Composer

The Composer Community

August 31, 2012

Contents

Chapter 1

Introduction

Composer is a tool for dependency management in PHP. It allows you to declare the dependent libraries your project needs and it will install them in your project for you.

1.1 Dependency management

Composer is not a package manager. Yes, it deals with “packages” or libraries, but it manages them on a per-project basis, installing them in a directory (e.g. `vendor`) inside your project. By default it will never install anything globally. Thus, it is a dependency manager.

This idea is not new and Composer is strongly inspired by node’s npm and ruby’s bundler. But there has not been such a tool for PHP.

The problem that Composer solves is this:

a) You have a project that depends on a number of libraries.

b) Some of those libraries depend on other libraries .

c) You declare the things you depend on

d) Composer finds out which versions of which packages need to be installed, and installs them (meaning it downloads them into your project).

1.2 Declaring dependencies

Let’s say you are creating a project, and you need a library that does logging. You decide to use monolog. In order to add it to your project, all you need to do is create a `composer.json` file which describes the project's dependencies.

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

We are simply stating that our project requires some `monolog/monolog` package, any version beginning with `1.0`.

1.3 Installation

1.3.1 Downloading the Composer Executable

Locally To actually get Composer, we need to do two things. The first one is installing Composer (again, this mean downloading it into your project):

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

This will just check a few PHP settings and then download `composer.phar` to your working directory. This file is the Composer binary. It is a PHAR (PHP archive), which is an archive format for PHP which can be run on the command line, amongst other things.

You can install Composer to a specific directory by using the `--install-dir` option and providing a target directory (it can be an absolute or relative path):

```
$ curl -s https://getcomposer.org/installer | php -- --install-
   dir=bin
```

Globally You can place this file anywhere you wish. If you put it in your `PATH`, you can access it globally. On unixy systems you can even make it executable and invoke it without `php`.

You can run these commands to easily access `composer` from anywhere on your system:

```
$ curl -s https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

Then, just run `composer` in order to run composer

1.3.2 Using Composer

Next, run the `install` command to resolve and download dependencies:

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

This will download monolog into the `vendor/monolog/monolog` directory.

1.4 Autoloading

Besides downloading the library, Composer also prepares an autoload file that's capable of autoloading all of the classes in any of the libraries that it downloads. To use it, just add the following line to your code's bootstrap process:

```
require 'vendor/autoload.php';
```

Woh! Now start using monolog! To keep learning more about Composer, keep reading the "Basic Usage" chapter.

Chapter 2

Basic usage

2.1 Installation

To install Composer, you just need to download the `composer.phar` executable.

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

For the details, see the Introduction chapter.

To check if Composer is working, just run the PHAR through `php`:

```
$ php composer.phar
```

This should give you a list of available commands.

> **Note:** You can also perform the checks only without downloading Composer by using the `--check` option. For more information, just use `--help`.
>
> ```
> $ curl -s http://getcomposer.org/installer | php -- --help
> ```

2.2 `composer.json`: Project Setup

To start using Composer in your project, all you need is a `composer.json` file. This file describes the dependencies of your project and may contain other metadata as well.

The JSON format is quite easy to write. It allows you to define nested structures.

2.2.1 The `require` Key

The first (and often only) thing you specify in `composer.json` is the `require` key. You're simply telling Composer which packages your project depends on.

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

As you can see, `require` takes an object that maps **package names** (e.g. `monolog/monolog`) to **package versions** (e.g. `1.0.*`).

2.2.2 Package Names

The package name consists of a vendor name and the project's name. Often these will be identical - the vendor name just exists to prevent naming clashes. It allows two different people to create a library named `json`, which would then just be named `igorw/json` and `seldaek/json`.

Here we are requiring `monolog/monolog`, so the vendor name is the same as the project's name. For projects with a unique name this is recommended. It also allows adding more related projects under the same namespace later on. If you are maintaining a library, this would make it really easy to split it up into smaller decoupled parts.

2.2.3 Package Versions

We are requiring version `1.0.*` of monolog. This means any version in the `1.0` development branch. It would match `1.0.0`, `1.0.2` or `1.0.20`.

Version constraints can be specified in a few different ways.

- **Exact version:** You can specify the exact version of a package, for example `1.0.2`. This is not used very often, but can be useful.
- **Range:** By using comparison operators you can specify ranges of valid versions. Valid operators are `>`, `>=`, `<`, `<=`, `!=`. An example range would be `>=1.0`. You can define multiple ranges, separated by a comma: `>=1.0,<2.0`.

- **Wildcard:** You can specify a pattern with a * wildcard. `1.0.*` is the equivalent of `>=1.0,<1.1-dev`.

2.3 Installing Dependencies

To fetch the defined dependencies into your local project, just run the `install` command of `composer.phar`.

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

This will find the latest version of `monolog/monolog` that matches the supplied version constraint and download it into the `vendor` directory. It's a convention to put third party code into a directory named `vendor`. In case of monolog it will put it into `vendor/monolog/-monolog`.

> **Tip:** If you are using git for your project, you probably want to add `vendor` into your `.gitignore`. You really don't want to add all of that code to your repository.

Another thing that the `install` command does is it adds a `composer.lock` file into your project root.

2.4 `composer.lock` - The Lock File

After installing the dependencies, Composer writes the list of the exact versions it installed into a `composer.lock` file. This locks the project to those specific versions.

Commit your application's `composer.lock` (along with `composer.json`) into version control.

This is important because the `install` command checks if a lock file is present, and if it is, it downloads the versions specified there (regardless of what `composer.json` says). This means that anyone who sets up the project will download the exact same version of the dependencies.

If no `composer.lock` file exists, Composer will read the dependencies and versions from `composer.json` and create the lock file.

This means that if any of the dependencies get a new version, you won't get the updates automatically. To update to the new version, use `update` command. This will fetch the latest matching versions (according to your `composer.json` file) and also update the lock file with the new version.

```
$ php composer.phar update
```

> **Note:** For libraries it is not necessarily recommended to commit the lock file, see also: Libraries - Lock file.

2.5 Packagist

Packagist is the main Composer repository. A Composer repository is basically a package source: a place where you can get packages from. Packagist aims to be the central repository that everybody uses. This means that you can automatically `require` any package that is available there.

If you go to the packagist website (packagist.org), you can browse and search for packages.

Any open source project using Composer should publish their packages on packagist. A library doesn't need to be on packagist to be used by Composer, but it makes life quite a bit simpler.

2.6 Autoloading

For libraries that specify autoload information, Composer generates a `vendor/autoload.php` file. You can simply include this file and you will get autoloading for free.

```
require 'vendor/autoload.php';
```

This makes it really easy to use third party code. For example: If your project depends on monolog, you can just start using classes from it, and they will be autoloaded.

```
$log = new Monolog\Logger('name');
$log->pushHandler(new Monolog\Handler\StreamHandler('app.log',
    Monolog\Logger::WARNING));

$log->addWarning('Foo');
```

You can even add your own code to the autoloader by adding an `autoload` field to `composer.json`.

