Managing dependencies is more than running “composer update”

Nils Adermann
@naderman
Private Packagist
https://packagist.com

What are Dependencies?

- Services
    - APIs
    - Client-side Integrations (OAuth / External JS / Analytics / …)
- Software
    - Libraries
    - Programs / Tools
- External Assets

What is Dependency Management?

- Assembly
- Dependency Change Management
- Risk Analysis & Reduction

May happen at build time or at runtime

Dependency Assembly

- Installation of Libraries, Tools, etc.
  - composer install
  - apt-get install foo
  - Application of Configuration Management (Puppet, Chef, Ansible, Salt, …)

- Configuration for Connections to Services, external APIs
  - Authentication
  - Glue Code
- Connection to Services (usually at Runtime)

Dependency Assembly

Past:

- Step-by-Step installation instructions
- Readmes, Delete and reinstall individual packages

Today:

- Description of a system state (e.g. composer.json, top.sls)
- Tools to move the system into the state (e.g. composer, salt)

Dependency Change Management

- Dependency Change
  - Adding, Removing, Updating, Replacing of Libraries
  - Replacing APIs
  - composer update
- Dependency Change Management
  - Balance Risks, Consequences, Cost & Advantages
  - Architecture Decisions which enable “Change”
    - Example: Abstraction to replace concrete service

Risk Analysis: Availability

Affects Assembly

Examples:

- Open Source Library deleted
- Payment Service unavailable
- EU VATId Service out of order
- Jenkins not accessible

Risk Reduction: Availability

- Software is available when you have a copy
    - composer cache
    - Forks
    - Private Packagist or Satis
- Services are available depending on external factors
    - Can the service be called asynchronously?
        - e.g. run VATId check after payment
        - e.g. Private Packagist inits package in worker, no GitHub access in controller
    - Are errors clearly presented to users?
        - e.g. low timeouts, error messages when external Service X not available

Risk Analysis: Compatibility

Affects Change Management

Examples:

- BC Break in Library Update
- API Semantics change:
  - Payment API no longer supports credit card tokens, only payment tokens valid for Apple Pay etc., too

Risk Reduction: (New) Dependencies

Quality Criteria for software libraries (and services)

- Number of Maintainers / Developers
- Actively Developed?
- How many users?
  - Packagist shows installation count
- Where is a library being installed from?
  - GitHub, self-hosted svn server? -> Availability
- Alternatives / how easy to replace? Complexity?
  - Could you take over maintenance?

Risk Reduction: Compatibility

Semantic Versioning (Semver) promises Compatibility

x.y.z

- Must be used consistently
- Only valuable if BC/Compatibility promise formalized
    - See http://symfony.com/doc/current/contributing/code/bc.html
- Otherwise choose narrower Version Constraints, check more frequently
    - e.g. ~1.2.3 instead of ^1.2.3

Risk Reduction: Compatibility

- Automated
  - Tests
  - Static Analysis
- Manual
  - Read Changelogs (and write them!)
  - Experience which libraries break BC

Nils Adermann
@naderman

Risk Reduction: Compatibility

- “`composer update`”
  - no isolation of problems unless run very frequently
- “`composer update <package...>`”
  - explicit conscious updates
- “`composer update --dry-run [<package...>]`”
  - Understanding and preparing effects of updates

How do partial updates work?

```
{   “name”: “zebra/zebra”,
    “require”: {
        “horse/horse”: “^1.0” }}

{   “name”: “giraffe/giraffe”,
    “require”: {
        “duck/duck”: “^1.0” }}
```

How do partial updates work?

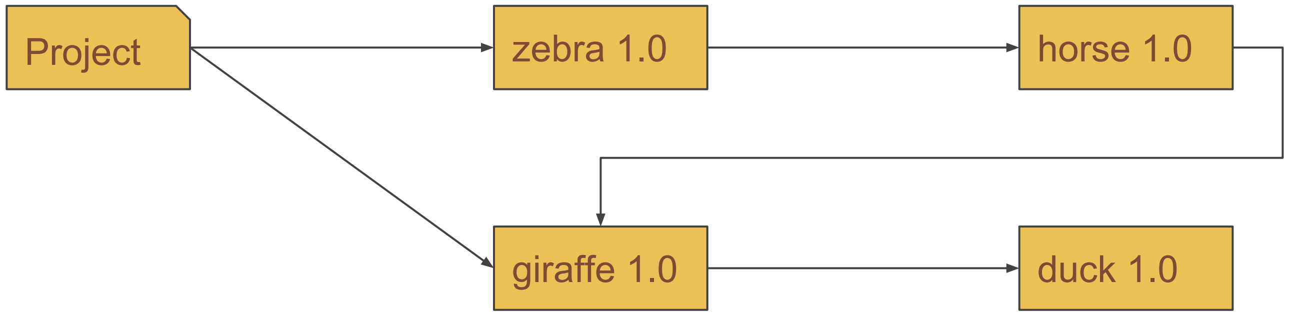
```
{   “name”: “horse/horse”,
    “require”: {
        “giraffe/giraffe”: “^1.0” }}

{   “name”: “duck/duck”,
    “require”: {}}
```

