Dependency Management with Composer

What are dependencies?

food
shelter
clothing
water
air
sleep

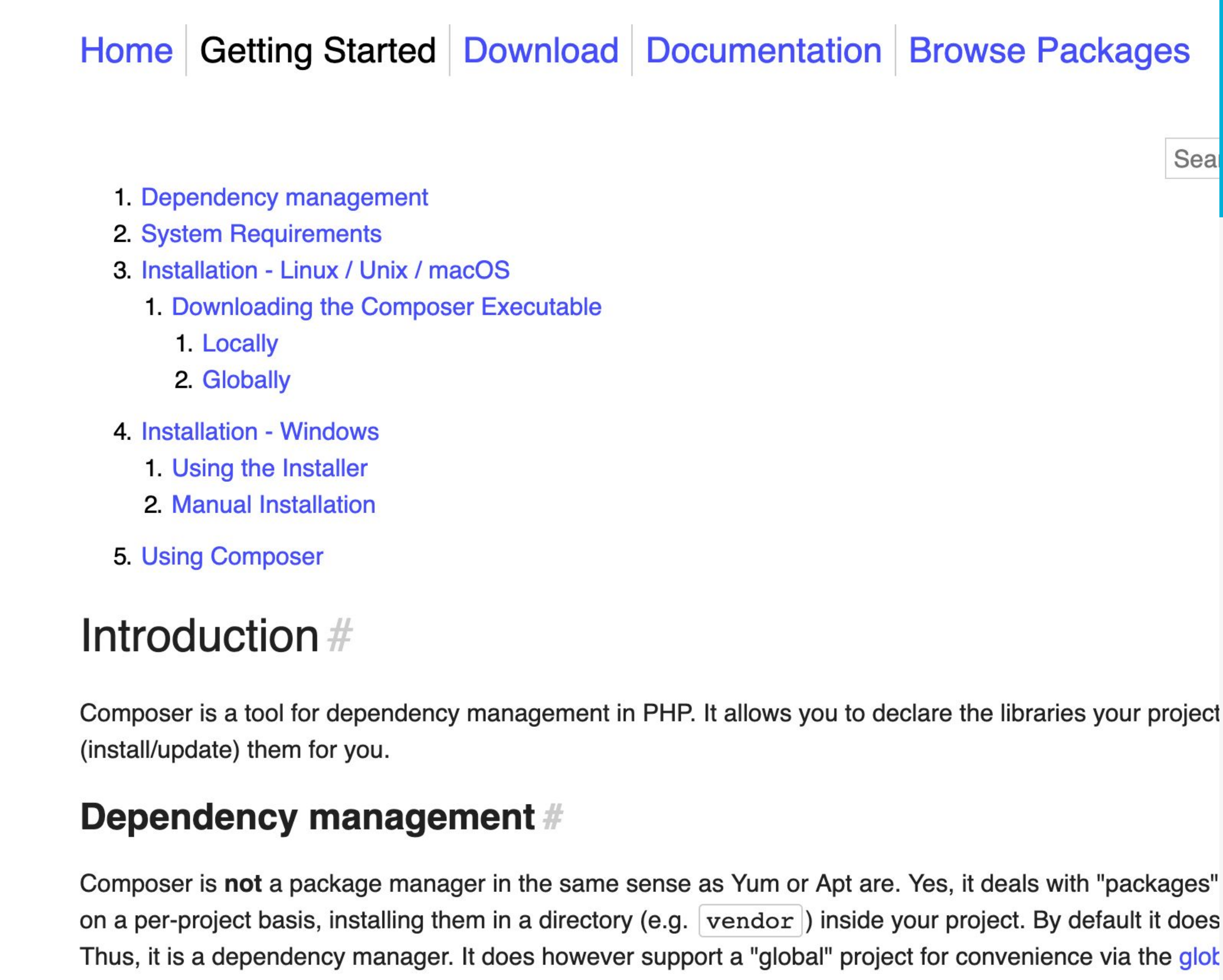
What is a Dependency Manager?

- Suppose:
  - You have a project that depends on a number of libraries.
  - Some of those libraries depend on other libraries.

- Dependency Manager:
  - Enables you to declare the libraries you depend on.
  - Finds out which versions of which packages can and need to be installed, and installs them (meaning it downloads them into your project).
  - Allows you to share specific package versions across environments and teams.

Meet Composer

getcomposer.org



Introduction

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project
(install/update) them for you.

