composer update

`composer update` will update your depencencies as they are specified in `composer.json`

For example, if you require this package as a dependency:

```
"mockery/mockery": "0.9.*",
```

and you have actually installed the `0.9.1` version of the package, running `composer update` will cause an upgrade of this package (for example to `0.9.2`, if it's already been released)

in detail `composer update` will:

- Read `composer.json`
- Remove installed packages that are no more required in `composer.json`
- Check the availability of the latest versions of your required packages
- Install the latest versions of your packages
- Update `composer.lock` to store the installed packages version

composer install

`composer install` will not update anything; it will just install all the dependencies as specified in the `composer.lock` file

In detail:

- Check if `composer.lock` file exists (if not, run `composer-update` and create it)
- Read `composer.lock` file
- Install the packages specified in the `composer.lock` file

When to install and when to update

- `composer update` is mostly used in the 'development phase', to upgrade our project packages according to what we have specified in the `composer.json` file,
- `composer install` is primarily used in the 'deploying phase' to install our application on a production server or on a testing environment, using the same dependencies stored in the composer.lock file created by composer update.

"This is the right way to use **composer**. ... You should never run **composer update** in production. If however you deploy a new **composer**.lock with new dependencies and/or versions (after having run **composer update** in dev) and then run **composer install** **composer** will **update** and **install** new your new dependencies." Mar 7, 2013

Unless you've been living under a rock, you know about composer[1] and packagist[2] for managing dependencies in PHP. A few days ago, an issue[3] was closed and merged into master which changes the default behaviour of `composer update` to be functionally equivellent to `composer update --require-dev`. This confused a few folks[4], and here's why:

You should only ever run `composer update` to get the newest versions of your dependencies, not to install them.

What's not massively clear (or at least wasn't early on) in the composer documentation[5] is the difference between `composer install` and `composer update` and the relevancy of `composer.lock`. This is exacerbated by composer displaying a warning when running `composer install` with a lockfile present and changes in `composer.json`:

```
$ composer install
Loading composer repositories with package information
Installing dependencies from lock file
Warning: The lock file is not up to date with the latest changes in composer.jso
n. You may be getting outdated dependencies. Run update to update them.
Nothing to install or update
Generating autoload files
```

Not very clear.

Here's a fairly standard composer work-flow:

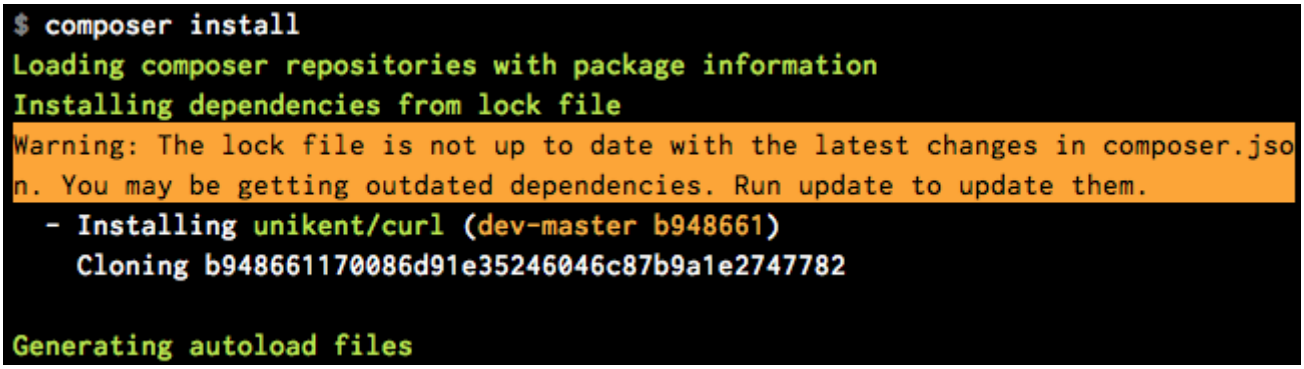
1. Add `composer.json` with some dependencies
2. Run `composer install`
3. Add some more dependencies
4. Run `composer update` as you've updated your dependencies

This is the *right* way to use composer. If you are using composer to deploy your dependencies into a production environment (which many people are), based on this work-flow you may incorrectly assume that you deploy your updated `composer.json` to production and run `composer update`again. This is the *wrong* way to use composer.

What's really happening when you run `composer update` is that it's fetching the newest version of your dependencies as specified by `composer.json`.

If you've been testing your code with monolog 1.2, and monolog 1.3 gets released, unless you're very explicit in your `composer.json` composer will fetch monolog 1.3. Now imagine that a backward incompatible change or bug is introduced with monolog 1.3. Suddenly your dependencies have broken your production environment. Not good.

