Composer

The Composer Community

May 19, 2014

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": "1.2.*"
    }
}
```

We are simply stating that our project requires some `monolog/monolog` package, any version beginning with `1.2`.

1.3 System Requirements

Composer requires PHP 5.3.2+ to run. A few sensitive php settings and compile flags are also required, but the installer will warn you about any incompatibilities.

To install packages from sources instead of simple zip archives, you will need git, svn or hg depending on how the package is version-controlled.

Composer is multi-platform and we strive to make it run equally well on Windows, Linux and OSX.

1.4 Installation - *nix

1.4.1 Downloading the Composer Executable

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

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

> **Note:** If the above fails for some reason, you can download the installer with `php` instead:

```
$ php -r "readfile('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 -sS 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 -sS https://getcomposer.org/installer | php
$ mv composer.phar /usr/local/bin/composer
```

> **Note:** If the above fails due to permissions, run the `mv` line again with sudo.

Then, just run `composer` in order to run Composer instead of `php composer.phar`.

Globally (on OSX via homebrew) Composer is part of the homebrew-php project.

```
brew update
brew tap josegonzalez/homebrew-php
brew tap homebrew/versions
brew install php55-intl
brew install josegonzalez/php/composer
```

1.5 Installation - Windows

1.5.1 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 just call `composer` from any directory in your command line.

1.5.2 Manual Installation

Change to a directory on your `PATH` and run the install snippet to download composer.phar:

```
C:\Users\username>cd C:\bin
C:\bin>php -r "readfile('https://getcomposer.org/installer');"
    | php
```

> **Note:** If the above fails due to readfile, use the `http` url or enable php_openssl.dll in php.ini

Create a new `composer.bat` file alongside `composer.phar`:

```
C:\bin>echo @php "%~dp0composer.phar" %*>composer.bat
```

Close your current terminal. Test usage with a new terminal:

```
C:\Users\username>composer -V
Composer version 27d8904

C:\Users\username>
```

1.6 Using Composer

We will now use Composer to install the dependencies of the project. If you don't have a `composer.json` file in the current directory please skip to the Basic Usage chapter.

To resolve and download dependencies, run the `install` command:

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

If you did a global install and do not have the phar in that directory run this instead:

```
$ composer install
```

Following the example above, this will download monolog into the `vendor/monolog/-monolog` directory.

1.7 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';
```

Woah! 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 -sS https://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 -sS https://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

In the previous example we were 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.

Name | Example | Description ——— | ——— | ——— Exact version | `1.0.2` | You can specify the exact version of a package. Range | `>=1.0` `>=1.0,<2.0` `>=1.0,<1.1` | `>=1.2` | By using comparison operators you can specify ranges of valid versions. Valid operators are `>`, `>=`, `<`, `<=`, `!=`. You can define multiple ranges, separated by a comma, which will be treated as a **logical AND**. A pipe symbol `|` will be treated as a **logical OR**. AND has higher precedence than OR. Wildcard | `1.0.*` | You can specify a pattern with a `*` wildcard. `1.0.*` is the equivalent of `>=1.0,<1.1`. Tilde Operator | $\sim$ | Very useful for projects that follow semantic versioning. $\sim$ is equivalent to `>=1.2,<2.0`. For more details, read the next section below.

2.2.4 Next Significant Release (Tilde Operator)

The ~ operator is best explained by example: ~ is equivalent to `>=1.2,<2.0`, while ~ is equivalent to `>=1.2.3,<1.3`. 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 ~ (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.

> **Note:** Though `2.0-beta.1` is strictly before `2.0`, a version constraint like ~ would not install it. As said above ~ only means the `.2` can change but the `1.` part is fixed.

2.2.5 Stability

By default only stable releases are taken into consideration. If you would like to also get RC, beta, alpha or dev versions of your dependencies you can do so using stability flags. To change that for all packages instead of doing per dependency you can also use the minimum-stability setting.

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. Your CI server, production machines, other developers in your team, everything and everyone runs on the same dependencies, which mitigates the potential for bugs affecting only some parts of the deployments. Even if you develop alone, in six months when reinstalling the project you can feel confident the dependencies installed are still working even if your dependencies released many new versions since then.

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
```

If you only want to install or update one dependency, you can whitelist them:

```
$ php composer.phar update monolog/monolog [...]
```

> **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-4": {"Acme\\": "src/"}
    }
}
```

Composer will register a PSR-4 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/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-4 autoloading, classmap is also supported. This allows classes to be

autoloaded even if they do not conform to PSR-4. 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_*.php` files, which return associative arrays allowing you to configure your own autoloader.

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 Platform packages

Composer has platform packages, which are virtual packages for things that are installed on the system but are not actually installable by Composer. This includes PHP itself, PHP extensions and some system libraries.

- `php` represents the PHP version of the user, allowing you to apply constraints, e.g. `>=5.4.0`. To require a 64bit version of php, you can require the `php-64bit` package.
- `hhvm` represents the version of the HHVM runtime (aka HipHop Virtual Machine) and allows you to apply a constraint, e.g., '>=2.3.3'.
- `ext-<name>` allows you to require PHP extensions (includes core extensions). Versioning can be quite inconsistent here, so it's often a good idea to just set the constraint to `*`. An example of an extension package name is `ext-gd`.
- `lib-<name>` allows constraints to be made on versions of libraries used by PHP. The following are available: `curl`, `iconv`, `icu`, `libxml`, `openssl`, `pcre`, `uuid`, `xsl`.

You can use `composer show --platform` to get a list of your locally available platform packages.