Dependency management

Composer is **not** a package manager in the same sense as Yum or Apt are. Yes, it deals with "packages"
on a per-project basis, installing them in a directory (e.g. `vendor`) inside your project. By default it does
Thus, it is a dependency manager. It does however support a "global" project for convenience via the glob

getcomposer.org

Installation

- Make sure the command line is running the correct version of PHP
- Install Composer globally
  - Install the version of composer that matches your development environment
- Install packages per project
  - DO NOT commit installed packages to Version Control

Installation - Windows

Using the Installer#

This is the easiest way to get Composer set up on your machine.

Download and run Composer-Setup.exe. It will install the latest Composer version and set up your PATH so that you can call `composer` from any directory in your command line.

> **Note:** Close your current terminal. Test usage with a new terminal: This is important since the PATH only gets loaded when the terminal starts.

https://getcomposer.org/doc/00-intro.md#installation-windows

PHP in the Command Line

```
> which php
```

I want it to return the same php version I am running on my local development environment, in this case MAMP

```
/Applications/MAMP/bin/php/php7.4.2/bin/php
```

```
> vi ~/.bash_profile
```
(or vim)

Remove any reference to php path, and add

```
export PATH=/Applications/MAMP/bin/php/php7.4.2/bin:$PATH
```

Close and reopen your command line

Installation - Linux / Unix / macOS

https://getcomposer.org/download/ To quickly install Composer in the current directory, run the provided script. The 4 lines will, in order:

- ▷ Download the installer to the current directory
- ▷ Verify the installer SHA-384, which you can also cross-check here
- ▷ Run the installer
- ▷ Remove the installer

```
php -r "copy('https://getcomposer.org/installer',
'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'e0012edf3e80b6978849f5eff0d4b4e4c79ff1609dd1e613307e16318854d24ae64f2
6d17af3ef0bf7cfb710ca74755a') { echo 'Installer verified'; } else {
echo 'Installer corrupt'; unlink('composer-setup.php'); } echo
PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Global Access - Linux / Unix / macOS

You can place the Composer PHAR anywhere you wish. If you put it in a directory that is part of your PATH, you can access it globally. On Unix systems you can even make it executable and invoke it without directly using the php interpreter.

After running the installer you can run this to move composer.phar to a directory that is in your path:

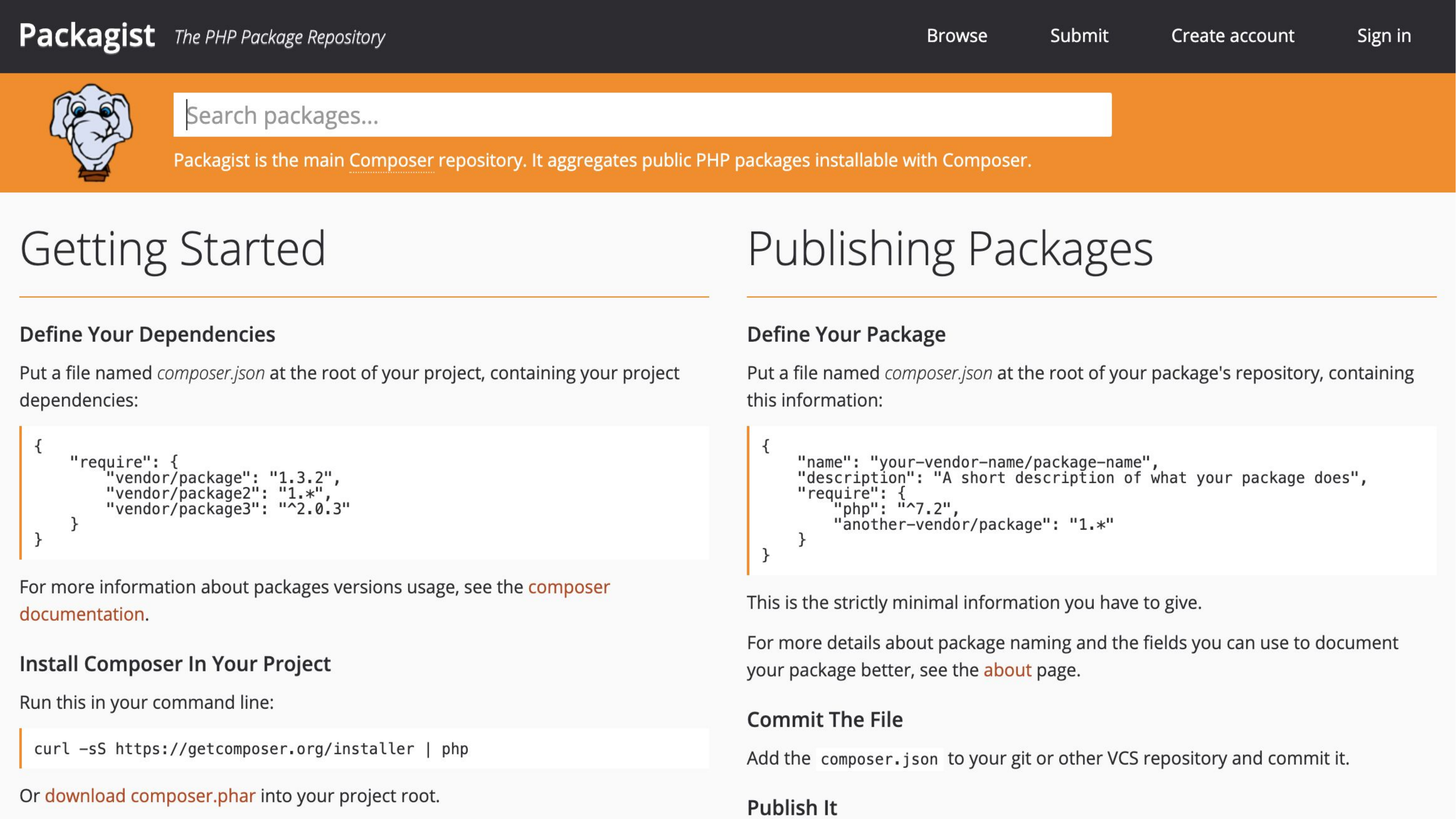
```
mv composer.phar ~/.local/bin/composer
```

Now run composer in order to run Composer instead of php composer.phar.

https://getcomposer.org/doc/00-intro.md#globally

Finding and Using Packages

Packagist: The PHP Package Repository



Packagist is the main Composer repository. It aggregates public PHP packages installable with Composer.

Getting Started

Define Your Dependencies

Put a file named *composer.json* at the root of your project, containing your project dependencies:

```
{
    "require": {
        "vendor/package": "1.3.2",
        "vendor/package2": "1.*",
        "vendor/package3": "^2.0.3"
    }
}
```

For more information about packages versions usage, see the composer documentation.

Install Composer In Your Project

Run this in your command line:

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

Or download composer.phar into your project root.

Publishing Packages

Define Your Package

Put a file named *composer.json* at the root of your package's repository, containing this information:

```
{
    "name": "your-vendor-name/package-name",
    "description": "A short description of what your package does",
    "require": {
        "php": "^7.2",
        "another-vendor/package": "1.*"
    }
}
```

This is the strictly minimal information you have to give.

For more details about package naming and the fields you can use to document your package better, see the about page.

Commit The File

Add the `composer.json` to your git or other VCS repository and commit it.

Publish It

Custom Path

```
{
  "extra": {
    "installer-paths": {
      "sites/example.com/modules/{$name}": ["vendor/package"]
    }
  }
}
```

Maintaining Versions

Semantic Versioning https://semver.org/

Given a version number MAJOR.MINOR.PATCH, increment the:

1. *MAJOR version when you make incompatible API changes*
2. *MINOR version when you add functionality that is backwards compatible*
3. *PATCH version when you make backwards compatible bug fixes*

”

Exact

- You can specify the exact version of a package. This will tell Composer to install this version and this version only. If other dependencies require a different version, the solver will ultimately fail and abort any install or update procedures.
- Example: 1.0.2

Range

- By using comparison operators you can specify ranges of valid versions. Valid operators are >, >=, <, <=, !=.
- You can define multiple ranges. Ranges separated by a space ( ) or comma (,) will be treated as a logical AND. A double pipe (||) will be treated as a logical OR. AND has higher precedence than OR.
- Note: Be careful when using unbounded ranges as you might end up unexpectedly installing versions that break backwards compatibility. Consider using the caret operator instead for safety.
- Examples:
    - >=1.0
    - >=1.0 <2.0
    - >=1.0 <1.1 || >=1.2*

Next Significant Release: Tilde

- The ~ operator is best explained by example: ~1.2 is equivalent to >=1.2 <2.0.0, while ~1.2.3 is equivalent to >=1.2.3 <1.3.0. As you can see it is mostly useful for projects respecting semantic versioning. A common usage would be to mark the minimum minor version you depend on, like ~1.2 (which allows anything up to, but not including, 2.0). Since in theory there should be no backwards compatibility breaks until 2.0, that works well. Another way of looking at it is that using ~ specifies a minimum version, but allows the last digit specified to go up.
- Example: ~1.2

Next Significant Release: Caret

- The ^ operator behaves very similarly but it sticks closer to semantic versioning, and will always allow non-breaking updates. For example ^1.2.3 is equivalent to >=1.2.3 <2.0.0 as none of the releases until 2.0 should break backwards compatibility. For pre-1.0 versions it also acts with safety in mind and treats ^0.3 as >=0.3.0 <0.4.0.
- This is the recommended operator for maximum interoperability when writing library code.
- Example: ^1.2.3

Testing Version Constraints

- There is a handy packagist semver checker (short for symantic versioning) that allows you to test out your version constraints.
    - https://semver.mwl.be/

Define Your Dependencies