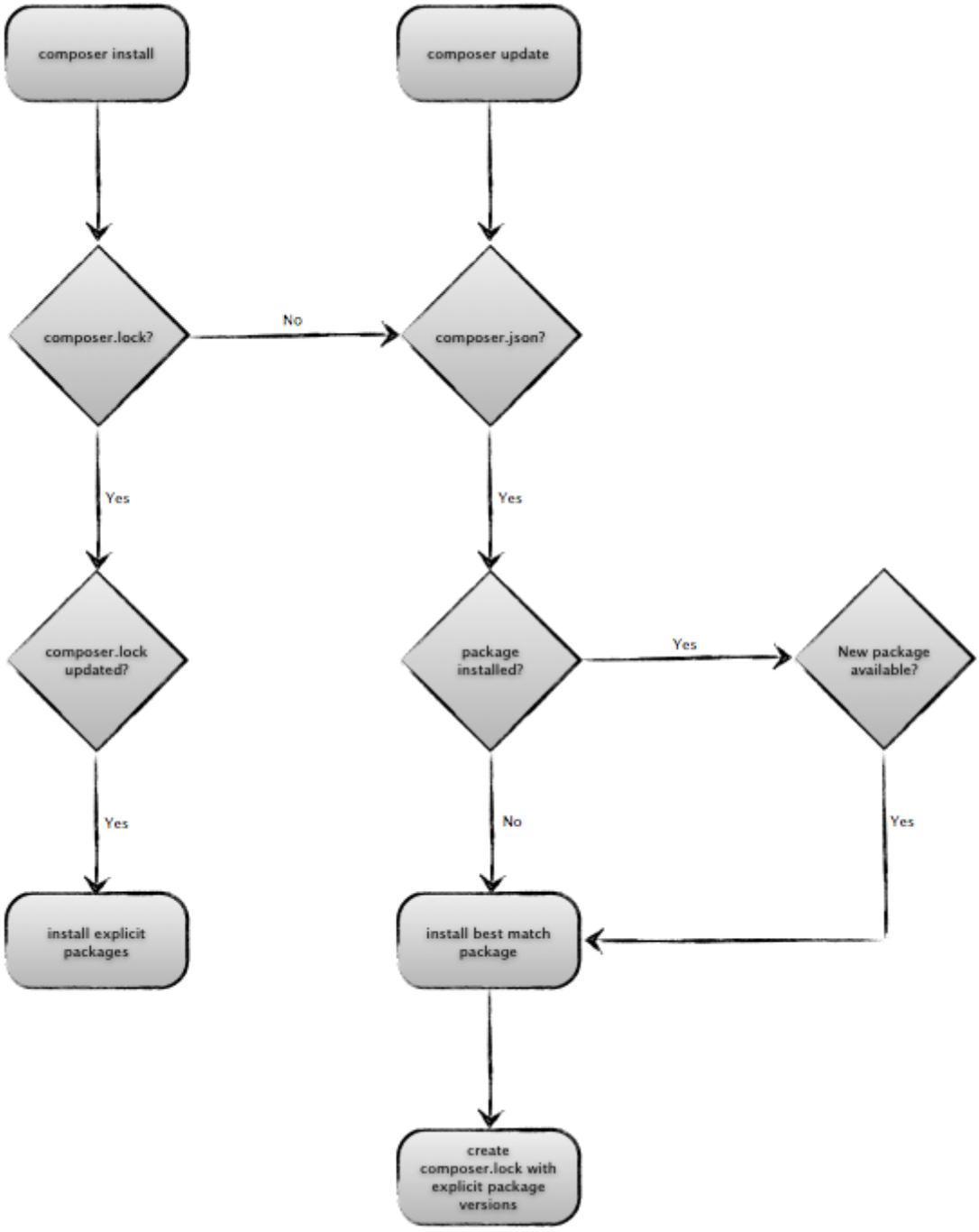
What you really need to do is deploy your updated `composer.lock`, and then re-run `composer install`. You should never run `composer update` in production. If however you deploy a new `composer.lock` with new dependencies and/or versions (after having run `composer update` in dev) and *then* run `composer install` composer will update and install new your new dependencies.



Whenever composer generates a new `composer.lock` it *locks* you to a specific set of dependencies and the latest versions of those dependencies it can resolve.

This means if your `composer.json` specifies `monolog/monolog: 1.*`, and it installs monolog 1.2, monolog 1.2 will be included in your lockfile. From then on when you run `composer install` you will only ever get monolog 1.2, even after monolog 1.3 has been released.

Here's the basic workflow:



Not too complicated.

Now we can come a full circle back to the issue that prompted this post. As we never run `composer update` in production, it follows that whenever we run it we will be in our dev environment, and the automatic inclusion of the `--require-dev` flag on `composer update` now makes sense.

If you're still not happy, you can ignore all of this and add the `--no-dev` flag to reverse the behaviour.