3.3 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"
}
```

> **Note:** You should avoid specifying the version field explicitly, because for tags the value must match the tag name.

3.3.1 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 of `-patch`, `-alpha`, `-beta` or `-RC`.

The suffixes can also be followed by a number.

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:** Even if your tag is prefixed with `v`, a version constraint in a `require` statement has to be specified without prefix (e.g. tag `v1.0.0` will result in version `1.0.0`).

3.3.2 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 development version, it will be automatically pulled from its `source`. See the `install` command for more details.

3.3.3 Aliases

It is possible to 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.

3.4 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`.

3.5 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/username/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/username/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.

To get help from the command-line, simply call `composer` or `composer list` to see the complete list of commands, then `--help` combined with any of those can give you more information.

4.1 Global Options

The following options are available with every command:

- **–verbose (-v):** Increase verbosity of messages.
- **–help (-h):** Display help information.
- **–quiet (-q):** Do not output any message.
- **–no-interaction (-n):** Do not ask any interactive question.
- **–working-dir (-d):** If specified, use the given directory as working directory.
- **–profile:** Display timing and memory usage information
- **–ansi:** Force ANSI output.
- **–no-ansi:** Disable ANSI output.
- **–version (-V):** Display this application version.

4.2 Process Exit Codes

- **0:** OK
- **1:** Generic/unknown error code
- **2:** Dependency solving error code

4.3 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
```

4.3.1 Options

- **–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**.
- **–stability (-s):** Value for the `minimum-stability` field.

4.4 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.

4.4.1 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.
- **–prefer-dist:** Reverse of `--prefer-source`, composer will install from `dist` if possible. This can speed up installs substantially on build servers and other use cases where you typically do not run updates of the vendors. It is also a way to circumvent problems with git if you do not have a proper setup.
- **–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:** Install packages listed in `require-dev` (this is the default behavior).
- **–no-dev:** Skip installing packages listed in `require-dev`.
- **–no-scripts:** Skips execution of scripts defined in `composer.json`.
- **–no-plugins:** Disables plugins.
- **–no-progress:** Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
- **–optimize-autoloader (-o):** Convert PSR-0/4 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.

4.5 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
```

You can also use wildcards to update a bunch of packages at once:

```
$ php composer.phar update vendor/*
```

4.5.1 Options

- **–prefer-source:** Install packages from `source` when available.
- **–prefer-dist:** Install packages from `dist` when available.
- **–dry-run:** Simulate the command without actually doing anything.
- **–dev:** Install packages listed in `require-dev` (this is the default behavior).
- **–no-dev:** Skip installing packages listed in `require-dev`.
- **–no-scripts:** Skips execution of scripts defined in `composer.json`.
- **–no-plugins:** Disables plugins.
- **–no-progress:** Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
- **–optimize-autoloader (-o):** Convert PSR-0/4 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.
- **–lock:** Only updates the lock file hash to suppress warning about the lock file being out of date.
- **–with-dependencies** Add also all dependencies of whitelisted packages to the whitelist. So all packages with their dependencies are updated recursively.

4.6 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
```

4.6.1 Options

- **–prefer-source:** Install packages from `source` when available.
- **–prefer-dist:** Install packages from `dist` when available.
- **–dev:** Add packages to `require-dev`.
- **–no-update:** Disables the automatic update of the dependencies.
- **–no-progress:** Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
- **–update-with-dependencies** Also update dependencies of the newly required packages.

4.7 global

The global command allows you to run other commands like `install`, `require` or `update` as if you were running them from the COMPOSER_HOME directory.

This can be used to install CLI utilities globally and if you add `$COMPOSER_HOME/vendor/bin` to your `$PATH` environment variable. Here is an example:

```
$ php composer.phar global require fabpot/php-cs-fixer:dev-
    master
```

Now the `php-cs-fixer` binary is available globally (assuming you adjusted your PATH). If you wish to update the binary later on you can just run a global update:

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

4.8 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.

4.8.1 Options

- **–only-name (-N):** Search only in name.

4.9 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
```

4.9.1 Options

- **–installed (-i):** List the packages that are installed.
- **–platform (-p):** List only platform packages (php & extensions).
- **–self (-s):** List the root package info.

4.10 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
```

4.10.1 Options

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

4.11 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
```

4.11.1 Options

- **–no-check-all:** Wether or not composer do a complete validation.

4.12 status

If you often need to modify the code of your dependencies and they are installed from source, the `status` command allows you to check if you have local changes in any of them.

```
$ php composer.phar status
```

With the `--verbose` option you get some more information about what was changed:

```
$ php composer.phar status -v
You have changes in the following dependencies:
vendor/seld/jsonlint:
    M README.mdown
```

4.13 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 would like to instead update to a specific release simply specify it:

```
$ composer self-update 1.0.0-alpha7
```

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

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

4.13.1 Options

- **–rollback (-r):** Rollback to the last version you had installed.
- **–clean-backups:** Delete old backups during an update. This makes the current version of composer the only backup available after the update.

4.14 config

The `config` command allows you to edit some basic composer settings in either the local composer.json file or the global config.json file.

```
$ php composer.phar config --list
```

4.14.1 Usage

```
config [options] [setting-key] [setting-value1] ...  [setting-valueN]
```

`setting-key` is a configuration option name and `setting-value1` is a configuration value. For settings that can take an array of values (like `github-protocols`), more than one setting-value arguments are allowed.

See the config schema section for valid configuration options.

4.14.2 Options

- **–global (-g):** Operate on the global config file located at `$COMPOSER_HOME/config.json` by default. Without this option, this command affects the local composer.json file or a file specified by `--file`.
- **–editor (-e):** Open the local composer.json file using in a text editor as defined by the `EDITOR` env variable. With the `--global` option, this opens the global config file.
- **–unset:** Remove the configuration element named by `setting-key`.
- **–list (-l):** Show the list of current config variables. With the `--global` option this lists the global configuration only.
- **–file="..." (-f):** Operate on a specific file instead of composer.json. Note that this cannot be used in conjunction with the `--global` option.

4.14.3 Modifying Repositories

In addition to modifying the config section, the `config` command also supports making changes to the repositories section by using it the following way:

```
$ php composer.phar config repositories.foo vcs http://github.
    com/foo/bar
```

4.15 create-project

You can use Composer to create new projects from an existing package. This is the equivalent of doing a git clone/svn checkout followed by a composer install of the vendors.

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.

If the directory does not currently exist, it will be created during installation.