```
{
    "require": {
        "vendor/package": "1.3.2",
        "vendor/package2": "1.*",
        "vendor/package3": "^2.0.3"
    }
}
```

Consistent Environments

composer.lock

```
{
  "_readme": [
    "This file locks the dependencies of your project to a known state",
    "Read more about it at https://getcomposer.org/doc/01-basic-usage.md#installing-dependencies",
    "This file is @generated automatically"
  ],
  "content-hash": xxxxx,
  "packages": [
    {
      "dist": {
        "type": "zip",
        "url": "https://api.github.com/repos/container-interop/container-interop/zipball/79cbf1341c22ec75643d841642dd5d6acd83bdb8",
        "reference": "79cbf1341c22ec75643d841642dd5d6acd83bdb8",
        "shasum": ""
      },
      "require": {
        "psr/container": "^1.0"

      },
      "type": "library",
      "autoload": {
        "psr-4": {
          "Interop\\Container\\": "src/Interop/Container/"
        }
      },
      "notification-url": "https://packagist.org/downloads/",
      "license": [
        "MIT"
      ],
      "description": "Promoting the interoperability of container objects (DIC, SL, etc.)",
      "homepage": "https://github.com/container-interop/container-interop",
      "time": "2017-02-14T19:40:03+00:00"
    }…
```

Autoloading

PSR-4, PSR-0, classmap and files

PSR-4

```
"autoload": {
    "psr-4": {
        “App\\": "src/",
        "Vendor\\Namespace\\": ""
    }
}
"autoload": {
    "psr-4": { "App\\": ["src/", "lib/"] } // same prefix in multiple directories
}
"autoload": {
    "psr-4": { "": "src/" } // fallback directory
}
```

PSR-0

```
"autoload": {
   "psr-0": {
      “App\\": "src/",
      "Vendor\\Namespace\\": "src/",
   }
}
"autoload": {
   "psr-0": { "App\\": ["src/", "lib/"] } // same prefix in multiple directories
}
"autoload": {
   “psr-0": { "": "src/" } // fallback directory
}
```

Classmap

```
"autoload": {
   "classmap": ["src/", "lib/", “src/Something.php"]
}
```

Files

```
"autoload": {
   "files": [“src/MyLibrary/functions.php”, “src/myfile.php”]
}
```

Storing and adding to the autoloader instance

Including the autoload.php file will also return the autoloader instance, so you can store the return value of the include call in a variable and add more namespaces.

This can be useful for autoloading classes in a test suite, for example.

```php
$loader = require __DIR__ . '/vendor/autoload.php';
$loader->addPsr4('Acme\\Test\\', __DIR__);
```

Creating your own Package

Every project is a package

Your Library Structure

- hello-world
    - src
        - hello-world
            - Greeting.php
    - composer.json
    - composer.lock

Greeting.php

```
<?php
namespace hello-world;
class Greeting {
    ...
}
```

Your Library: composer.json

```
{
    "name": “acme/hello-world”,
    "require": {
        "monolog/monolog": “1.0.*"
    }
}
```

Your Project: composer.json

```
{
  "name": "sketchings/blog",
  "repositories": [
    {
      "type": "vcs",
      "url": "https://github.com/sketchings/hello-world"
    }
  ],
  "require": {
    "acme/hello-world": ^0.1
  }
}
```

Questions and Answers

Thank you for your participation