```
{
    "autoload": {
        "psr-0": {"Acme": "src/"}
    }
}
```

Composer will register a PSR-0 autoloader for the `Acme` namespace.

You define a mapping from namespaces to directories. The `src` directory would be in your project root, on the same level as `vendor` directory is. An example filename would be `src/Acme/Foo.php` containing an `Acme\Foo` class.

After adding the `autoload` field, you have to re-run `install` to re-generate the `vendor/autoload.php` file.

Including that 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.

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

In addition to PSR-0 autoloading, classmap is also supported. This allows classes to be autoloaded even if they do not conform to PSR-0. See the autoload reference for more details.

> **Note:** Composer provides its own autoloader. If you don't want to use that one, you can just include `vendor/composer/autoload_namespaces.php`, which returns an associative array mapping namespaces to directories.

Chapter 3

Libraries

This chapter will tell you how to make your library installable through composer.

3.1 Every project is a package

As soon as you have a `composer.json` in a directory, that directory is a package. When you add a `require` to a project, you are making a package that depends on other packages. The only difference between your project and libraries is that your project is a package without a name.

In order to make that package installable you need to give it a name. You do this by adding a `name` to `composer.json`:

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

In this case the project name is `acme/hello-world`, where `acme` is the vendor name. Supplying a vendor name is mandatory.

> **Note:** If you don't know what to use as a vendor name, your GitHub username is usually a good bet. While package names are case insensitive, the convention is all lowercase and dashes for word separation.

3.2 Specifying the version

You need to specify the package's version some way. When you publish your package on Packagist, it is able to infer the version from the VCS (git, svn, hg) information, so in that case you do not have to specify it, and it is recommended not to. See tags and branches to see how version numbers are extracted from these.

If you are creating packages by hand and really have to specify it explicitly, you can just add a `version` field:

```
{
    "version": "1.0.0"
}
```

3.2.1 Tags

For every tag that looks like a version, a package version of that tag will be created. It should match 'X.Y.Z' or 'vX.Y.Z', with an optional suffix for RC, beta, alpha or patch.

Here are a few examples of valid tag names:

```
1.0.0
v1.0.0
1.10.5-RC1
v4.4.4beta2
v2.0.0-alpha
v2.0.4-p1
```

> **Note:** If you specify an explicit version in `composer.json`, the tag name must match the specified version.

3.2.2 Branches

For every branch, a package development version will be created. If the branch name looks like a version, the version will be `{branchname}-dev`. For example a branch `2.0` will get a version `2.0.x-dev` (the `.x` is added for technical reasons, to make sure it is recognized as a branch, a `2.0.x` branch would also be valid and be turned into `2.0.x-dev` as well. If the branch does not look like a version, it will be `dev-{branchname}`. `master` results in a `dev-master` version.

Here are some examples of version branch names:

```
1.x
1.0 (equals 1.0.x)
1.1.x
```

> **Note:** When you install a dev version, it will install it from source.

3.2.3 Aliases

It is possible alias branch names to versions. For example, you could alias `dev-master` to `1.0.x-dev`, which would allow you to require `1.0.x-dev` in all the packages.

See Aliases for more information.

3.3 Lock file

For your library you may commit the `composer.lock` file if you want to. This can help your team to always test against the same dependency versions. However, this lock file will not have any effect on other projects that depend on it. It only has an effect on the main project.

If you do not want to commit the lock file and you are using git, add it to the `.gitignore`.

3.4 Light-weight distribution packages

Including the tests and other useless information like .travis.yml in distributed packages is not a good idea.

The `.gitattributes` file is a git specific file like `.gitignore` also living at the root directory of your library. It overrides local and global configuration (`.git/config` and `~/.gitconfig` respectively) when present and tracked by git.

Use `.gitattributes` to prevent unwanted files from bloating the zip distribution packages.

```
// .gitattributes
Tests/ export-ignore
phpunit.xml.dist export-ignore
Resources/doc/ export-ignore
.travis.yml export-ignore
```

Test it by inspecting the zip file generated manually:

```
git archive branchName --format zip -o file.zip
```

> **Note:** files would be still tracked by git just not included in the distribution. This will only work for GitHub packages installed from dist (i.e. tagged releases) for now.

3.5 Publishing to a VCS

Once you have a vcs repository (version control system, e.g. git) containing a `composer.json` file, your library is already composer-installable. In this example we will publish the `acme/hello-world` library on GitHub under `github.com/composer/hello-world`.

Now, To test installing the `acme/hello-world` package, we create a new project locally. We will call it `acme/blog`. This blog will depend on `acme/hello-world`, which in turn depends on `monolog/monolog`. We can accomplish this by creating a new `blog` directory somewhere, containing a `composer.json`:

```
{
    "name": "acme/blog",
    "require": {
        "acme/hello-world": "dev-master"
    }
}
```

The name is not needed in this case, since we don't want to publish the blog as a library. It is added here to clarify which `composer.json` is being described.

Now we need to tell the blog app where to find the `hello-world` dependency. We do this by adding a package repository specification to the blog's `composer.json`:

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

For more details on how package repositories work and what other types are available, see Repositories.

That's all. You can now install the dependencies by running composer's `install` command!

Recap: Any git/svn/hg repository containing a `composer.json` can be added to your project by specifying the package repository and declaring the dependency in the `require` field.

3.6 Publishing to packagist

Alright, so now you can publish packages. But specifying the vcs repository every time is cumbersome. You don't want to force all your users to do that.

The other thing that you may have noticed is that we did not specify a package repository for `monolog/monolog`. How did that work? The answer is packagist.

Packagist is the main package repository for composer, and it is enabled by default. Anything that is published on packagist is available automatically through composer. Since monolog is on packagist, we can depend on it without having to specify any additional repositories.

If we wanted to share `hello-world` with the world, we would publish it on packagist as well. Doing so is really easy.

You simply hit the big "Submit Package" button and sign up. Then you submit the URL to your VCS repository, at which point packagist will start crawling it. Once it is done, your package will be available to anyone.

Chapter 4

Command-line interface

You've already learned how to use the command-line interface to do some things. This chapter documents all the available commands.

4.1 init

In the Libraries chapter we looked at how to create a `composer.json` by hand. There is also an `init` command available that makes it a bit easier to do this.

When you run the command it will interactively ask you to fill in the fields, while using some smart defaults.

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

4.1.1 Options

- **–no-interaction:** (**-n**) Run the command in non-interactive mode. The rest of these options only make sense when you are in this mode.
- **–name:** Name of the package.
- **–description:** Description of the package.
- **–author:** Author name of the package.
- **–homepage:** Homepage of the package.
- **–require:** Package to require with a version constraint. Should be in format `foo/bar:1.0.0`.
- **–require-dev:** Development requirements, see **–require**.

4.2 install

The `install` command reads the `composer.json` file from the current directory, resolves the dependencies, and installs them into `vendor`.

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

If there is a `composer.lock` file in the current directory, it will use the exact versions from there instead of resolving them. This ensures that everyone using the library will get the same versions of the dependencies.

If there is no `composer.lock` file, composer will create one after dependency resolution.

4.2.1 Options

- **–prefer-source:** There are two ways of downloading a package: `source` and `dist`. For stable versions composer will use the `dist` by default. The `source` is a version control repository. If `--prefer-source` is enabled, composer will install from `source` if there is one. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly.
- **–dry-run:** If you want to run through an installation without actually installing a package, you can use `--dry-run`. This will simulate the installation and show you what would happen.
- **–dev:** By default composer will only install required packages. By passing this option you can also make it install packages referenced by `require-dev`.
- **–no-scripts:** Skips execution of scripts defined in `composer.json`.

4.3 update

In order to get the latest versions of the dependencies and to update the `composer.lock` file, you should use the `update` command.

```
$ php composer.phar update
```

This will resolve all dependencies of the project and write the exact versions into `composer.lock`.

If you just want to update a few packages and not all, you can list them as such:

```
$ php composer.phar update vendor/package vendor/package2
```

4.3.1 Options

- **–prefer-source:** Install packages from `source` when available.
- **–dry-run:** Simulate the command without actually doing anything.
- **–dev:** Install packages listed in `require-dev`.
- **–no-scripts:** Skips execution of scripts defined in `composer.json`.

4.4 require

The `require` command adds new packages to the `composer.json` file from the current directory.

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

After adding/changing the requirements, the modified requirements will be installed or updated.

If you do not want to choose requirements interactively, you can just pass them to the command.

```
$ php composer.phar require vendor/package:2.* vendor/package2:
    dev-master
```

4.4.1 Options

- **–prefer-source:** Install packages from `source` when available.
- **–dev:** Add packages to `require-dev`.

4.5 search

The search command allows you to search through the current project's package repositories. Usually this will be just packagist. You simply pass it the terms you want to search for.

```
$ php composer.phar search monolog
```

You can also search for more than one term by passing multiple arguments.

4.6 show

To list all of the available packages, you can use the `show` command.

```
$ php composer.phar show
```

If you want to see the details of a certain package, you can pass the package name.

