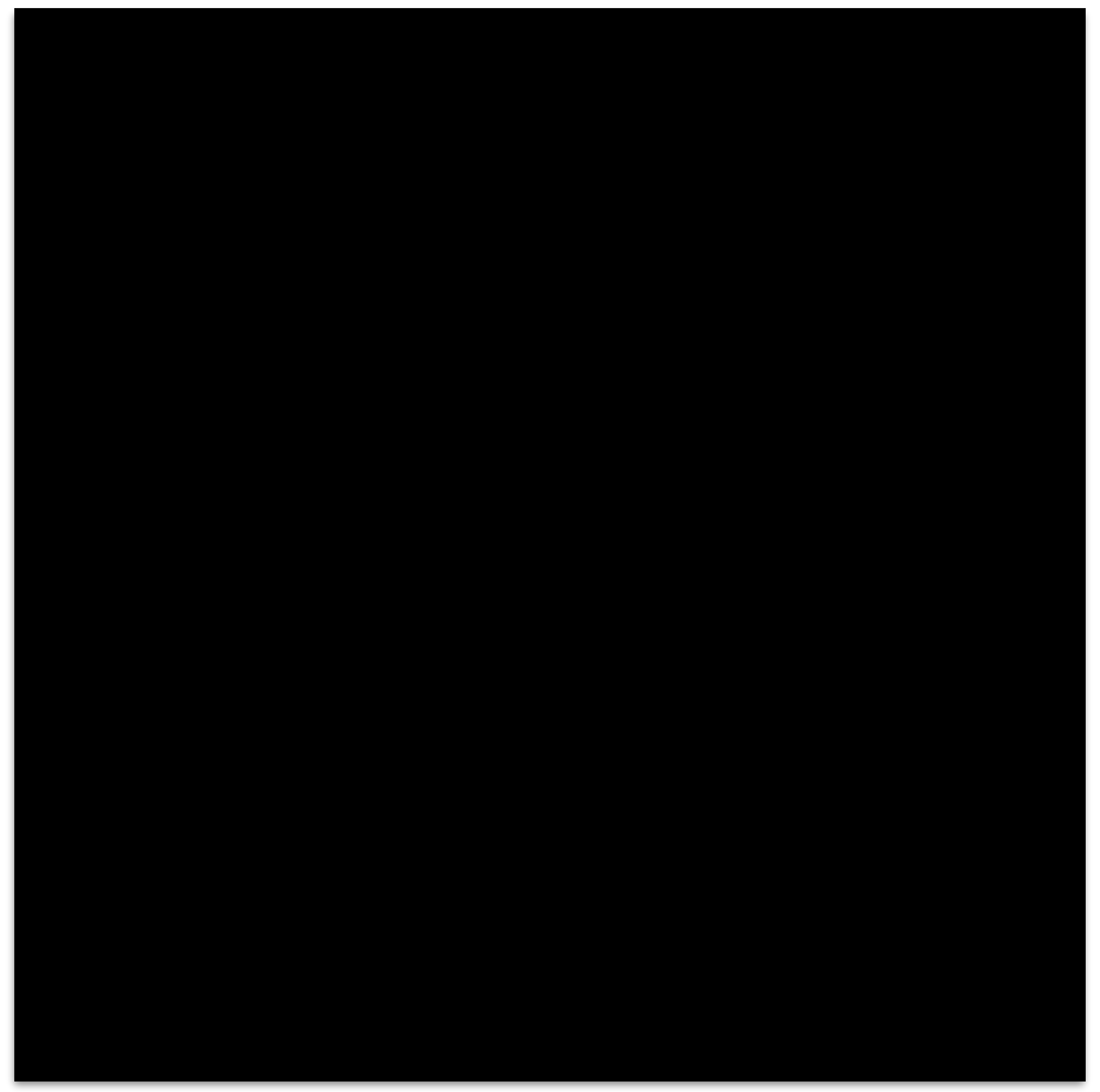
```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