```
php composer.phar create-project doctrine/orm path 2.2.*
```

It is also possible to run the command without params in a directory with an existing `composer.json` file to bootstrap a project.

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

4.15.1 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.
- **–stability (-s):** Minimum stability of package. Defaults to `stable`.
- **–prefer-source:** Install packages from `source` when available.
- **–prefer-dist:** Install packages from `dist` when available.
- **–dev:** Install packages listed in `require-dev`.
- **–no-install:** Disables installation of the vendors.
- **–no-plugins:** Disables plugins.
- **–no-scripts:** Disables the execution of the scripts defined in the root package.
- **–no-progress:** Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
- **–keep-vcs:** Skip the deletion of the VCS metadata for the created project. This is mostly useful if you run the command in non-interactive mode.

4.16 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-0/4 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-0/4 for convenience and classmaps for performance.

4.16.1 Options

- **–optimize (-o):** Convert PSR-0/4 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.
- **–no-dev:** Disables autoload-dev rules.

4.17 licenses

Lists the name, version and license of every package installed. Use `--format=json` to get machine readable output.

4.18 run-script

To run scripts manually you can use this command, just give it the script name and optionally –no-dev to disable the dev mode.

4.19 diagnose

If you think you found a bug, or something is behaving strangely, you might want to run the `diagnose` command to perform automated checks for many common problems.

```
$ php composer.phar diagnose
```

4.20 archive

This command is used to generate a zip/tar archive for a given package in a given version. It can also be used to archive your entire project without excluded/ignored files.

```
$ php composer.phar archive vendor/package 2.0.21 --format=zip
```

4.20.1 Options

- **–format (-f):** Format of the resulting archive: tar or zip (default: “tar”)
- **–dir:** Write the archive to this directory (default: “.”)

4.21 help

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

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

4.22 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 the env vars will always take precedence over the values specified in `composer.json`.

4.22.1 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
```

4.22.2 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`.

4.22.3 COMPOSER_VENDOR_DIR

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

4.22.4 COMPOSER_BIN_DIR

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

4.22.5 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>`.

4.22.6 no_proxy

If you are behind a proxy and would like to disable it for certain domains, you can use the `no_proxy` env var. Simply set it to a comma separated list of domains the proxy should *not* be used for.

The env var accepts domains, IP addresses, and IP address blocks in CIDR notation. You can restrict the filter to a particular port (e.g. `:80`). You can also set it to `*` to ignore the proxy for all HTTP requests.

4.22.7 HTTP_PROXY_REQUEST_FULLURI

If you use a proxy but it does not support the request_fulluri flag, then you should set this env var to `false` or `0` to prevent composer from setting the request_fulluri option.

4.22.8 HTTPS_PROXY_REQUEST_FULLURI

If you use a proxy but it does not support the request_fulluri flag for HTTPS requests, then you should set this env var to `false` or `0` to prevent composer from setting the request_fulluri option.

4.22.9 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.

4.22.10 COMPOSER_CACHE_DIR

The `COMPOSER_CACHE_DIR` var allows you to change the composer cache directory, which is also configurable via the `cache-dir` option.

By default it points to $COMPOSER_HOME/cache on *nix and OSX, and `C:\Users\<user>\AppData\Local\Compose`
(or `%LOCALAPPDATA%/Composer`) on Windows.

4.22.11 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).

4.22.12 COMPOSER_DISCARD_CHANGES

This env var controls the discard-changes config option.

4.22.13 COMPOSER_NO_INTERACTION

If set to 1, this env var will make composer behave as if you passed the `--no-interaction` flag to every command. This can be set on build boxes/CI.

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. In most cases this is not required and should be omitted (see below).

This must follow the format of `X.Y.Z` or `vX.Y.Z` with an optional suffix of `-dev`, `-patch`, `-alpha`, `-beta` or `-RC`. The patch, alpha, beta and RC suffixes can also be followed by a number.

Examples:

```
1.0.0
1.0.2
1.1.0
0.2.5
1.0.0-dev
1.0.0-alpha3
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 four types:

- **library:** This is the default. It will simply copy the files to `vendor`.
- **project:** This denotes a project rather than a library. For example application shells like the Symfony standard edition, CMSs like the SilverStripe installer or full fledged applications distributed as packages. This can for example be used by IDEs to provide listings of projects to initialize when creating a new workspace.
- **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-plugin:** A package of type `composer-plugin` may provide 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.

For closed-source software, you may use `"proprietary"` as the license identifier.

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 for example.

Example:

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

If one of your dependencies has a dependency on an unstable package you need to explicitly require it as well, along with its sufficient stability flag.

Example:

```
{
    "require": {
        "doctrine/doctrine-fixtures-bundle": "dev-master",
        "doctrine/data-fixtures": "@dev"
    }
}
```

`require` and `require-dev` additionally support explicit references (i.e. commit) for dev versions to make sure they are locked 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>`.

Example:

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

> **Note:** While this is convenient at times, it should not be how you use packages in the long term because it comes with a technical limitation. The composer.json metadata will still be read from the branch name you specify before the hash. Because of that in some cases it will not be a practical workaround, and you should always try to switch to tagged releases as soon as you can.

It is also possible to inline-alias a package constraint so that it matches a constraint that it otherwise would not. For more information see the aliases article.

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 are installed by default. Both `install` or `update` support the `--no-dev` option that prevents dev dependencies from being installed.

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

Note that when specifying ranges like `<1.0, >= 1.1` in a `conflict` link, this will state a conflict with all versions that are less than 1.0 *and* equal or newer than 1.1 at the same time, which is probably not what you want. You probably want to go for `<1.0 | >= 1.1` in this case.

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, `PSR-4` autoloading, `classmap` generation and `files` includes are supported. PSR-4 is the recommended way though since it offers greater ease of use (no need to regenerate the autoloader when you add classes).

PSR-4 Under the `psr-4` key you define a mapping from namespaces to paths, relative to the package root. When autoloading a class like `Foo\\Bar\\Baz` a namespace prefix `Foo\\` pointing to a directory `src/` means that the autoloader will look for a file named `src/Bar/Baz.php` and include it if present. Note that as opposed to the older PSR-0 style, the prefix (`Foo\\`) is **not** present in the file path.

Namespace prefixes must end in \\ to avoid conflicts between similar prefixes. For example `Foo` would match classes in the `FooBar` namespace so the trailing backslashes solve the problem: `Foo\\` and `FooBar\\` are distinct.

The PSR-4 references are all combined, during install/update, into a single key => value array which may be found in the generated file `vendor/composer/autoload_psr4.php`.

Example:

```
{
    "autoload": {
        "psr-4": {
            "Monolog\\": "src/",
            "Vendor\\Namespace\\": ""
        }
    }
}
```

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

```
{
    "autoload": {
        "psr-4": { "Monolog\\": ["src/", "lib/"] }
    }
}
```

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

```
{
    "autoload": {
        "psr-4": { "": "src/" }
    }
}
```

PSR-0 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.

Please note namespace declarations should end in \\ to make sure the autoloader responds exactly. For example `Foo` would match in `FooBar` so the trailing backslashes solve the problem: `Foo\\` and `FooBar\\` are distinct.

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/",
            "Vendor_Namespace_": "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/" }
    }
}
```

Classmap 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`. This map is built by scanning for classes in all `.php` and `.inc` files in the given directories/files.