```
$ php composer.phar show monolog/monolog

name     : monolog/monolog
versions : master-dev, 1.0.2, 1.0.1, 1.0.0, 1.0.0-RC1
type     : library
names    : monolog/monolog
source   : [git] http://github.com/Seldaek/monolog.git 3
   d4e60d0cbc4b888fe5ad223d77964428b1978da
dist     : [zip] http://github.com/Seldaek/monolog/zipball/3
   d4e60d0cbc4b888fe5ad223d77964428b1978da 3
   d4e60d0cbc4b888fe5ad223d77964428b1978da
license  : MIT

autoload
psr-0
Monolog : src/

requires
php >=5.3.0
```

You can even pass the package version, which will tell you the details of that specific version.

```
$ php composer.phar show monolog/monolog 1.0.2
```

4.6.1 Options

- **–installed:** Will list the packages that are installed.
- **–platform:** Will list only platform packages (php & extensions).

4.7 depends

The `depends` command tells you which other packages depend on a certain package. You can specify which link types (`require`, `require-dev`) should be included in the listing. By default both are used.

```
$ php composer.phar depends --link-type=require monolog/monolog

nrk/monolog-fluent
poc/poc
propel/propel
symfony/monolog-bridge
symfony/symfony
```

4.7.1 Options

- **–link-type:** The link types to match on, can be specified multiple times.

4.8 validate

You should always run the `validate` command before you commit your `composer.json` file, and before you tag a release. It will check if your `composer.json` is valid.

```
$ php composer.phar validate
```

4.9 self-update

To update composer itself to the latest version, just run the `self-update` command. It will replace your `composer.phar` with the latest version.

```
$ php composer.phar self-update
```

If you have installed composer for your entire system (see global installation), you have to run the command with `root` privileges

```
$ sudo composer self-update
```

4.10 create-project

You can use Composer to create new projects from an existing package. There are several applications for this:

1. You can deploy application packages.
2. You can check out any package and start developing on patches for example.
3. Projects with multiple developers can use this feature to bootstrap the initial application for development.

To create a new project using composer you can use the "create-project" command. Pass it a package name, and the directory to create the project in. You can also provide a version as third argument, otherwise the latest version is used.

The directory is not allowed to exist, it will be created during installation.

```
php composer.phar create-project doctrine/orm path 2.2.0
```

By default the command checks for the packages on packagist.org.

4.10.1 Options

- **–repository-url:** Provide a custom repository to search for the package, which will be used instead of packagist. Can be either an HTTP URL pointing to a `composer` repository, or a path to a local `packages.json` file.
- **–prefer-source:** Get a development version of the code checked out from version control.
- **–dev:** Install packages listed in `require-dev`.

4.11 dump-autoload

If you need to update the autoloader because of new classes in a classmap package for example, you can use "dump-autoload" to do that without having to go through an install or update.

Additionally, it can dump an optimized autoloader that converts PSR-0 packages into classmap ones for performance reasons. In large applications with many classes, the autoloader can take up a substantial portion of every request's time. Using classmaps for everything is less convenient in development, but using this option you can still use PSR-0 for convenience and classmaps for performance.

4.11.1 Options

- **–optimize:** Convert PSR-0 autoloading to classmap to get a faster autoloader. This is recommended especially for production, but can take a bit of time to run so it is currently not done by default.

4.12 help

To get more information about a certain command, just use `help`.

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

4.13 Environment variables

You can set a number of environment variables that override certain settings. Whenever possible it is recommended to specify these settings in the `config` section of `composer.json` instead. It is worth noting that that the env vars will always take precedence over the values specified in `composer.json`.

4.13.1 COMPOSER

By setting the `COMPOSER` env variable it is possible to set the filename of `composer.json` to something else.

For example:

```
$ COMPOSER=composer-other.json php composer.phar install
```

4.13.2 COMPOSER_ROOT_VERSION

By setting this var you can specify the version of the root package, if it can not be guessed from VCS info and is not present in `composer.json`.

4.13.3 COMPOSER_VENDOR_DIR

By setting this var you can make composer install the dependencies into a directory other than `vendor`.

4.13.4 COMPOSER_BIN_DIR

By setting this option you can change the `bin` (Vendor Bins) directory to something other than `vendor/bin`.

4.13.5 http_proxy or HTTP_PROXY

If you are using composer from behind an HTTP proxy, you can use the standard `http_proxy` or `HTTP_PROXY` env vars. Simply set it to the URL of your proxy. Many operating systems already set this variable for you.

Using `http_proxy` (lowercased) or even defining both might be preferable since some tools like git or curl will only use the lower-cased `http_proxy` version. Alternatively you can also define the git proxy using `git config --global http.proxy <proxy url>`.

4.13.6 COMPOSER_HOME

The `COMPOSER_HOME` var allows you to change the composer home directory. This is a hidden, global (per-user on the machine) directory that is shared between all projects.

By default it points to `/home/<user>/.composer` on *nix, `/Users/<user>/.composer` on OSX and `C:\Users\<user>\AppData\Roaming\Composer` on Windows.

COMPOSER_HOME/config.json You may put a `config.json` file into the location which `COMPOSER_HOME` points to. Composer will merge this configuration with your project's `composer.json` when you run the `install` and `update` commands.

This file allows you to set configuration and repositories for the user's projects.

In case global configuration matches *local* configuration, the *local* configuration in the project's `composer.json` always wins.

4.13.7 COMPOSER_PROCESS_TIMEOUT

This env var controls the time composer waits for commands (such as git commands) to finish executing. The default value is 300 seconds (5 minutes).

Chapter 5

composer.json

This chapter will explain all of the fields available in `composer.json`.

5.1 JSON schema

We have a JSON schema that documents the format and can also be used to validate your `composer.json`. In fact, it is used by the `validate` command. You can find it at: `res/composer-schema.json`.

5.2 Root Package

The root package is the package defined by the `composer.json` at the root of your project. It is the main `composer.json` that defines your project requirements.

Certain fields only apply when in the root package context. One example of this is the `config` field. Only the root package can define configuration. The config of dependencies is ignored. This makes the `config` field `root-only`.

If you clone one of those dependencies to work on it, then that package is the root package. The `composer.json` is identical, but the context is different.

> **Note:** A package can be the root package or not, depending on the context. For example, if your project depends on the `monolog` library, your project is the root package. However, if you clone `monolog` from GitHub in order to fix a bug in it, then `monolog` is the root package.

5.3 Properties

5.3.1 name

The name of the package. It consists of vendor name and project name, separated by `/`.

Examples:

- monolog/monolog
- igorw/event-source

Required for published packages (libraries).

5.3.2 description

A short description of the package. Usually this is just one line long.

Required for published packages (libraries).

5.3.3 version

The version of the package.

This must follow the format of `X.Y.Z` with an optional suffix of `-dev`, `alphaN`, `-betaN` or `-RCN`.

Examples:

```
1.0.0
1.0.2
1.1.0
0.2.5
1.0.0-dev
1.0.0-beta2
1.0.0-RC5
```

Optional if the package repository can infer the version from somewhere, such as the VCS tag name in the VCS repository. In that case it is also recommended to omit it.

> **Note:** Packagist uses VCS repositories, so the statement above is very much true for Packagist as well. Specifying the version yourself will most likely end up creating problems at some point due to human error.

5.3.4 type

The type of the package. It defaults to `library`.

Package types are used for custom installation logic. If you have a package that needs some special logic, you can define a custom type. This could be a `symfony-bundle`, a `wordpress-plugin` or a `typo3-module`. These types will all be specific to certain projects, and they will need to provide an installer capable of installing packages of that type.

Out of the box, composer supports three types:

- **library:** This is the default. It will simply copy the files to `vendor`.
- **metapackage:** An empty package that contains requirements and will trigger their installation, but contains no files and will not write anything to the filesystem. As such, it does not require a dist or source key to be installable.
- **composer-installer:** A package of type `composer-installer` provides an installer for other packages that have a custom type. Read more in the dedicated article.

Only use a custom type if you need custom logic during installation. It is recommended to omit this field and have it just default to `library`.

5.3.5 keywords

An array of keywords that the package is related to. These can be used for searching and filtering.

Examples:

```
logging
events
database
redis
templating
```

Optional.

5.3.6 homepage

An URL to the website of the project.

Optional.

5.3.7 time

Release date of the version.

Must be in `YYYY-MM-DD` or `YYYY-MM-DD HH:MM:SS` format.

Optional.

5.3.8 license

The license of the package. This can be either a string or an array of strings.

The recommended notation for the most common licenses is (alphabetical):

