SymfonyCon Vienna 2024

Composer Behind the Scenes

Nils Adermann
@naderman

Private Packagist
https://packagist.com

Why is Composer 2 so much faster?

Why is Composer 2 so much faster?

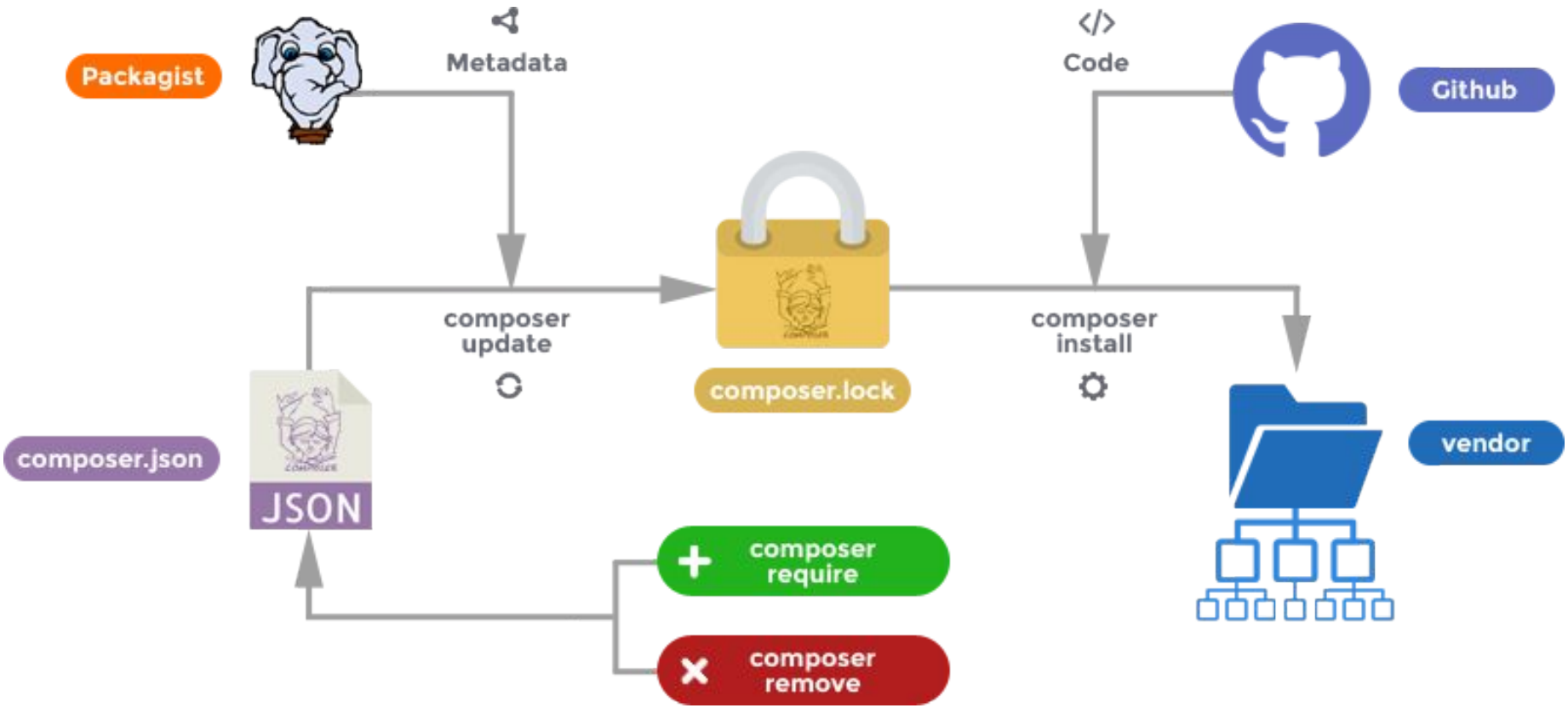
- Benchmarks
  - install 30% to 50% faster
  - update 30% to 90% faster & drop in memory usage of 70% to 98%

- Easy answers
  - parallel downloads, making use of HTTP/2 features
  - parallel archive extraction
  - more efficient metadata format