You can use the classmap generation support to define autoloading for all libraries that do not follow PSR-0/4. To configure this you specify all directories or files to search for classes.

Example:

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

Files 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 autoload-dev (root-only)

This section allows to define autoload rules for development purposes.

Classes needed to run the test suite should not be included in the main autoload rules to avoid polluting the autoloader in production and when other people use your package as a dependency.

Therefore, it is a good idea to rely on a dedicated path for your unit tests and to add it within the autoload-dev section.

Example:

```
{
    "autoload": {
        "psr-4": { "MyLibrary\\": "src/" }
    },
    "autoload-dev": {
        "psr-4": { "MyLibrary\\Tests\\": "tests/" }
    }
}
```

5.3.15 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.

5.3.16 target-dir

> **DEPRECATED**: This is only present to support legacy PSR-0 style autoloading, and all new code should preferably use PSR-4 without target-dir and projects using PSR-0 with PHP namespaces are encouraged to migrate to PSR-4 instead.

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.

5.3.17 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 (in order of stability) are `dev`, `alpha`, `beta`, `RC`, and `stable`.

5.3.18 prefer-stable (root-only)

When this is enabled, Composer will prefer more stable packages over unstable ones when finding compatible stable packages is possible. If you require a dev version or only alphas are available for a package, those will still be selected granted that the minimum-stability allows for it.

Use `"prefer-stable": true` to enable.

5.3.19 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 the net-

work (HTTP, FTP, SSH), that contains a list of `composer.json` objects with additional `dist` and/or `source` information. The `packages.json` file is loaded using a PHP stream. You can set extra options on that stream using the `options` parameter.

- **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": "composer",
            "url": "https://packages.example.com",
            "options": {
                "ssl": {
                    "verify_peer": "true"
                }
            }
        },
        {
            "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.

5.3.20 config (root-only)

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

The following options are supported:

- **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.
- **use-include-path:** Defaults to `false`. If true, the Composer autoloader will also look for classes in the PHP include path.
- **preferred-install:** Defaults to `auto` and can be any of `source`, `dist` or `auto`. This option allows you to set the install method Composer will prefer to use.
- **github-protocols:** Defaults to `["git", "https", "ssh"]`. A list of protocols to use when cloning from github.com, in priority order. You can reconfigure it to for example prioritize the https protocol if you are behind a proxy or have somehow bad performances with the git protocol.
- **github-oauth:** A list of domain names and oauth keys. For example using `{"github.com": "oauthtoken"}` as the value of this option will use `oauthtoken` to access private repositories on github and to circumvent the low IP-based rate limiting of their API. Read more on how to get an OAuth token for GitHub.
- **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.
- **cache-dir:** Defaults to `$home/cache` on unix systems and `C:\Users\<user>\AppData\Local\Composer` on Windows. Stores all the caches used by composer. See also COMPOSER_HOME.
- **cache-files-dir:** Defaults to `$cache-dir/files`. Stores the zip archives of packages.
- **cache-repo-dir:** Defaults to `$cache-dir/repo`. Stores repository metadata for the `composer` type and the VCS repos of type `svn`, `github` and `bitbucket`.
- **cache-vcs-dir:** Defaults to `$cache-dir/vcs`. Stores VCS clones for loading VCS repository metadata for the `git/hg` types and to speed up installs.
- **cache-files-ttl:** Defaults to `15552000` (6 months). Composer caches all dist (zip, tar, ..) packages that it downloads. Those are purged after six months of being unused by default. This option allows you to tweak this duration (in seconds) or disable it completely by setting it to 0.

- **cache-files-maxsize:** Defaults to `300MiB`. Composer caches all dist (zip, tar, ..) packages that it downloads. When the garbage collection is periodically ran, this is the maximum size the cache will be able to use. Older (less used) files will be removed first until the cache fits.
- **prepend-autoloader:** Defaults to `true`. If false, the composer autoloader will not be prepended to existing autoloaders. This is sometimes required to fix interoperability issues with other autoloaders.
- **autoloader-suffix:** Defaults to `null`. String to be used as a suffix for the generated Composer autoloader. When null a random one will be generated.
- **optimize-autoloader** Defaults to `false`. Always optimize when dumping the autoloader.
- **github-domains:** Defaults to `["github.com"]`. A list of domains to use in github mode. This is used for GitHub Enterprise setups.
- **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.
- **discard-changes:** Defaults to `false` and can be any of `true`, `false` or `"stash"`. This option allows you to set the default style of handling dirty updates when in non-interactive mode. `true` will always discard changes in vendors, while `"stash"` will try to stash and reapply. Use this for CI servers or deploy scripts if you tend to have modified vendors.

Example:

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

5.3.21 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.

5.3.22 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.

5.3.23 bin

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

See Vendor Binaries for more details.

Optional.

5.3.24 archive

A set of options for creating package archives.

The following options are supported:

- **exclude:** Allows configuring a list of patterns for excluded paths. The pattern syntax matches .gitignore files. A leading exclamation mark (!) will result in any matching files to be included even if a previous pattern excluded them. A leading slash will only match at the beginning of the project relative path. An asterisk will not expand to a directory separator.

Example:

```
{
    "archive": {
        "exclude": ["/foo/bar", "baz", "/*.test", "!/foo/bar/
           baz"]
    }
}
```

The example will include `/dir/foo/bar/file`, `/foo/bar/baz`, `/file.php`, `/foo/my.test` but it will exclude `/foo/bar/any`, `/foo/baz`, and `/my.test`.