```
Apache-2.0
BSD-2-Clause
BSD-3-Clause
BSD-4-Clause
GPL-2.0
GPL-2.0+
GPL-3.0
GPL-3.0+
LGPL-2.1
LGPL-2.1+
LGPL-3.0
LGPL-3.0+
MIT
```

Optional, but it is highly recommended to supply this. More identifiers are listed at the SPDX Open Source License Registry.

An Example:

```
{
    "license": "MIT"
}
```

For a package, when there is a choice between licenses ("disjunctive license"), multiple can be specified as array.

An Example for disjunctive licenses:

```
{
    "license": [
       "LGPL-2.1",
       "GPL-3.0+"
    ]
}
```

Alternatively they can be separated with "or" and enclosed in parenthesis;

```
{
    "license": "(LGPL-2.1 or GPL-3.0+)"
}
```

Similarly when multiple licenses need to be applied ("conjunctive license"), they should be separated with "and" and enclosed in parenthesis.

5.3.9 authors

The authors of the package. This is an array of objects.

Each author object can have following properties:

- **name:** The author's name. Usually his real name.
- **email:** The author's email address.
- **homepage:** An URL to the author's website.
- **role:** The authors' role in the project (e.g. developer or translator)

An example:

```
{
    "authors": [
        {
            "name": "Nils Adermann",
            "email": "naderman@naderman.de",
            "homepage": "http://www.naderman.de",
            "role": "Developer"
        },
        {
            "name": "Jordi Boggiano",
            "email": "j.boggiano@seld.be",
            "homepage": "http://seld.be",
            "role": "Developer"
        }
    ]
}
```

Optional, but highly recommended.

5.3.10 support

Various information to get support about the project.

Support information includes the following:

- **email:** Email address for support.
- **issues:** URL to the Issue Tracker.
- **forum:** URL to the Forum.
- **wiki:** URL to the Wiki.
- **irc:** IRC channel for support, as irc://server/channel.
- **source:** URL to browse or download the sources.

An example:

```
{
    "support": {
        "email": "support@example.org",
        "irc": "irc://irc.freenode.org/composer"
    }
}
```

Optional.

5.3.11 Package links

All of the following take an object which maps package names to version constraints.

Example:

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

All links are optional fields.

`require` and `require-dev` additionally support stability flags (root-only). These allow you to further restrict or expand the stability of a package beyond the scope of the minimum-stability setting. You can apply them to a constraint, or just apply them to an empty constraint if you want to allow unstable packages of a dependency's dependency for example.

Example:

```
{
    "require": {
        "monolog/monolog": "1.0.*@beta",
        "acme/foo": "@dev"
    }
}
```

`require` and `require-dev` additionally support explicit references (i.e. commit) for dev versions to make sure they are blocked to a given state, even when you run update. These only work if you explicitly require a dev version and append the reference with `#<ref>`. Note that while this is convenient at times, it should not really be how you use packages in the long term. You should always try to switch to tagged releases as soon as you can, especially if the project you work on will not be touched for a while.

Example:

```
{
    "require": {
        "monolog/monolog": "dev-master#2
           eb0c0978d290a1c45346a1955188929cb4e5db7",
        "acme/foo": "1.0.x-dev#abc123"
    }
}
```

require Lists packages required by this package. The package will not be installed unless those requirements can be met.

require-dev (root-only) Lists packages required for developing this package, or running tests, etc. The dev requirements of the root package only will be installed if `install` or `update` is ran with `--dev`.

Packages listed here and their dependencies can not overrule the resolution found with the packages listed in require. This is even true if a different version of a package would be installable and solve the conflict. The reason is that `install --dev` produces the exact same state as just `install`, apart from the additional dev packages.

If you run into such a conflict, you can specify the conflicting package in the require section and require the right version number to resolve the conflict.

conflict Lists packages that conflict with this version of this package. They will not be allowed to be installed together with your package.

replace Lists packages that are replaced by this package. This allows you to fork a package, publish it under a different name with its own version numbers, while packages requiring the original package continue to work with your fork because it replaces the original package.

This is also useful for packages that contain sub-packages, for example the main symfony/symfony package contains all the Symfony Components which are also available as individual packages. If you require the main package it will automatically fulfill any requirement of one of the individual components, since it replaces them.

Caution is advised when using replace for the sub-package purpose explained above. You should then typically only replace using `self.version` as a version constraint, to make sure the main package only replaces the sub-packages of that exact version, and not any other version, which would be incorrect.

provide List of other packages that are provided by this package. This is mostly useful for common interfaces. A package could depend on some virtual `logger` package, any library that implements this logger interface would simply list it in `provide`.

5.3.12 suggest

Suggested packages that can enhance or work well with this package. These are just informational and are displayed after the package is installed, to give your users a hint that they could add more packages, even though they are not strictly required.

The format is like package links above, except that the values are free text and not version constraints.

Example:

```
{
    "suggest": {
        "monolog/monolog": "Allows more advanced logging of the
            application flow"
    }
}
```

5.3.13 autoload

Autoload mapping for a PHP autoloader.

Currently PSR-0 autoloading, classmap generation and files are supported. PSR-0 is the recommended way though since it offers greater flexibility (no need to regenerate the autoloader when you add classes).

Under the `psr-0` key you define a mapping from namespaces to paths, relative to the package root. Note that this also supports the PEAR-style non-namespaced convention.

The PSR-0 references are all combined, during install/update, into a single key => value array which may be found in the generated file `vendor/composer/autoload_namespaces.php`.

Example:

```
{
    "autoload": {
        "psr-0": {
            "Monolog": "src/",
            "Vendor\\Namespace": "src/",
            "Pear_Style": "src/"
        }
    }
}
```

If you need to search for a same prefix in multiple directories, you can specify them as an array as such:

```
{
    "autoload": {
        "psr-0": { "Monolog": ["src/", "lib/"] }
    }
}
```

The PSR-0 style is not limited to namespace declarations only but may be specified right down to the class level. This can be useful for libraries with only one class in the global namespace. If the php source file is also located in the root of the package, for example, it may be declared like this:

```
{
    "autoload": {
        "psr-0": { "UniqueGlobalClass": "" }
    }
}
```

If you want to have a fallback directory where any namespace can be, you can use an empty prefix like:

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

The `classmap` references are all combined, during install/update, into a single key => value array which may be found in the generated file `vendor/composer/autoload_classmap.php`.

You can use the classmap generation support to define autoloading for all libraries that do not follow PSR-0. To configure this you specify all directories or files to search for classes.

Example:

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

If you want to require certain files explicitly on every request then you can use the 'files' autoloading mechanism. This is useful if your package includes PHP functions that cannot be autoloaded by PHP.

Example:

```
{
    "autoload": {
        "files": ["src/MyLibrary/functions.php"]
    }
}
```

5.3.14 include-path

> **DEPRECATED**: This is only present to support legacy projects, and all new code should preferably use autoloading. As such it is a deprecated practice, but the feature itself will not likely disappear from Composer.

A list of paths which should get appended to PHP's `include_path`.

Example:

```
{
    "include-path": ["lib/"]
}
```

Optional.

5.3.15 target-dir

Defines the installation target.

In case the package root is below the namespace declaration you cannot autoload properly. `target-dir` solves this problem.

An example is Symfony. There are individual packages for the components. The Yaml component is under `Symfony\Component\Yaml`. The package root is that `Yaml` directory. To make autoloading possible, we need to make sure that it is not installed into `vendor/symfony/yaml`, but instead into `vendor/symfony/yaml/Symfony/Component/Yaml`, so that the autoloader can load it from `vendor/symfony/yaml`.

To do that, `autoload` and `target-dir` are defined as follows:

```
{
    "autoload": {
        "psr-0": { "Symfony\\Component\\Yaml": "" }
    },
    "target-dir": "Symfony/Component/Yaml"
}
```

Optional.

5.3.16 minimum-stability (root-only)

This defines the default behavior for filtering packages by stability. This defaults to `stable`, so if you rely on a `dev` package, you should specify it in your file to avoid surprises.

All versions of each package are checked for stability, and those that are less stable than the `minimum-stability` setting will be ignored when resolving your project dependencies. Specific changes to the stability requirements of a given package can be done in `require` or `require-dev` (see package links).

Available options are `dev`, `alpha`, `beta`, `RC`, and `stable`.

5.3.17 repositories (root-only)

Custom package repositories to use.

By default composer just uses the packagist repository. By specifying repositories you can get packages from elsewhere.

Repositories are not resolved recursively. You can only add them to your main `composer.json`. Repository declarations of dependencies' `composer.json`s are ignored.

The following repository types are supported:

- **composer:** A composer repository is simply a `packages.json` file served via HTTP, that contains a list of `composer.json` objects with additional `dist` and/or `source` information.

- **vcs:** The version control system repository can fetch packages from git, svn and hg repositories.
- **pear:** With this you can import any pear repository into your composer project.
- **package:** If you depend on a project that does not have any support for composer whatsoever you can define the package inline using a `package` repository. You basically just inline the `composer.json` object.

For more information on any of these, see Repositories.

Example:

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "http://packages.example.com"
        },
        {
            "type": "vcs",
            "url": "https://github.com/Seldaek/monolog"
        },
        {
            "type": "pear",
            "url": "http://pear2.php.net"
        },
        {
            "type": "package",
            "package": {
                "name": "smarty/smarty",
                "version": "3.1.7",
                "dist": {
                    "url": "http://www.smarty.net/files/Smarty
                       -3.1.7.zip",
                    "type": "zip"
                },
                "source": {
                    "url": "http://smarty-php.googlecode.com/
                       svn/",
                    "type": "svn",
                    "reference": "tags/Smarty_3_1_7/
                       distribution/"
                }
            }
        }
    ]
}
```

Note: Order is significant here. When looking for a package, Composer will look from the first to the last repository, and pick the first match. By default Packagist is added last which means that custom repositories can override packages from it.

5.3.18 config (root-only)

A set of configuration options. It is only used for projects.

The following options are supported:

- **vendor-dir:** Defaults to `vendor`. You can install dependencies into a different directory if you want to.
- **bin-dir:** Defaults to `vendor/bin`. If a project includes binaries, they will be symlinked into this directory.
- **process-timeout:** Defaults to `300`. The duration processes like git clones can run before Composer assumes they died out. You may need to make this higher if you have a slow connection or huge vendors.
- **notify-on-install:** Defaults to `true`. Composer allows repositories to define a notification URL, so that they get notified whenever a package from that repository is installed. This option allows you to disable that behaviour.

Example:

```
{
    "config": {
        "bin-dir": "bin"
    }
}
```

5.3.19 scripts (root-only)

Composer allows you to hook into various parts of the installation process through the use of scripts.

See Scripts for events details and examples.

5.3.20 extra

Arbitrary extra data for consumption by `scripts`.

This can be virtually anything. To access it from within a script event handler, you can do:

```
$extra = $event->getComposer()->getPackage()->getExtra();
```

Optional.

5.3.21 bin

A set of files that should be treated as binaries and symlinked into the `bin-dir` (from config).

See Vendor Bins for more details.

Optional.

Chapter 6

Repositories

This chapter will explain the concept of packages and repositories, what kinds of repositories are available, and how they work.

6.1 Concepts

Before we look at the different types of repositories that exist, we need to understand some of the basic concepts that composer is built on.

6.1.1 Package

Composer is a dependency manager. It installs packages locally. A package is essentially just a directory containing something. In this case it is PHP code, but in theory it could be anything. And it contains a package description which has a name and a version. The name and the version are used to identify the package.

In fact, internally composer sees every version as a separate package. While this distinction does not matter when you are using composer, it's quite important when you want to change it.

In addition to the name and the version, there is useful metadata. The information most relevant for installation is the source definition, which describes where to get the package contents. The package data points to the contents of the package. And there are two options here: dist and source.

Dist: The dist is a packaged version of the package data. Usually a released version, usually a stable release.

Source: The source is used for development. This will usually originate from a source code repository, such as git. You can fetch this when you want to modify the downloaded package.

Packages can supply either of these, or even both. Depending on certain factors, such as user-supplied options and stability of the package, one will be preferred.

6.1.2 Repository

A repository is a package source. It's a list of packages/versions. Composer will look in all your repositories to find the packages your project requires.

By default only the Packagist repository is registered in Composer. You can add more repositories to your project by declaring them in `composer.json`.

Repositories are only available to the root package and the repositories defined in your dependencies will not be loaded. Read the FAQ entry if you want to learn why.

6.2 Types

6.2.1 Composer

The main repository type is the `composer` repository. It uses a single `packages.json` file that contains all of the package metadata.

This is also the repository type that packagist uses. To reference a `composer` repository, just supply the path before the `packages.json` file. In case of packagist, that file is located at `/packages.json`, so the URL of the repository would be `packagist.org`. For `example.org/packages.json` the repository URL would be `example.org`.

packages The only required field is `packages`. The JSON structure is as follows:

```
{
    "packages": {
        "vendor/package-name": {
            "dev-master": { @composer.json },
            "1.0.x-dev": { @composer.json },
            "0.0.1": { @composer.json },
            "1.0.0": { @composer.json }
        }
    }
}
```

The `@composer.json` marker would be the contents of the `composer.json` from that package version including as a minimum:

- name
- version
- dist or source

Here is a minimal package definition:

```
{
    "name": "smarty/smarty",
    "version": "3.1.7",
    "dist": {
        "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
        "type": "zip"
    }
}
```

It may include any of the other fields specified in the schema.

notify The `notify` field allows you to specify an URL template for a URL that will be called every time a user installs a package. The URL can be either an absolute path (that will use the same domain as the repository) or a fully qualified URL.

An example value:

```
{
    "notify": "/downloads/%package%"
}
```

For `example.org/packages.json` containing a `monolog/monolog` package, this would send a `POST` request to `example.org/downloads/monolog/monolog` with following parameters:

- **version:** The version of the package.
- **version_normalized:** The normalized internal representation of the version.

This field is optional.

includes For large repositories it is possible to split the `packages.json` into multiple files. The `includes` field allows you to reference these additional files.

An example:

```
{
    "includes": {
        "packages-2011.json": {
            "sha1": "525a85fb37edd1ad71040d429928c2c0edec9d17"
        },
        "packages-2012-01.json": {
            "sha1": "897cde726f8a3918faf27c803b336da223d400dd"
        },
        "packages-2012-02.json": {
            "sha1": "26f911ad717da26bbcac3f8f435280d13917efa5"
        }
    }
}
```

The SHA-1 sum of the file allows it to be cached and only re-requested if the hash changed.

This field is optional. You probably don't need it for your own custom repository.

6.2.2 VCS

VCS stands for version control system. This includes versioning systems like git, svn or hg. Composer has a repository type for installing packages from these systems.

There are a few use cases for this. The most common one is maintaining your own fork of a third party library. If you are using a certain library for your project and you decide to change something in the library, you will want your project to use the patched version. If the library is on GitHub (this is the case most of the time), you can simply fork it there and push your changes to your fork. After that you update the project's `composer.json`. All you have to do is add your fork as a repository and update the version constraint to point to your custom branch. For version constraint naming conventions see Libraries for more information.