```
$ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing psr/log (1.0.0)
  - Installing symfony/routing (v2.3.7)
  - Installing symfony/debug (v2.3.7)
  - Installing symfony/http-foundation (v2.3.7)
  - Installing symfony/event-dispatcher (v2.3.7)
  - Installing symfony/http-kernel (v2.3.7)
  - Installing pimple/pimple (v1.1.0)
  - Installing silex/silex (v1.1.2)
Writing lock file
Generating autoload files
$
```

The dependencies are installed into a directory named **vendor**

vendor =



By all means, learn about **vendor** and what happens in there, but don't obsess.

It isn't (usually) that important.

```
require “vendor/autoload.php”;
```

Autoloading

```
//
// life without autoloading
//

require “../vendor/whizbang/classes.php”;

$service = new Acme\WhizBang\Thing()
```

Autoloading your classes means you can just use them

```
# Acme\User is not defined

$user = new Acme\User();

# Acme\User is defined
```

__autoload

since 5

The __autoload function is called anytime a class does not exist

```php
function __autoload($class) {
    if ($class === "Acme\\Account\\User") {
        // do something to cause this class
        // to become defined
        require __DIR__."/src/user.inc";
    }
}
```

The major limitation of __autoload is that it is only one function

`spl_autoload_register`

since 5.1.2

With spl_autoload_register more than one autoloader implementation can be registered at the same time

```php
spl_autoload_register(
    // Registers Acme's autoloader
    Acme::autoloader
);

spl_autoload_register(
    // Registers Doctrine's autoloader
    Doctrine::autoloader
);

spl_autoload_register(function($class) {
    // Register a rarely used class
    if ($class === "Acme\\RarelyUsed\\User") {
       require __DIR__."/src/user.inc";
    }
});
```

```
// this allowed projects to ship autoloaders with
// their packages so they could be easily enabled

require “../vendor/whizbang/bootstrap.php”;
require “../vendor/awesomesoft/bootstrap.php”;
require “../vendor/lessawesome/bootstrapper.php”;
require “../vendor/ultraframework/classloader.php”;

LessAwesome\BootStrapper::register();

UltraFramework\ClassLoader::register(array(
    “\\Acme\\MyApp\\” = “../src”
));
```

Composer is a configurable autoloader

Each Composer package gets to configure its own rules

```
require “vendor/autoload.php”;
```

Pick an autoloading strategy and configure Composer to use it

psr-0

PHP-FIG

PHP Framework Interoperability Group

php-fig.com

"For sufficiently vague definitions of 'accepted', May 2009 is the date I use."

–Larry Garfield

Namespaces are directories

Classes are files with **.php** suffix

```
Acme\Account\User
```

```
Acme/Account/User.php
```

PSR-0 also supports legacy PEAR style naming conventions

```
Acme\Account\User

Acme_Account_User
```

```
Acme/Account/User.php

Acme/Account/User.php
```

Legacy rules are kinda convoluted

_ is converted to /

but only in the **class name**

```
Acme\Web_Site\User_Controller
```

```
Acme/Web_Site/User/Controller.php
```

PSR-0 had a handful of other relatively insignificant* issues

* the significance of the issues varies wildly depending on who you ask

```
{
  "autoload": {
    "psr-0": {
      "Acme\\Account\\": "src"
    }
  }
}

new Acme\Account\User();

# src/Acme/Account/User.php
```

```
{
  "autoload": {
    "psr-0": {
      "Acme_Account_": "src"
    }
  }
}

new Acme_Account_User();

# src/Acme/Account/User.php
```

psr-4

PHP-FIG

PHP Framework Interoperability Group

php-fig.com

Accepted December 3rd, 2013

Finally!

Very similar to PSR-0

Only supports namespaces
so no PEAR style naming

Introduces a namespace prefix and base directory for mapping

Reduces the number of directories that are required to exist