  - doesn't really explain improvements for update

https://blog.packagist.com/composer-2-0-is-now-available/
https://susi.dev/composer2-perf
https://developers.ibexa.co/blog/benchmarks-of-composer-2.0-vs-1.10
https://metadrop.net/es/articulos/drupal-composer-2

Separating update & install - Declaring state over manipulating state

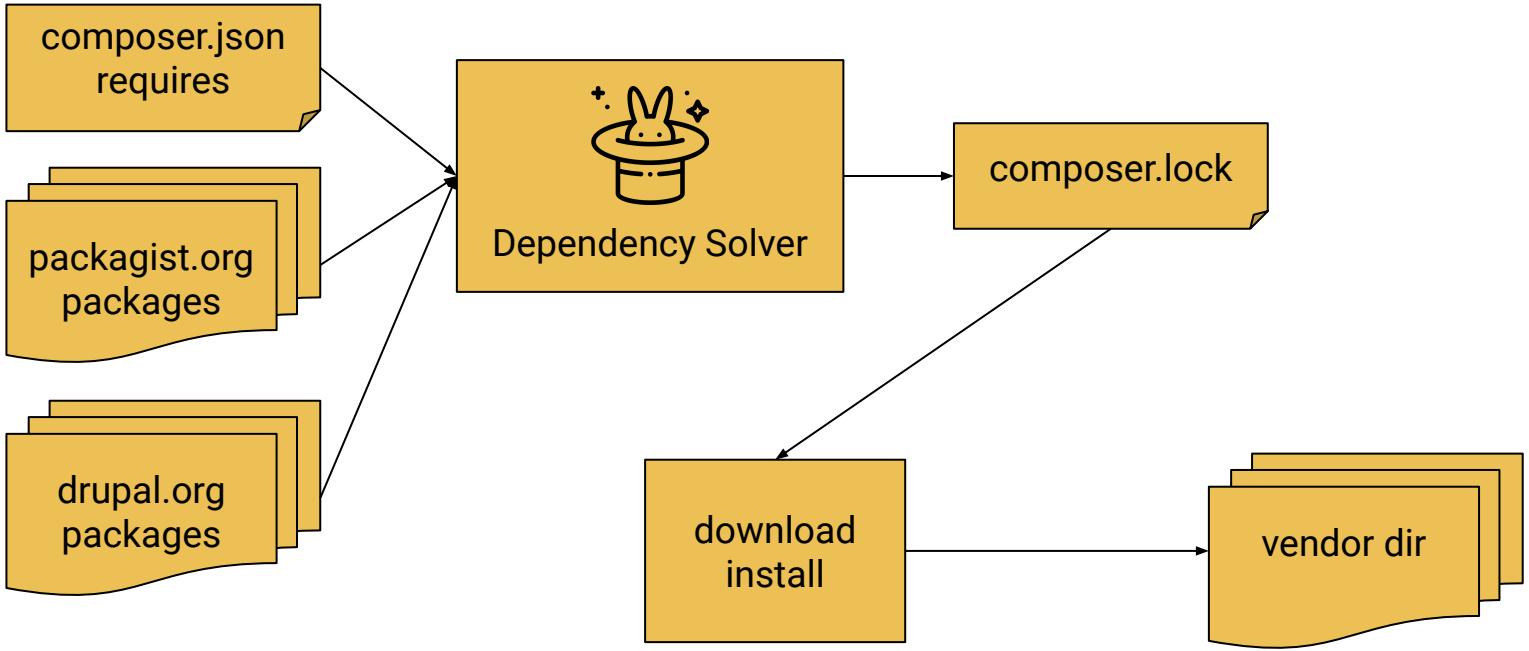


Separating update & install

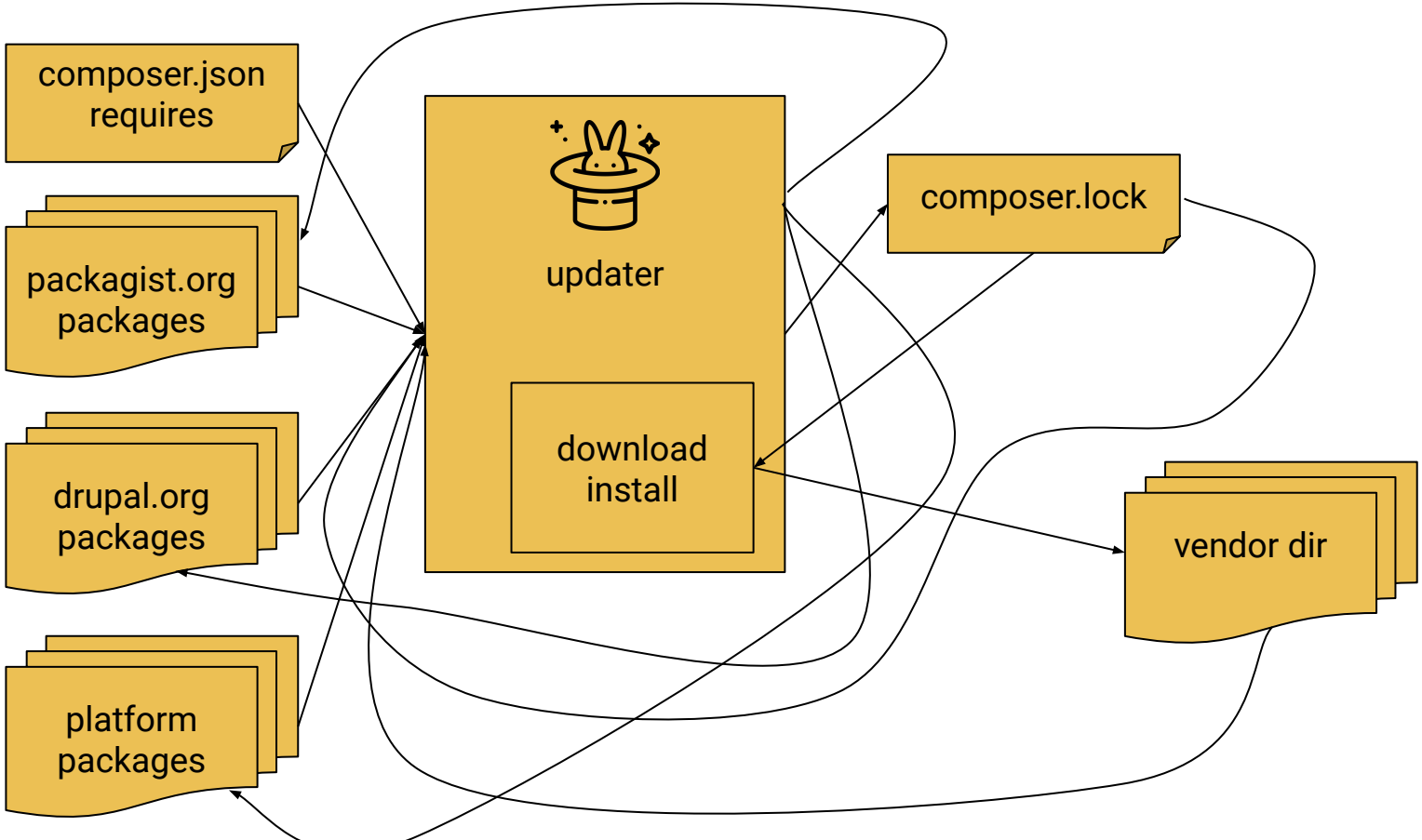
vendor
    symfony/http-foundation: 7.2.0 previous local upgrade attempt
composer.lock
    symfony/http-foundation: 6.4.16 old production state
composer.json
    symfony/http-foundation: 7.1.* limited upgrade for now, because of 6.3 issues

```
naderman@saumur:~/projects/composer/test/symfony-http-foundation$ composer update
Loading composer repositories with package information
Updating dependencies
Lock file operations: 0 installs, 1 update, 0 removals
 - Upgrading symfony/http-foundation (v6.4.16 => v7.1.9)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 3 installs, 1 update, 1 removal
 - Removing symfony/deprecation-contracts (v3.5.1)
 - Downgrading symfony/http-foundation (v7.2.0 => v7.1.9): Extracting archive
Generating autoload files
6 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
```