Example assuming you patched monolog to fix a bug in the `bugfix` branch:

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "http://github.com/igorw/monolog"
        }
    ],
    "require": {
        "monolog/monolog": "dev-bugfix"
    }
}
```

When you run `php composer.phar update`, you should get your modified version of `monolog/-monolog` instead of the one from packagist.

Git is not the only version control system supported by the VCS repository. The following are supported:

- **Git:** git-scm.com
- **Subversion:** subversion.apache.org
- **Mercurial:** mercurial.selenic.com

To get packages from these systems you need to have their respective clients installed. That can be inconvenient. And for this reason there is special support for GitHub and BitBucket that use the APIs provided by these sites, to fetch the packages without having to install the version control system. The VCS repository provides `dists` for them that fetch the packages as zips.

- **GitHub:** github.com (Git)
- **BitBucket:** bitbucket.org (Git and Mercurial)

The VCS driver to be used is detected automatically based on the URL. However, should you need to specify one for whatever reason, you can use `git`, `svn` or `hg` as the repository type instead of `vcs`.

6.2.3 PEAR

It is possible to install packages from any PEAR channel by using the `pear` repository. Composer will prefix all package names with `pear-{channelName}/` to avoid conflicts. All packages are also aliased with prefix `pear-{channelAlias}/`

Example using `pear2.php.net`:

```
{
    "repositories": [
        {
            "type": "pear",
            "url": "http://pear2.php.net"
        }
    ],
    "require": {
        "pear-pear2.php.net/PEAR2_Text_Markdown": "*",
        "pear-pear2/PEAR2_HTTP_Request": "*"
    }
}
```

In this case the short name of the channel is `pear2`, so the `PEAR2_HTTP_Request` package name becomes `pear-pear2/PEAR2_HTTP_Request`.

> **Note:** The `pear` repository requires doing quite a few requests per package, so this may considerably slow down the installation process.

Custom channel alias It is possible to alias all pear channel packages with custom name.

Example:

You own private pear repository and going to use composer abilities to bring dependencies from vcs or transit to composer repository scheme. Your repository list of packages:

- BasePackage, requires nothing
- IntermediatePackage, depends on BasePackage
- TopLevelPackage1 and TopLevelPackage2 both dependth on IntermediatePackage.

For composer it looks like:

- "pear-pear.foobar.repo/IntermediatePackage" depends on "pear-pear.foobar.repo/BasePackage",

- "pear-pear.foobar.repo/TopLevelPackage1" depends on "pear-pear.foobar.repo/IntermediatePackage",
- "pear-pear.foobar.repo/TopLevelPackage2" depends on "pear-pear.foobar.repo/IntermediatePackage"

When you update one of your packages to composer naming scheme or made it available through vcs, your older dependencies would not see new version, cause it would be named like "foobar/IntermediatePackage". Specifying 'vendor- alias' for pear repository, you will get all its packages aliased with composer-like names. Following example would take BasePackage, TopLevelPackage1 and TopLevelPackage2 packages from pear repository and IntermediatePackage from github repository:

```
{
    "repositories": [
        {
            "type": "git",
            "url": "https://github.com/foobar/intermediate.git"
        },
        {
            "type": "pear",
            "url": "http://pear.foobar.repo",
            "vendor-alias": "foobar"
        }
    ],
    "require": {
        "foobar/TopLevelPackage1": "*",
        "foobar/TopLevelPackage2": "*"
    }
}
```

6.2.4 Package

If you want to use a project that does not support composer through any of the means above, you still can define the package yourself by using a `package` repository.

Basically, you define the same information that is included in the `composer` repository's `packages.json`, but only for a single package. Again, the minimum required fields are `name`, `version`, and either of `dist` or `source`.

Here is an example for the smarty template engine:

```
{
    "repositories": [
        {
            "type": "package",
            "package": {
                "name": "smarty/smarty",
                "version": "3.1.7",
                "dist": {
                    "url": "http://www.smarty.net/files/Smarty
                       -3.1.7.zip",
                    "type": "zip"
                },
                "source": {
                    "url": "http://smarty-php.googlecode.com/
                       svn/",
                    "type": "svn",
                    "reference": "tags/Smarty_3_1_7/
                       distribution/"
                },
                "autoload": {
                    "classmap": ["libs/"]
                }
            }
        }
    ],
    "require": {
        "smarty/smarty": "3.1.*"
    }
}
```

Typically you would leave the source part off, as you don't really need it.

6.3 Hosting your own

While you will probably want to put your packages on packagist most of the time, there are some use cases for hosting your own repository.

- **Private company packages:** If you are part of a company that uses composer for their packages internally, you might want to keep those packages private.
- **Separate ecosystem:** If you have a project which has its own ecosystem, and the packages aren't really reusable by the greater PHP community, you might want to keep them separate to packagist. An example of this would be wordpress plugins.

When hosting your own package repository it is recommended to use a `composer` one. This is type that is native to composer and yields the best performance.

There are a few tools that can help you create a `composer` repository.

6.3.1 Packagist

The underlying application used by packagist is open source. This means that you can just install your own copy of packagist, re-brand, and use it. It's really quite straight-forward to do. However due to its size and complexity, for most small and medium sized companies willing to track a few packages will be better off using Satis.

Packagist is a Symfony2 application, and it is available on GitHub. It uses composer internally and acts as a proxy between VCS repositories and the composer users. It holds a list of all VCS packages, periodically re-crawls them, and exposes them as a composer repository.

To set your own copy, simply follow the instructions from the packagist github repository.

6.3.2 Satis

Satis is a static `composer` repository generator. It is a bit like an ultra- lightweight, static file-based version of packagist.

You give it a `composer.json` containing repositories, typically VCS and package repository definitions. It will fetch all the packages that are `required` and dump a `packages.json` that is your `composer` repository.

Check the satis GitHub repository and the Satis article for more information.

6.4 Disabling Packagist

You can disable the default Packagist repository by adding this to your `composer.json`:

```
{
    "repositories": [
        {
            "packagist": false
        }
    ]
}
```

Chapter 7

Community

There are a lot of people using composer already, quite a few are also already contributing.

7.1 Contributing

If you would like to contribute to composer, please read the README.

The most important guidelines are described as follows:

> All code contributions - including those of people having commit access - must go through a pull request and approved by a core developer before being merged. This is to ensure proper review of all the code.
>
> Fork the project, create a feature branch, and send us a pull request.
>
> To ensure a consistent code base, you should make sure the code follows the Coding Standards which we borrowed from Symfony.

7.2 IRC / mailing list

The developer mailing list is on google groups IRC channels are available for discussion as well, on irc.freenode.org #composer for users and #composer-dev for development.

Chapter 8

Articles

8.1 Aliases

8.1.1 Why aliases?

When you are using a VCS repository, you will only get comparable versions for branches that look like versions, such as `2.0`. For your `master` branch, you will get a `dev-master` version. For your `bugfix` branch, you will get a `dev-bugfix` version.

If your `master` branch is used to tag releases of the `1.0` development line, i.e. `1.0.1`, `1.0.2`, `1.0.3`, etc., any package depending on it will probably require version `1.0.*`.

If anyone wants to require the latest `dev-master`, they have a problem: Other packages may require `1.0.*`, so requiring that dev version will lead to conflicts, since `dev-master` does not match the `1.0.*` constraint.

Enter aliases.

8.1.2 Branch alias

The `dev-master` branch is one in your main VCS repo. It is rather common that someone will want the latest master dev version. Thus, Composer allows you to alias your `dev-master` branch to a `1.0.x-dev` version. It is done by specifying a `branch-alias` field under `extra` in `composer.json`:

```
{
    "extra": {
        "branch-alias": {
            "dev-master": "1.0.x-dev"
        }
    }
}
```

The branch version must begin with `dev-` (non-comparable version), the alias must be a comparable dev version. The `branch-alias` must be present on the branch that it references. For `dev-master`, you need to commit it on the `master` branch.

As a result, you can now require `1.0.*` and it will happily install `dev-master` for you.

8.1.3 Require inline alias

Branch aliases are great for aliasing main development lines. But in order to use them you need to have control over the source repository, and you need to commit changes to version control.

This is not really fun when you just want to try a bugfix of some library that is a dependency of your local project.

For this reason, you can alias packages in your `require` and `require-dev` fields. Let's say you found a bug in the `monolog/monolog` package. You cloned Monolog on GitHub and fixed the issue in a branch named `bugfix`. Now you want to install that version of monolog in your local project.

You are using `symfony/monolog-bundle` which requires `monolog/monolog` version `1.*`. So you need your `dev-bugfix` to match that constraint.

Just add this to your project's root `composer.json`:

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "https://github.com/you/monolog"
        }
    ],
    "require": {
        "symfony/monolog-bundle": "2.0",
        "monolog/monolog": "dev-bugfix as 1.0.x-dev"
    }
}
```

That will fetch the `dev-bugfix` version of `monolog/monolog` from your GitHub and alias it to `1.0.x-dev`.