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-batch The `notify-batch` field allows you to specify an 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-batch": "/downloads/"
}
```

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

```
{
    "downloads": [
        {"name": "monolog/monolog", "version": "1.2.1.0"},
    ]
}
```

The version field will contain the normalized representation of the version number.

This field is optional.

includes For larger 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.

provider-includes and providers-url For very large repositories like packagist.org using the so-called provider files is the preferred method. The `provider-includes` field allows you to list a set of files that list package names provided by this repository. The hash should be a sha256 of the files in this case.

The `providers-url` describes how provider files are found on the server. It is an absolute path from the repository root.

An example:

```
{
    "provider-includes": {
        "providers-a.json": {
            "sha256": "
                f5b4bc0b354108ef08614e569c1ed01a2782e67641744864a74e788982886f4c
                "
        },
        "providers-b.json": {
            "sha256": "
                b38372163fac0573053536f5b8ef11b86f804ea8b016d239e706191203f6efac
                "
        }
    },
    "providers-url": "/p/%package%$%hash%.json"
}
```

Those files contain lists of package names and hashes to verify the file integrity, for example:

```
{
    "providers": {
        "acme/foo": {
            "sha256": "38968
                de1305c2e17f4de33aea164515bc787c42c7e2d6e25948539a14268bb82
                "
        },
        "acme/bar": {
            "sha256": "4
                dd24c930bd6e1103251306d6336ac813b563a220d9ca14f4743c032fb047233
                "
        }
    }
}
```

The file above declares that acme/foo and acme/bar can be found in this repository, by loading the file referenced by `providers-url`, replacing `%package%` by the package name and `%hash%` by the sha256 field. Those files themselves just contain package definitions as described above.

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

stream options The `packages.json` file is loaded using a PHP stream. You can set extra options on that stream using the `options` parameter. You can set any valid PHP stream context option. See Context options and parameters for more information.

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.

Loading a package from a VCS repository 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": "https://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.

Note that you should not rename the package unless you really intend to fork it in the long term, and completely move away from the original package. Composer will correctly pick your package over the original one since the custom repository has priority over packagist. If you want to rename the package, you should do so in the default (often master) branch and not in a feature branch, since the package name is taken from the default branch.

If other dependencies rely on the package you forked, it is possible to inline-alias it so that it matches a constraint that it otherwise would not. For more information see the aliases article.

Using private repositories Exactly the same solution allows you to work with your private repositories at GitHub and BitBucket:

```
{
    "require": {
        "vendor/my-private-repo": "dev-master"
    },
    "repositories": [
        {
            "type": "vcs",
            "url":  "git@bitbucket.org:vendor/my-private-repo.
                git"
        }
    ]
}
```

The only requirement is the installation of SSH keys for a git client.

Git alternatives 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`.

If you set the `no-api` key to `true` on a github repository it will clone the repository as it would with any other git repository instead of using the GitHub API. But unlike using the `git` driver directly, composer will still attempt to use github's zip files.

Subversion Options Since Subversion has no native concept of branches and tags, Composer assumes by default that code is located in `$url/trunk`, `$url/branches` and `$url/tags`.

If your repository has a different layout you can change those values. For example if you used capitalized names you could configure the repository like this:

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "http://svn.example.org/projectA/",
            "trunk-path": "Trunk",
            "branches-path": "Branches",
            "tags-path": "Tags"
        }
    ]
}
```

If you have no branches or tags directory you can disable them entirely by setting the `branches-path` or `tags-path` to `false`.

If the package is in a sub-directory, e.g. `/trunk/foo/bar/composer.json` and `/tags/1.0/foo/bar/composer.json`, then you can make composer access it by setting the `"package-path"` option to the sub-directory, in this example it would be `"package-path": "foo/bar/"`.

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 vendor alias It is possible to alias PEAR channel packages with a custom vendor name.

Example:

Suppose you have a private PEAR repository and wish to use Composer to incorporate dependencies from a VCS. Your PEAR repository contains the following packages:

- `BasePackage`
- `IntermediatePackage`, which depends on `BasePackage`
- `TopLevelPackage1` and `TopLevelPackage2` which both depend on `IntermediatePackage`

Without a vendor alias, Composer will use the PEAR channel name as the vendor portion of the package name:

- `pear-pear.foobar.repo/BasePackage`
- `pear-pear.foobar.repo/IntermediatePackage`
- `pear-pear.foobar.repo/TopLevelPackage1`
- `pear-pear.foobar.repo/TopLevelPackage2`

Suppose at a later time you wish to migrate your PEAR packages to a Composer repository and naming scheme, and adopt the vendor name of `foobar`. Projects using your PEAR packages would not see the updated packages, since they have a different vendor name (`foobar/IntermediatePackage` vs `pear-pear.foobar.repo/IntermediatePackage`).

By specifying `vendor-alias` for the PEAR repository from the start, you can avoid this scenario and future-proof your package names.

To illustrate, the following example would get the `BasePackage`, `TopLevelPackage1`, and `TopLevelPackage2` packages from your PEAR repository and `IntermediatePackage` from a 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.

> **Note**: This repository type has a few limitations and should be avoided whenever possible:
>
> - Composer will not update the package unless you change the `version` field.
> - Composer will not update the commit references, so if you use `master` as reference you will have to delete the package to force an update, and will have to deal with an unstable lock file.

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.

For hosting your own packages, a native `composer` type of repository is recommended, which provides 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.

6.3.3 Artifact

There are some cases, when there is no ability to have one of the previously mentioned repository types online, even the VCS one. Typical example could be cross-organisation library exchange through built artifacts. Of course, most of the times they are private. To simplify maintenance, one can simply use a repository of type `artifact` with a folder containing ZIP archives of those private packages:

```
{
    "repositories": [
        {
            "type": "artifact",
            "url": "path/to/directory/with/zips/"
        }
    ],
    "require": {
        "private-vendor-one/core": "15.6.2",
        "private-vendor-two/connectivity": "*",
        "acme-corp/parser": "10.3.5"
    }
}
```

Each zip artifact is just a ZIP archive with `composer.json` in root folder:

```
$ unzip -l acme-corp-parser-10.3.5.zip
composer.json
...
```

If there are two archives with different versions of a package, they are both imported. When an archive with a newer version is added in the artifact folder and you run `update`, that version will be imported as well and Composer will update to the latest version.