```
Acme\Account\User (class)

Acme\Account (namespace prefix)

src (base directory)

src/User.php (resulting file path)
```

```
{
  "autoload": {
    "psr-4": {
      "Acme\\Account\\": "src"
    }
  }
}

new Acme\Account\User();

# src/User.php
```

Composer currently recommends new projects use PSR-4

Migrate from PSR-0 to PSR-4

```
{
  "autoload": {
    "psr-0": {
      "Acme\\Account\\": "src"
    }
  }
}

new Acme\Account\User();

# src/Acme/Account/User.php
```

```
{
  "autoload": {
    "psr-4": {
      "Acme\\Account\\": "src/Acme/Account"
    }
  }
}

new Acme\Account\User();

# src/Acme/Account/User.php
```

files

Explicitly include specific files

```
{
  “autoload”: {
    “files”: [
      “src/foo.class.php”,
      “src/bar.class.php”
    ]
  }
}
```

```
{
  “autoload”: {
    “files”: [“src/functions.php”]
  }
}
```

```
{
  “autoload”: {
    “files”: [“src/autoload.php”]
  }
}
```

The `files` autoloader is really an **alwaysloader** `files` are included right when `vendor/autoload.php` is

classmap

A key => value map of class names to files on disk

It will look inside `.php` and `.inc` files to find classes

The classmap is generated anytime Composer dumps its autoloader

Extremely fast and powerful but **not** super developer friendly

```
{
  “autoload”: {
    “classmap”: [
      “src/includes/”,
      “resources/config.php”
    ]
  }
}
```

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

Versioning

Pretty much anything can be used as a Composer version

If you want to leverage Composer to its fullest use Semantic Versioning

Semantic Versioning

semver.org

MAJOR.MINOR.PATCH

Which number do you increment and why?

MAJOR.MINOR.PATCH

When you break backwards compatibility

MAJOR.**MINOR**.PATCH

When you add backwards compatible features

MAJOR.MINOR.**PATCH**

When you make backwards compatible bug fixes

Pre-Release Identifiers

Composer calls this "stability"

1.0.0-alpha

@alpha

1.0.0-beta.1

@beta

1.0.0-RC2

@RC

1.0.0

(stable)

Version Constraints

Exact Versions

1.0.2

Ranges

>=1.0.2,<2.0

Wildcards

1.0.*

Next Significant Release

Tilde Operator

Next Significant Release

~1.2

>=1.2,<2.0

Next Significant Release

~1.2.3

>=1.2.3,<1.3

Semantic Versioning let's you know what you are getting into

Safe

1.3.*

Only get bug fixes

Reasonably Safe

1.*

Get bug fixes and new features

Crazy sauce

*

Composer allows this, but don't.
Just dont.

Stability and the Root Package

Stability is controlled by the root package

Even if your package requires something @dev, users of your package won't get @dev unless they explicitly ask for it

The root package is defined in the working `composer.json`

```
{
  “require”: {
    “silex/silex”: “~1.1@dev”,
    “symfony/http-foundation”: “@beta”
  },
  “minimum-stability”: “alpha”
}
```

A package is only a root package when it is being developed

```
{
  “name”: “silex/silex”,
  “require”: {
    “pimple/pimple”: “1.*@dev”
  }
}
```

```
{
  “name”: “dflydev/doctrine-orm-service-provider”,
  “require”: {
    “pimple/pimple”: “1.*@beta”,
    “silex/silex”: “1.1.*”,
    “doctrine/orm”: “~2.3”
  }
}
```

```
{
  “name”: “silex/silex”,
  “require”: {
    “pimple/pimple”: “1.*@dev”
  }
}
```

```
{
  “require”: {
    “dflydev/doctrine-orm-service-provider”: “1.0.*”,

    “pimple/pimple”: “1.0.*”
  }
}
```

```
{
  “name”: “dflydev/doctrine-orm-service-provider”,
  “require”: {
    “pimple/pimple”: “1.*@beta”,
    “silex/silex”: “1.1.*”,
    “doctrine/orm”: “~2.3”
  }
}
```