> **Note:** If a package with inline aliases is required, the alias (right of the `as`) is used as the version constraint. The part left of the `as` is discarded. As a consequence, if A requires B and B requires `monolog/monolog` version `dev-bugfix as 1.0.x-dev`, installing A will make B require `1.0.x-dev`, which may exist as a branch alias or an actual `1.0` branch. If it does not, it must be re-inline-aliased in A's `composer.json`.

> **Note:** Inline aliasing should be avoided, especially for published packages. If you found a bug, try and get your fix merged upstream. This helps to avoid issues for users of your package.

8.2 Setting up and using custom installers

8.2.1 Synopsis

At times it may be necessary for a package to require additional actions during installation, such as installing packages outside of the default `vendor` library.

In these cases you could consider creating a Custom Installer to handle your specific logic.

8.2.2 Calling a Custom Installer

Suppose that your project already has a Custom Installer for specific modules then invoking that installer is a matter of defining the correct type in your package file.

> *See the next chapter for an instruction how to create Custom Installers.*

Every Custom Installer defines which type string it will recognize. Once recognized it will completely override the default installer and only apply its own logic.

An example use-case would be:

> phpDocumentor features Templates that need to be installed outside of the default /vendor folder structure. As such they have chosen to adopt the `phpdocumentor-template` type and create a Custom Installer to send these templates to the correct folder.

An example composer.json of such a template package would be:

```
{
    "name": "phpdocumentor/template-responsive",
    "type": "phpdocumentor-template",
    "require": {
        "phpdocumentor/template-installer": "*"
    }
}
```

> **IMPORTANT**: to make sure that the template installer is present at the time the template package is installed, template packages should require the installer package.

8.2.3 Creating an Installer

A Custom Installer is defined as a class that implements the `Composer\Installer\InstallerInterface` and is contained in a Composer package that has the type `composer-installer`.

A basic Installer would thus compose of two files:

1. the package file: composer.json
2. The Installer class, i.e.: `Composer\Installer\MyInstaller.php`

> **NOTE**: *The namespace does not need to be `Composer\Installer`, it must only implement the right interface.*

composer.json

The package file is the same as any other package file but with the following requirements:

1. the type attribute must be `composer-installer`.
2. the extra attribute must contain an element `class` defining the class name of the installer (including namespace). If a package contains multiple installers this can be array of class names.

Example:

```
{
    "name": "phpdocumentor/template-installer",
    "type": "composer-installer",
    "license": "MIT",
    "autoload": {
        "psr-0": {"phpDocumentor\\Composer": "src/"}
    },
    "extra": {
        "class": "phpDocumentor\\Composer\\TemplateInstaller"
    }
}
```

The Custom Installer class

The class that executes the custom installation should implement the `Composer\Installer\InstallerInterface` (or extend another installer that implements that interface).

The class may be placed in any location and have any name, as long as it is autoloadable and matches the `extra.class` element in the package definition. It will also define the type string as it will be recognized by packages that will use this installer in the `supports()` method.

> **NOTE**: *choose your type name carefully, it is recommended to follow the format: `vendor-type`*. For example: `phpdocumentor-template`.

The InstallerInterface class defines the following methods (please see the source for the exact signature):

- **supports()**, here you test whether the passed type matches the name that you declared for this installer (see the example).
- **isInstalled()**, determines whether a supported package is installed or not.
- **install()**, here you can determine the actions that need to be executed upon installation.
- **update()**, here you define the behavior that is required when Composer is invoked with the update argument.
- **uninstall()**, here you can determine the actions that need to be executed when the package needs to be removed.
- **getInstallPath()**, this method should return the location where the package is to be installed, *relative from the location of composer.json.*

Example:

```
namespace phpDocumentor\Composer;

use Composer\Package\PackageInterface;
use Composer\Installer\LibraryInstaller;

class TemplateInstaller extends LibraryInstaller
{
    /**
     * {@inheritDoc}
     */
    public function getInstallPath(PackageInterface $package)
    {
        $prefix = substr($package->getPrettyName(), 0, 23);
        if ('phpdocumentor/template-' !== $prefix) {
            throw new \InvalidArgumentException(
                'Unable to install template, phpdocumentor
                   templates '
                .'should always start their package name with '
                .'"phpdocumentor/template-"'
            );
        }

        return 'data/templates/'.substr($package->getPrettyName
            (), 23);
    }

    /**
     * {@inheritDoc}
     */
    public function supports($packageType)
    {
        return 'phpdocumentor-template' === $packageType;
    }
}
```

The example demonstrates that it is quite simple to extend the `Composer\Installer\LibraryInstaller` class to strip a prefix (`phpdocumentor/template-`) and use the remaining part to assemble a completely different installation path.

> *Instead of being installed in `/vendor` any package installed using this Installer will be put in the `/data/templates/<stripped name>` folder.*

8.3 Handling private packages with Satis

Satis can be used to host the metadata of your company's private packages, or your own. It basically acts as a micro-packagist. You can get it from GitHub or install via CLI: `composer.phar create-project composer/satis`.

8.3.1 Setup

For example let's assume you have a few packages you want to reuse across your company but don't really want to open-source. You would first define a Satis configuration file, which is basically a stripped-down version of a `composer.json` file. It contains a few repositories, and then you use the require key to say which packages it should dump in the static repository it creates, or use require-all to select all of them.

Here is an example configuration, you see that it holds a few VCS repositories, but those could be any types of repositories. Then it uses `"require-all": true` which selects all versions of all packages in the repositories you defined.

```
{
    "name": "My Repository",
    "homepage": "http://packages.example.org",
    "repositories": [
        { "type": "vcs", "url": "http://github.com/mycompany/
           privaterepo" },
        { "type": "vcs", "url": "http://svn.example.org/private
           /repo" },
        { "type": "vcs", "url": "http://github.com/mycompany/
           privaterepo2" }
    ],
    "require-all": true
}
```

If you want to cherry pick which packages you want, you can list all the packages you want to have in your satis repository inside the classic composer `require` key, using a `"*"` constraint to make sure all versions are selected, or another constraint if you want really specific versions.

```
{
    "repositories": [
        { "type": "vcs", "url": "http://github.com/mycompany/
            privaterepo" },
        { "type": "vcs", "url": "http://svn.example.org/private
            /repo" },
        { "type": "vcs", "url": "http://github.com/mycompany/
            privaterepo2" }
    ],
    "require": {
        "company/package": "*",
        "company/package2": "*",
        "company/package3": "2.0.0"
    }
}
```

Once you did this, you just run `php bin/satis build <configuration file> <build dir>`. For example `php bin/satis build config.json web/` would read the `config.json` file and build a static repository inside the `web/` directory.

When you ironed out that process, what you would typically do is run this command as a cron job on a server. It would then update all your package info much like Packagist does.

Note that if your private packages are hosted on GitHub, your server should have an ssh key that gives it access to those packages, and then you should add the `--no-interaction` (or `-n`) flag to the command to make sure it falls back to ssh key authentication instead of prompting for a password. This is also a good trick for continuous integration servers.

Set up a virtual-host that points to that `web/` directory, let's say it is `packages.example.org`.

8.3.2 Usage

In your projects all you need to add now is your own composer repository using the `packages.example.org` as URL, then you can require your private packages and everything should work smoothly. You don't need to copy all your repositories in every project anymore. Only that one unique repository that will update itself.

```
{
    "repositories": [ { "type": "composer", "url": "http://
       packages.example.org/" } ],
    "require": {
        "company/package": "1.2.0",
        "company/package2": "1.5.2",
        "company/package3": "dev-master"
    }
}
```

8.4 Scripts

8.4.1 What is a script?

A script is a callback (defined as a static method) that will be called when the event it listens on is triggered.

Scripts are only executed on the root package, not on the dependencies that are installed.

8.4.2 Event types

- **pre-install-cmd**: occurs before the install command is executed.
- **post-install-cmd**: occurs after the install command is executed.
- **pre-update-cmd**: occurs before the update command is executed.
- **post-update-cmd**: occurs after the update command is executed.
- **pre-package-install**: occurs before a package is installed.
- **post-package-install**: occurs after a package is installed.
- **pre-package-update**: occurs before a package is updated.
- **post-package-update**: occurs after a package is updated.
- **pre-package-uninstall**: occurs before a package has been uninstalled.
- **post-package-uninstall**: occurs after a package has been uninstalled.