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 many people using composer already, and quite a few of them are 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

Mailing lists for user support and development.

IRC channels are on irc.freenode.org: #composer for users and #composer-dev for development.

Stack Overflow has a growing collection of Composer related questions.

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` or `2.0.x`. 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 (i.e. start with numbers, and end with `.x-dev`). 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, anyone can now require `1.0.*` and it will happily install `dev-master`.

In order to use branch aliasing, you must own the repository of the package being aliased. If you want to alias a third party package without maintaining a fork of it, use inline aliases as described below.

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 plugin providing the 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-plugin": "*"
    }
}
```

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

8.2.3 Creating an Installer

A Custom Installer is defined as a class that implements the `Composer\Installer\InstallerInterface` and is usually distributed in a Composer Plugin.

A basic Installer Plugin would thus compose of three files:

1. the package file: composer.json
2. The Plugin class, e.g.: `My\Project\Composer\Plugin.php`, containing a class that implements `Composer\Plugin\PluginInterface`.
3. The Installer class, e.g.: `My\Project\Composer\Installer.php`, containing a class that implements `Composer\Installer\InstallerInterface`.

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

Example:

```
{
    "name": "phpdocumentor/template-installer-plugin",
    "type": "composer-plugin",
    "license": "MIT",
    "autoload": {
        "psr-0": {"phpDocumentor\\Composer": "src/"}
    },
    "extra": {
        "class": "phpDocumentor\\Composer\\
           TemplateInstallerPlugin"
    },
    "require": {
        "composer-plugin-api": "1.0.0"
    }
}
```

The Plugin class

The class defining the Composer plugin must implement the `Composer\Plugin\PluginInterface`. It can then register the Custom Installer in its `activate()` method.

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.

Example:

```
namespace phpDocumentor\Composer;

use Composer\Composer;
use Composer\IO\IOInterface;
use Composer\Plugin\PluginInterface;

class TemplateInstallerPlugin implements PluginInterface
{
    public function activate(Composer $composer, IOInterface
        $io)
    {
        $installer = new TemplateInstaller($io, $composer);
        $composer->getInstallationManager()->addInstaller(
            $installer);
    }
}
```

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). It defines 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 getPackageBasePath(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 is a static `composer` repository generator. It is a bit like an ultra- lightweight, static file-based version of packagist and 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 --stability=dev`.

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: a json file with an arbitrary name that lists your curated repositories.

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.

The default file Satis looks for is `satis.json` in the root of the repository.

```
{
    "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`. Alternatively, with PHP >= 5.4.0, you can use the built-in CLI server `php -S localhost:port -t satis-output-dir/` for a temporary solution.

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"
    }
}
```

Security

To secure your private repository you can host it over SSH or SSL using a client certificate. In your project you can use the `options` parameter to specify the connection options for the server.

Example using a custom repository using SSH (requires the SSH2 PECL extension):

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "ssh2.sftp://example.org",
            "options": {
                "ssh2": {
                    "username": "composer",
                    "pubkey_file": "/home/composer/.ssh/id_rsa.
                       pub",
                    "privkey_file": "/home/composer/.ssh/id_rsa
                       "
                }
            }
        }
    ]
}
```

Tip: See ssh2 context options for more information.

Example using HTTP over SSL using a client certificate:

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "https://example.org",
            "options": {
                "ssl": {
                    "local_cert": "/home/composer/.ssl/composer
                        .pem"
                }
            }
        }
    ]
}
```

Tip: See ssl context options for more information.

Downloads

When GitHub or BitBucket repositories are mirrored on your local satis, the build process will include the location of the downloads these platforms make available. This means that the repository and your setup depend on the availability of these services.

At the same time, this implies that all code which is hosted somewhere else (on another service or for example in Subversion) will not have downloads available and thus installations usually take a lot longer.

To enable your satis installation to create downloads for all (Git, Mercurial and Subversion) your packages, add the following to your `satis.json`:

```
{
    "archive": {
        "directory": "dist",
        "format": "tar",
        "prefix-url": "https://amazing.cdn.example.org",
        "skip-dev": true
    }
}
```

Options explained

- `directory`: the location of the dist files (inside the `output-dir`)

- `format`: optional, `zip` (default) or `tar`
- `prefix-url`: optional, location of the downloads, homepage (from `satis.json`) followed by `directory` by default
- `skip-dev`: optional, `false` by default, when enabled (`true`) satis will not create downloads for branches

Once enabled, all downloads (include those from GitHub and BitBucket) will be replaced with a *local* version.

prefix-url Prefixing the URL with another host is especially helpful if the downloads end up in a private Amazon S3 bucket or on a CDN host. A CDN would drastically improve download times and therefore package installation.

Example: A `prefix-url` of `http://my-bucket.s3.amazonaws.com` (and `directory` set to `dist`) creates download URLs which look like the following: `http://my-bucket.s3.amazonaws.com/dist/vendor-package-version-ref.zip`.

Resolving dependencies

It is possible to make satis automatically resolve and add all dependencies for your projects. This can be used with the Downloads functionality to have a complete local mirror of packages. Just add the following to your `satis.json`:

```
{
    "require-dependencies": true
}
```

When searching for packages, satis will attempt to resolve all the required packages from the listed repositories. Therefore, if you are requiring a package from Packagist, you will need to define it in your `satis.json`.

8.4 Setting up and using plugins

8.4.1 Synopsis

You may wish to alter or expand Composer's functionality with your own. For example if your environment poses special requirements on the behaviour of Composer which do not apply to the majority of its users or if you wish to accomplish something with composer in a way that is not desired by most users.

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

8.4.2 Creating a Plugin

A plugin is a regular composer package which ships its code as part of the package and may also depend on further packages.

Plugin Package

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

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

Additionally you must require the special package called `composer-plugin-api` to define which composer API versions your plugin is compatible with. The current composer plugin API version is 1.0.0.

For example