```
{
  “name”: “silex/silex”,
  “require”: {
    “pimple/pimple”: “1.*@dev”
  }
}
```

Version Constraint Considerations

“If the dependency specifications are too tight, you are in danger of version lock (the inability to upgrade a package without having to release new versions of every dependent package).”

–Semantic Versioning

“If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable).”

–Semantic Versioning

Libraries should generally have more permissive constraints

End projects may want to have more restrictive constraints

VCS Repositories

Any VCS repository can be treated like a Composer package

GitHub

Composer treats tags as versions for VCS repositories

Tags and Versions

If a tag can be parsed as semver, awesome!

If it cannot be parsed as semver, it is treated as an "exact" version

v2.0.1

2.0.1

(2.0.*)

2.0.1

2.0.1

(2.0.*)

2.0.1-RC1

2.0.1-RC1

(2.0.*@RC)

2.0.1g

2.0.1g

(2.0.1g)

3.4-cuddly-cat

3.4-cuddly-cat

(3.4-cuddly-cat)

Branches and Versions

Composer treats branches as @dev stability versions

Numbered branches are treated as development versions

2.0

2.0.x-dev

(2.0.*@dev)

Named branches default to their name with a **dev-** prefix

master

dev-master

(dev-master)

testing

dev-testing

(dev-testing)

2.0-experimental

dev-2.0-experimental

(2.0.*@dev won't work!)

Named branches can be aliased to be semver friendly

```
{
  “extra”: {
    “branch-alias”: {
      “dev-master”: “2.0.x-dev”
    }
  }
}
```

master

dev-master / 2.0.x-dev

(dev-master **or** 2.0.*@dev)

dev-master considered harmful

“When starting a new library that is to be distributed via Packagist / Composer, be SURE to set up your dev-master branch alias.”

–Don Gilbert

https://twitter.com/dilbert4life/status/380137097614458881

Publishing and Discovery

Packagist

The PHP package archivist.

packagist.org

Publishing is as easy as pasting your repositories GitHub URL

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

Discovery is as easy as typing something into the search box

Over 23,600 packages

Check Packagist before you start a new library from scratch

Basic Usage

Install Composer

getcomposer.org/download

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

$ php composer.phar --version
```

WARNING

Security people think this is bad, but it is all the rage

```
$ chmod 755 composer.phar

$ mv composer.phar ~/bin/composer

$ composer --version
```

`composer.json` and `composer.lock`

`composer.json` describes a package and its dependencies

```
{
  “name”: “acme/my-project”,
  “description”: “Acme’s My Project”,
  “license”: “MIT”,
  “require”: {
    “silex/silex”: “1.1.*”
  },
  “autoload”: {
    “psr-4”: {
      “Acme\\MyProject\\”: “src”
    }
  }
}
```

`composer.lock` describes exactly what should be installed `composer.lock` is not meant for interactions with humans

Check your `composer.lock` into your repository

Common Commands

`$ composer install`

If `composer.lock` exists, install exactly what is in the lock file.

Otherwise, read `composer.json` to find out what should be installed, install the dependencies, and write out `composer.lock`.

`$ composer update`

Installs dependencies from `composer.json` and creates or updates `composer.lock`.

`$ composer require [pkg]`

Add a package to `composer.json`.

The `[pkg]` is the name with a version constraint.

`foo/bar:1.0.0` or `foo/bar=1.0.0` or `"foo/bar 1.0.0"`

```
$ composer diag
```

Check environment and `composer.json` for common errors

```
$ composer validate
```

Check `composer.json` for common errors

#composer

Autoload your classes

Use Semantic Versioning

Devs don't let devs dev-master

Search Packagist first

Publish your code on Packagist

Questions?

@beausimensen

ddd.io/ssp14-composer