How do partial updates work?

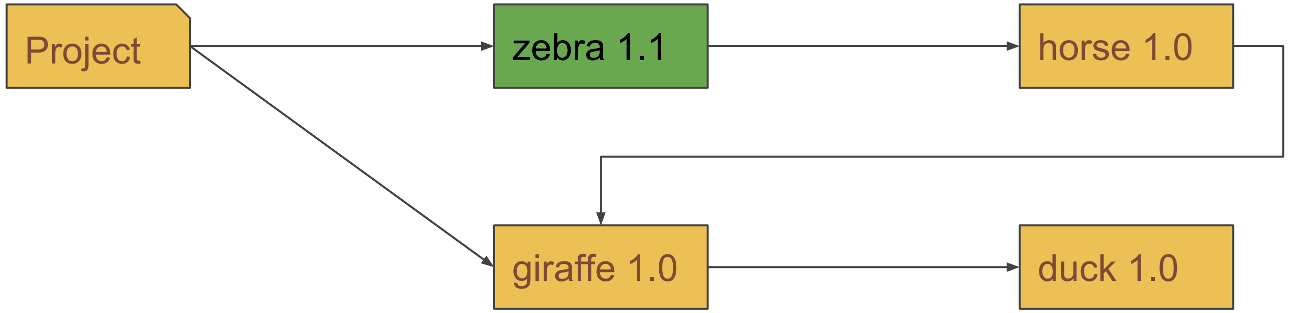
```
{
    “name”: “my-project”,
    “require”: {
        “zebra/zebra”: “^1.0”,
        “giraffe/giraffe”: “^1.0”
    }
}
```

How do partial updates work?



```
Now each package releases 1.1
```

How do partial updates work?

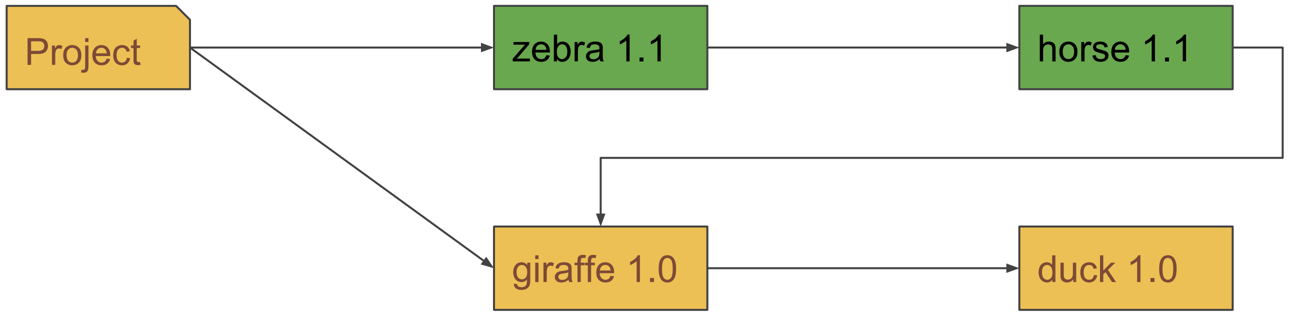


```
$ composer update --dry-run zebra/zebra
    Updating zebra/zebra (1.0 -> 1.1)
```