```
{
    "name": "my/plugin-package",
    "type": "composer-plugin",
    "require": {
        "composer-plugin-api": "1.0.0"
    }
}
```

Plugin Class

Every plugin has to supply a class which implements the `Composer\Plugin\PluginInterface`. The `activate()` method of the plugin is called after the plugin is loaded and receives an instance of `Composer\Composer` as well as an instance of `Composer\IO\IOInterface`. Using these two objects all configuration can be read and all internal objects and state can be manipulated as desired.

Example:

```
namespace phpDocumentor\Composer;

use Composer\Composer;
use Composer\IO\IOInterface;
use Composer\Plugin\PluginInterface;

class TemplateInstallerPlugin implements PluginInterface
{
    public function activate(Composer $composer, IOInterface
       $io)
    {
        $installer = new TemplateInstaller($io, $composer);
        $composer->getInstallationManager()->addInstaller(
           $installer);
    }
}
```

8.4.3 Event Handler

Furthermore plugins may implement the `Composer\EventDispatcher\EventSubscriberInterface` in order to have its event handlers automatically registered with the `EventDispatcher` when the plugin is loaded.

The events available for plugins are:

- **COMMAND**, is called at the beginning of all commands that load plugins. It provides you with access to the input and output objects of the program.
- **PRE_FILE_DOWNLOAD**, is triggered before files are downloaded and allows you to manipulate the `RemoteFilesystem` object prior to downloading files based on the URL to be downloaded.

  A plugin can also subscribe to script events.

Example:

```
namespace Naderman\Composer\AWS;

use Composer\Composer;
use Composer\EventDispatcher\EventSubscriberInterface;
use Composer\IO\IOInterface;
use Composer\Plugin\PluginInterface;
use Composer\Plugin\PluginEvents;
use Composer\Plugin\PreFileDownloadEvent;

class AwsPlugin implements PluginInterface,
    EventSubscriberInterface
{
    protected $composer;
    protected $io;

    public function activate(Composer $composer, IOInterface
        $io)
    {
        $this->composer = $composer;
        $this->io = $io;
    }

    public static function getSubscribedEvents()
    {
        return array(
            PluginEvents::PRE_FILE_DOWNLOAD => array(
                array('onPreFileDownload', 0)
            ),
        );
    }

    public function onPreFileDownload(PreFileDownloadEvent
        $event)
    {
        $protocol = parse_url($event->getProcessedUrl(),
            PHP_URL_SCHEME);

        if ($protocol === 's3') {
            $awsClient = new AwsClient($this->io, $this->
                composer->getConfig());
            $s3RemoteFilesystem = new S3RemoteFilesystem($this
                ->io, $event->getRemoteFilesystem()->getOptions
                (), $awsClient);
            $event->setRemoteFilesystem($s3RemoteFilesystem);
        }
    }
}
```

8.4.4 Using Plugins

Plugin packages are automatically loaded as soon as they are installed and will be loaded when composer starts up if they are found in the current project's list of installed packages. Additionally all plugin packages installed in the `COMPOSER_HOME` directory using the composer global command are loaded before local project plugins are loaded.

> You may pass the `--no-plugins` option to composer commands to disable all installed commands. This may be particularly helpful if any of the plugins causes errors and you wish to update or uninstall it.

8.5 Scripts

8.5.1 What is a script?

A script, in Composer's terms, can either be a PHP callback (defined as a static method) or any command-line executable command. Scripts are useful for executing a package's custom code or package-specific commands during the Composer execution process.

NOTE: Only scripts defined in the root package's `composer.json` are executed. If a dependency of the root package specifies its own scripts, Composer does not execute those additional scripts.

8.5.2 Event names

Composer fires the following named events during its execution process:

- **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-status-cmd**: occurs before the `status` command is executed.
- **post-status-cmd**: occurs after the `status` 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.
- **pre-autoload-dump**: occurs before the autoloader is dumped, either during `install/update`, or via the `dump-autoload` command.
- **post-autoload-dump**: occurs after the autoloader is dumped, either during `install/update`, or via the `dump-autoload` command.
- **post-root-package-install**: occurs after the root package has been installed, during the `create-project` command.
- **post-create-project-cmd**: occurs after the `create-project` command is executed.
- **pre-archive-cmd**: occurs before the `archive` command is executed.
- **post-archive-cmd**: occurs after the `archive` command is executed.

NOTE: Composer makes no assumptions about the state of your dependencies prior to `install` or `update`. Therefore, you should not specify scripts that require Composer-managed dependencies in the `pre-update-cmd` or `pre-install-cmd` event hooks. If you need to execute scripts prior to `install` or `update` please make sure they are self-contained within your root package.

8.5.3 Defining scripts

The root JSON object in `composer.json` should have a property called `"scripts"`, which contains pairs of named events and each event's corresponding scripts. An event's scripts can be defined as either as a string (only for a single script) or an array (for single or multiple scripts.)

For any given event:

- Scripts execute in the order defined when their corresponding event is fired.
- An array of scripts wired to a single event can contain both PHP callbacks and command-line executables commands.
- PHP classes containing defined callbacks must be autoloadable via Composer's autoload functionality.

Script definition example:

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

Using the previous definition example, here's the class `MyVendor\MyClass` that might be used to execute the PHP callbacks:

```
<?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
    }

    public static function warmCache(Event $event)
    {
        // make cache toasty
    }
}
```

When an event is fired, Composer's internal event handler receives a `Composer\Script\Event` object, which is passed as the first argument to your PHP callback. This `Event` object has getters for other contextual objects:

- `getComposer()`: returns the current instance of `Composer\Composer`

- `getName()`: returns the name of the event being fired as a string
- `getIO()`: returns the current input/output stream which implements `Composer\IO\IOInterface` for writing to the console

8.5.4 Running scripts manually

If you would like to run the scripts for an event manually, the syntax is:

```
$ composer run-script [--dev] [--no-dev] script
```

For example `composer run-script post-install-cmd` will run any **post-install-cmd** scripts that have been defined. # Troubleshooting

This is a list of common pitfalls on using Composer, and how to avoid them.

8.5.5 General

1. Before asking anyone, run `composer diagnose` to check for common problems. If it all checks out, proceed to the next steps.
2. When facing any kind of problems using Composer, be sure to **work with the latest version**. See self-update for details.
3. Make sure you have no problems with your setup by running the installer's checks via `curl -sS https://getcomposer.org/installer | php -- --check`.
4. Ensure you're **installing vendors straight from your** `composer.json` via `rm -rf vendor && composer update -v` when troubleshooting, excluding any possible interferences with existing vendor installations or `composer.lock` entries.

8.5.6 Package not found

1. Double-check you **don't have typos** in your `composer.json` or repository branches and tag names.
2. Be sure to **set the right minimum-stability**. To get started or be sure this is no issue, set `minimum-stability` to "dev".
3. Packages **not coming from Packagist** should always be **defined in the root package** (the package depending on all vendors).
4. Use the **same vendor and package name** throughout all branches and tags of your repository, especially when maintaining a third party fork and using `replace`.

8.5.7 Package not found on travis-ci.org

1. Check the "Package not found" item above.
2. If the package tested is a dependency of one of its dependencies (cyclic dependency), the problem might be that composer is not able to detect the version of the package properly. If it is a git clone it is generally alright and Composer will detect the version of the current branch, but travis does shallow clones so that process can fail when testing pull requests and feature branches in general. The best solution is to define the version you are on via an environment variable called COMPOSER_ROOT_VERSION. You set it to `dev-master` for example to define the root package's version as `dev-master`. Use: `before_script: COMPOSER_ROOT_VERSION=dev-master composer install` to export the variable for the call to composer.

8.5.8 Need to override a package version

Let say your project depends on package A which in turn depends on a specific version of package B (say 0.1) and you need a different version of that package - version 0.11.

You can fix this by aliasing version 0.11 to 0.1:

composer.json:

```
{
    "require": {
        "A": "0.2",
        "B": "0.11 as 0.1"
    }
}
```

See aliases for more information.

8.5.9 Memory limit errors

If composer shows memory errors on some commands:

```
PHP Fatal error:  Allowed memory size of XXXXXX bytes exhausted
    <...>
```

The PHP `memory_limit` should be increased.

> **Note:** Composer internally increases the `memory_limit` to `512M`. If you have memory issues when using composer, please consider creating an issue ticket so we can look into it.

To get the current `memory_limit` value, run:

```
php -r "echo ini_get('memory_limit').PHP_EOL;"
```

Try increasing the limit in your `php.ini` file (ex. `/etc/php5/cli/php.ini` for Debian-like systems):

```
; Use -1 for unlimited or define an explicit value like 512M
memory_limit = -1
```

Or, you can increase the limit with a command-line argument:

```
php -d memory_limit=-1 composer.phar <...>
```

8.5.10 "The system cannot find the path specified" (Windows)

1. Open regedit.
2. Search for an `AutoRun` key inside `HKEY_LOCAL_MACHINE\Software\Microsoft\Command Processor` or `HKEY_CURRENT_USER\Software\Microsoft\Command Processor`.
3. Check if it contains any path to non-existent file, if it's the case, just remove them.

8.5.11 API rate limit and OAuth tokens

Because of GitHub's rate limits on their API it can happen that Composer prompts for authentication asking your username and password so it can go ahead with its work.

If you would prefer not to provide your GitHub credentials to Composer you can manually create a token using the following procedure:

1. Create an OAuth token on GitHub. Read more on this.
2. Add it to the configuration running `composer config -g github-oauth.github.com <oauthtoken>`

Now Composer should install/update without asking for authentication.

8.5.12 proc_open(): fork failed errors

If composer shows proc_open() fork failed on some commands:

```
PHP Fatal error: Uncaught exception 'ErrorException' with
   message 'proc_open(): fork failed - Cannot allocate memory'
    in phar
```

This could be happening because the VPS runs out of memory and has no Swap space enabled.

```
[root@my_tiny_vps htdocs]# free -m
total used free shared buffers cached
Mem: 2048 357 1690 0 0 237
-/+ buffers/cache: 119 1928
Swap: 0 0 0
```

To enable the swap you can use for example:

```
/bin/dd if=/dev/zero of=/var/swap.1 bs=1M count=1024
/sbin/mkswap /var/swap.1
/sbin/swapon /var/swap.1
```

8.6 Vendor binaries and the `vendor/bin` directory

8.6.1 What is a vendor binary?

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 vendor binary.

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 vendor binary.

8.6.2 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 binaries can be added for any given project.

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

8.6.3 What does defining a vendor binary in composer.json do?

It instructs Composer to install the package's binaries 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.

8.6.4 What happens when Composer is run on a composer.json that defines vendor binaries?

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

8.6.5 What happens when Composer is run on a composer.json that has dependencies with vendor binaries listed?

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

Say package `my-vendor/project-a` has binaries 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",
    "require": {
        "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.

8.6.6 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 binaries in a special way when run in a Windows environment:

- A `.bat` file is generated automatically to reference the binary
- A Unix-style proxy file with the same name as the binary 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 binary as it is not needed.

8.6.7 Can vendor binaries be installed somewhere other than vendor/bin?

Yes, there are two ways an alternate vendor binary location can be specified:

1. Setting the `bin-dir` configuration setting in `composer.json`
2. 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 binaries 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": "~1.0"
    }
}
```

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 a few options:

1. Limit yourself to installing tagged releases (no dev versions), so that you only get zipped installs, and avoid problems with the git "submodules".
2. Use –prefer-dist or set `preferred-install` to `dist` in your config.
3. 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.

4. Add a .gitignore rule (`vendor/.git`) to ignore all the vendor `.git` folders. This approach does not require that you delete dependencies from disk prior to running a composer update.

9.3 Why are unbound version constraints a bad idea?

A version constraint without an upper bound such as `*`, `>=3.4` or `dev-master` will allow updates to any future version of the dependency. This includes major versions breaking backward compatibility.

Once a release of your package is tagged, you cannot tweak its dependencies anymore in case a dependency breaks BC - you have to do a new release but the previous one stays broken.

The only good alternative is to define an upper bound on your constraints, which you can increase in a new release after testing that your package is compatible with the new major version of your dependency.

For example instead of using `>=3.4` you should use ∼ which allows all versions up to `3.999` but does not include `4.0` and above. The ∼ operator works very well with libraries follow semantic versioning.

Note: As a package maintainer, you can make the life of your users easier by providing an alias version for your development branch to allow it to match bound constraints.

9.4 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.

9.5 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.