8.4.3 Defining scripts

Scripts are defined by adding the `scripts` key to a project's `composer.json`.

They are specified as an array of classes and static method names.

The classes used as scripts must be autoloadable via Composer's autoload functionality.

Script definition example:

```
{
    "scripts": {
        "post-update-cmd": "MyVendor\\MyClass::postUpdate",
        "post-package-install": [
            "MyVendor\\MyClass::postPackageInstall"
        ]
    }
}
```

The event handler receives a `Composer\Script\Event` object as an argument, which gives you access to the `Composer\Composer` instance through the `getComposer` method.

Using the previous example, here's an event listener example :

```
<?php

namespace MyVendor;

use Composer\Script\Event;

class MyClass
{
    public static function postUpdate(Event $event)
    {
        $composer = $event->getComposer();
        // do stuff
    }

    public static function postPackageInstall(Event $event)
    {
        $installedPackage = $event->getOperation()->getPackage
            ();
        // do stuff
    }
}
```

8.5 bin and vendor/bin

8.5.1 What is a bin?

Any command line script that a Composer package would like to pass along to a user who installs the package should be listed as a bin.

If a package contains other scripts that are not needed by the package users (like build or compile scripts) that code should not be listed as a bin.

8.5.2 How is it defined?

It is defined by adding the `bin` key to a project's `composer.json`. It is specified as an array of files so multiple bins can be added for any given project.

```
{
    "bin": ["bin/my-script", "bin/my-other-script"]
}
```

8.5.3 What does defining a bin in composer.json do?

It instructs Composer to install the package's bins to `vendor/bin` for any project that **depends** on that project.

This is a convenient way to expose useful scripts that would otherwise be hidden deep in the `vendor/` directory.

8.5.4 What happens when Composer is run on a composer.json that defines bins?

For the bins that a package defines directly, nothing happens.

8.5.5 What happens when Composer is run on a composer.json that has dependencies with bins listed?

Composer looks for the bins defined in all of the dependencies. A symlink is created from each dependency's bins to `vendor/bin`.

Say package `my-vendor/project-a` has bins setup like this:

```
{
    "name": "my-vendor/project-a",
    "bin": ["bin/project-a-bin"]
}
```

Running `composer install` for this `composer.json` will not do anything with `bin/project-a-bin`.

Say project `my-vendor/project-b` has requirements setup like this:

```
{
    "name": "my-vendor/project-b",
    "requires": {
        "my-vendor/project-a": "*"
    }
}
```

Running `composer install` for this `composer.json` will look at all of project-b's dependencies and install them to `vendor/bin`.

In this case, Composer will make `vendor/my-vendor/project-a/bin/project-a-bin` available as `vendor/bin/project-a-bin`. On a Unix-like platform this is accomplished by creating a symlink.

8.5.6 What about Windows and .bat files?

Packages managed entirely by Composer do not *need* to contain any `.bat` files for Windows compatibility. Composer handles installation of bins in a special way when run in a Windows environment:

- A `.bat` files is generated automatically to reference the bin
- A Unix-style proxy file with the same name as the bin is generated automatically (useful for Cygwin or Git Bash)

Packages that need to support workflows that may not include Composer are welcome to maintain custom `.bat` files. In this case, the package should **not** list the `.bat` file as a bin as it is not needed.

8.5.7 Can vendor bins be installed somewhere other than vendor/bin?

Yes, there are two ways that an alternate vendor bin location can be specified.

- Setting the `bin-dir` configuration setting in `composer.json`
- Setting the environment variable `COMPOSER_BIN_DIR`

An example of the former looks like this:

```
{
    "config": {
        "bin-dir": "scripts"
    }
}
```

Running `composer install` for this `composer.json` will result in all of the vendor bins being installed in `scripts/` instead of `vendor/bin/`.

Chapter 9

FAQs

9.1 How do I install a package to a custom path for my framework?

Each framework may have one or many different required package installation paths. Composer can be configured to install packages to a folder other than the default `vendor` folder by using composer/installers.

If you are a **package author** and want your package installed to a custom directory, simply require `composer/installers` and set the appropriate `type`. This is common if your package is intended for a specific framework such as CakePHP, Drupal or WordPress. Here is an example composer.json file for a WordPress theme:

```
{
    "name": "you/themename",
    "type": "wordpress-theme",
    "require": {
        "composer/installers": "*"
    }
}
```

Now when your theme is installed with Composer it will be placed into `wp-content/themes/themename/` folder. Check the current supported types for your package.

As a **package consumer** you can set or override the install path for a package that requires composer/installers by configuring the `installer-paths` extra. A useful example would be for a Drupal multisite setup where the package should be installed into your sites subdirectory. Here we are overriding the install path for a module that uses composer/installers:

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

Now the package would be installed to your folder location, rather than the default composer/installers determined location.

> **Note:** You cannot use this to change the path of any package. This is only applicable to packages that require `composer/installers` and use a custom type that it handles.

9.2 Should I commit the dependencies in my vendor directory?

The general recommendation is **no**. The vendor directory (or wherever your dependencies are installed) should be added to `.gitignore`/`svn:ignore`/etc.

The best practice is to then have all the developers use Composer to install the dependencies. Similarly, the build server, CI, deployment tools etc should be adapted to run Composer as part of their project bootstrapping.

While it can be tempting to commit it in some environment, it leads to a few problems:

- Large VCS repository size and diffs when you update code.
- Duplication of the history of all your dependencies in your own VCS.
- Adding dependencies installed via git to a git repo will show them as submodules. This is problematic because they are not real submodules, and you will run into issues.

If you really feel like you must do this, you have two options:

- Limit yourself to installing tagged releases (no dev versions), so that you only get zipped installs, and avoid problems with the git "submodules".
- Remove the .git directory of every dependency after the installation, then you can add them to your git repo. You can do that with `rm -rf vendor/**/.git` but this means you will have to delete those dependencies from disk before running composer update.

9.3 Why are version constraints combining comparisons and wildcards a bad idea?

This is a fairly common mistake people make, defining version constraints in their package requires like `>=2.*` or `>=1.1.*`.

If you think about it and what it really means though, you will quickly realize that it does not make much sense. If we decompose `>=2.*`, you have two parts:

- `>=2` which says the package should be in version 2.0.0 or above.
- `2.*` which says the package should be between version 2.0.0 (inclusive) and 3.0.0 (exclusive).

As you see, both rules agree on the fact that the package must be >=2.0.0, but it is not possible to determine if when you wrote that you were thinking of a package in version 3.0.0 or not. Should it match because you asked for `>=2` or should it not match because you asked for a `2.*`?

For this reason, Composer just throws an error and says that this is invalid. The easy way to fix it is to think about what you really mean, and use only one of those rules.

9.4 Why can't Composer load repositories recursively?

You may run into problems when using custom repositories because Composer does not load the repositories of your requirements, so you have to redefine those repositories in all your `composer.json` files.

Before going into details as to why this is like that, you have to understand that the main use of custom VCS & package repositories is to temporarily try some things, or use a fork of a project until your pull request is merged, etc. You should not use them to keep track of private packages. For that you should look into setting up Satis for your company or even for yourself.

There are three ways the dependency solver could work with custom repositories:

- Fetch the repositories of root package, get all the packages from the defined repositories, resolve requirements. This is the current state and it works well except for the limitation of not loading repositories recursively.
- Fetch the repositories of root package, while initializing packages from the defined repos, initialize recursively all repos found in those packages, and their package's packages, etc, then resolve requirements. It could work, but it slows down the initialization

a lot since VCS repos can each take a few seconds, and it could end up in a completely broken state since many versions of a package could define the same packages inside a package repository, but with different dist/source. There are many many ways this could go wrong.

- Fetch the repositories of root package, then fetch the repositories of the first level dependencies, then fetch the repositories of their dependencies, etc, then resolve requirements. This sounds more efficient, but it suffers from the same problems than the second solution, because loading the repositories of the dependencies is not as easy as it sounds. You need to load all the repos of all the potential matches for a requirement, which again might have conflicting package definitions.