PRIVATE PACKAGIST

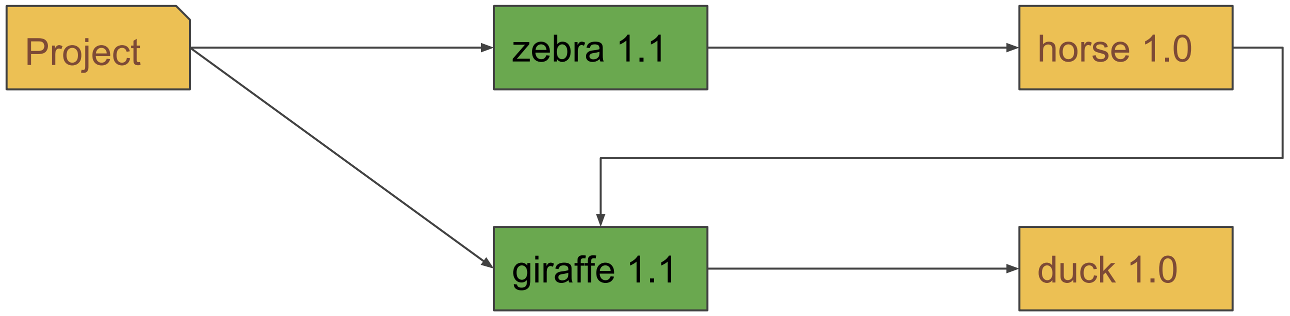
Nils Adermann
@naderman

How do partial updates work?



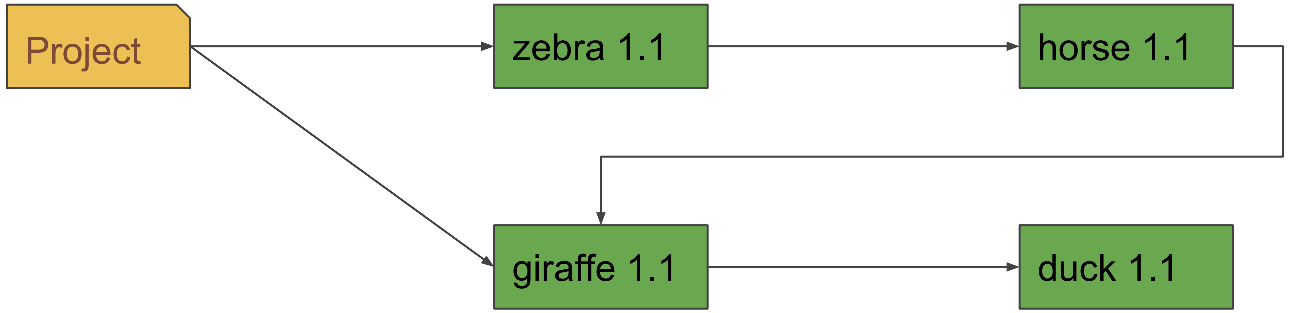
```
$ composer update --dry-run zebra/zebra --with-dependencies
   Updating horse/horse (1.0 -> 1.1)
   Updating zebra/zebra (1.0 -> 1.1)
```

How do partial updates work?



```
$ composer update --dry-run zebra/zebra giraffe/giraffe
   Updating zebra/zebra (1.0 -> 1.1)
   Updating giraffe/giraffe (1.0 -> 1.1)
```

How do partial updates work?



```
$ composer update zebra/zebra giraffe/giraffe --with-dependencies
    Updating duck/duck (1.0 -> 1.1)
    Updating giraffe/giraffe (1.0 -> 1.1)
    Updating horse/horse (1.0 -> 1.1)
    Updating zebra/zebra (1.0 -> 1.1)
```

The Lock File

- Contents
  - all dependencies including transitive dependencies
  - Exact version for every package
  - download URLs (source, dist, mirrors)
  - Hashes of files

- Purpose
  - Reproducibility across teams, users and servers
  - Isolation of bug reports to code vs. potential dependency breaks
  - Transparency through explicit updating process

Commit The Lock File

- If you don't
  - composer install without a lock file is a composer update
  - Affects Assembly
    - Conflict can randomly occur on install
    - You may not get the same code
  - You no longer manage change
    Change is managing you!

- The lock file exists to be commited!

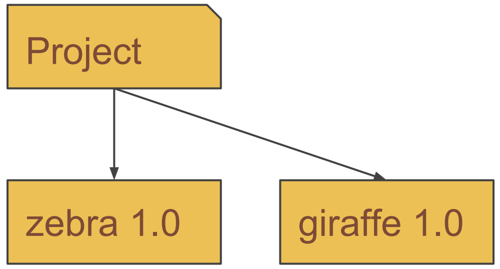
Nils Adermann
@naderman

The Lock file will conflict

How to resolve lock merge conflicts?

- composer.lock cannot be merged without conflicts
  - contains hash over relevant composer.json values
- `git checkout <refspec> -- composer.lock`
  - `git checkout master -- composer.lock`
- Repeat: `composer update <list of deps>`
  - Store parameters in commit message
  - Separate commit for the lock file update

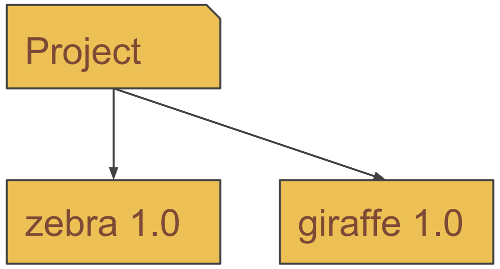
Day 0: “Initial Commit”



master

composer.lock

- zebra	1.0
- giraffe	1.0

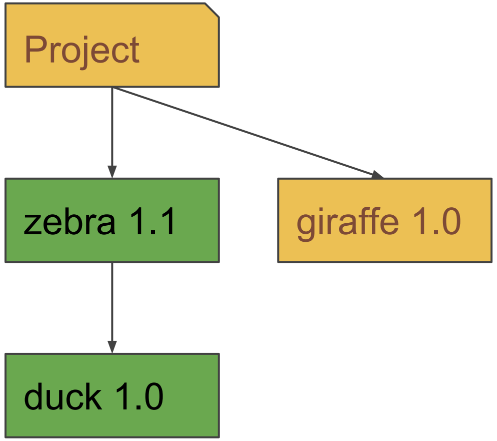


dna-upgrade

composer.lock

- zebra	1.0
- giraffe	1.0

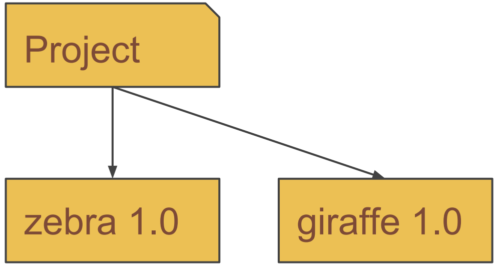
Week 2: Strange new zebras require duck



master

composer.lock

- zebra	1.1
- giraffe	1.0
- duck	1.0



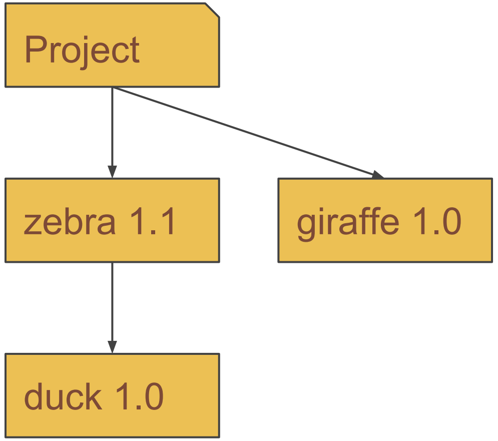
dna-upgrade

composer.lock

- zebra	1.0
- giraffe	1.0

Week 3: Duck 2.0

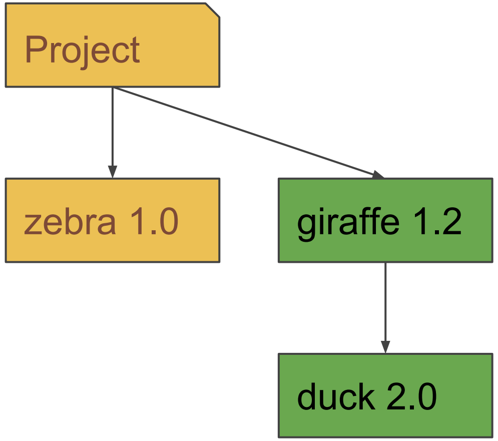
Week 4: Giraffe evolves to require duck 2.0



master

composer.lock

- zebra	1.1
- giraffe	1.0
- duck	1.0

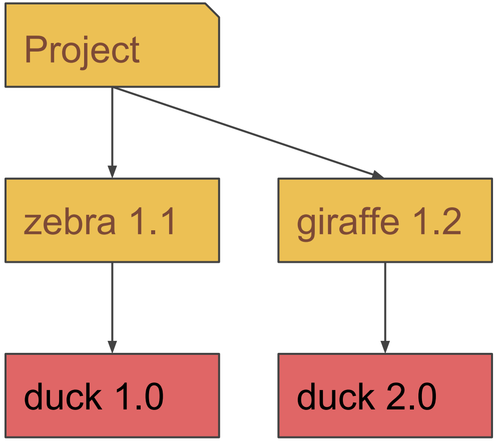


dna-upgrade

composer.lock

- zebra	1.0
- giraffe	1.2
- duck	2.0

Text-based Merge

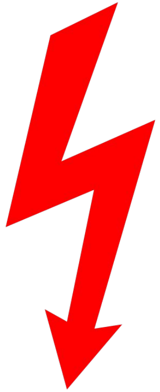


master

composer.lock

- zebra	1.1
- giraffe	1.2
- duck	**1.0**
- duck	**2.0**

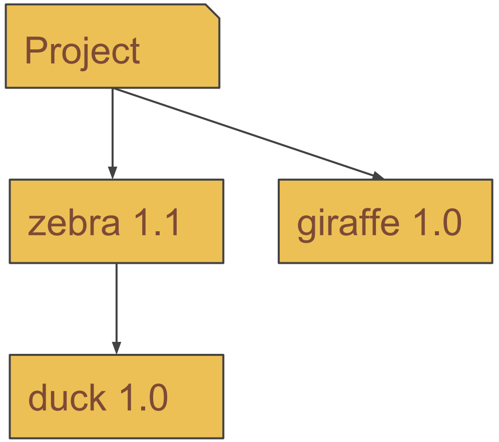
Merge results in invalid dependencies



Nils Adermann
@naderman

Reset composer.lock

```
git checkout <refspec> -- composer.lock
git checkout master -- composer.lock
```


$$\begin{tikzcd}
\text{Project} \arrow[d] \arrow[dr] & \\
\text{zebra 1.1} \arrow[d] & \text{giraffe 1.0} \\
\text{duck 1.0} &
\end{tikzcd}$$

dna-upgrade

composer.lock

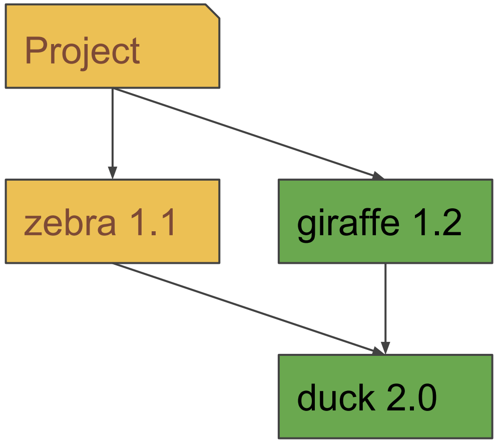
- zebra	1.1
- giraffe	1.0
- duck	1.0

PRIVATE PACKAGIST

Nils Adermann
@naderman

Apply the update again

```
composer update giraffe
  --with-dependencies
```



master

composer.lock

- zebra	1.1
- giraffe	1.2
- duck	2.0

Risk Analysis: Compliance / Legal

Affects Change Management

Examples:

- Viral Copy-Left License not compatible with proprietary product
- Terms of Service
  - May I use an API for my services?
    Cloudflare / packagist.org
  - How much time do I have when a supplier terminates the service?
  - SLA with sufficient support?

Risk Minimization: Compliance / Legal

- Software dependency license must fit product license or customer requirements
  - `composer licenses`
  - Private Packagist License Review

- Terms of Service / SLA / Contracts
  - Criteria for selection
  - Negotiable
  - Strong dependencies justify financial expenses to create security

Nils Adermann
@naderman

Assessing & Managing Risk

- Formulate a Plan B
- Identify problems which are probable and which have great effects

- **Dependencies are great!** They can save tons of money and time
- Only spend resources on reducing risk until the risk is acceptable

Summary

- composer update [--dry-run] <package>
- git checkout <branch> -- composer.lock
- Formalize BC promises for users of your libraries
- SemVer: Don’t be afraid to increase the major version
- Document changes to dependencies
- Have a plan B
- Don’t waste resources on potential problems which are unlikely to occur or have insignificant effects
- **Dependencies are great!**
  Benefit usually greater than cost

Developers must consider dependency management from a business perspective
Business / Management must not ignore risk from software dependencies

Thank you!
Questions / Feedback?

https://joind.in/talk/8f188

https://packagist.com

10% off first 12 months with code **phptek2018**

E-Mail: n.adermann@packagist.com
Twitter: @naderman