composer update: The idea



composer update: Reality in Composer 1

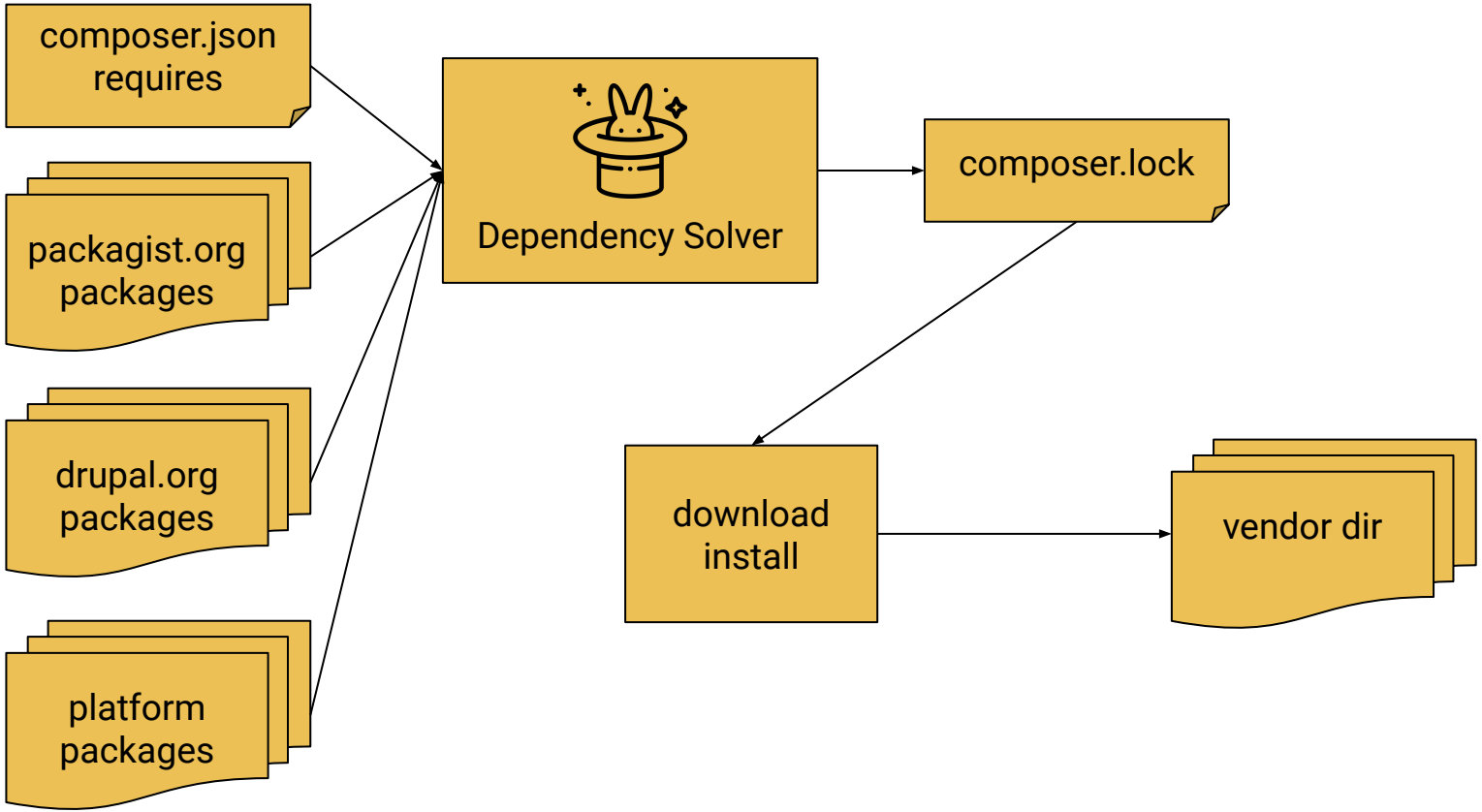


composer update: Reality in Composer 1 - aka some terrible ideas

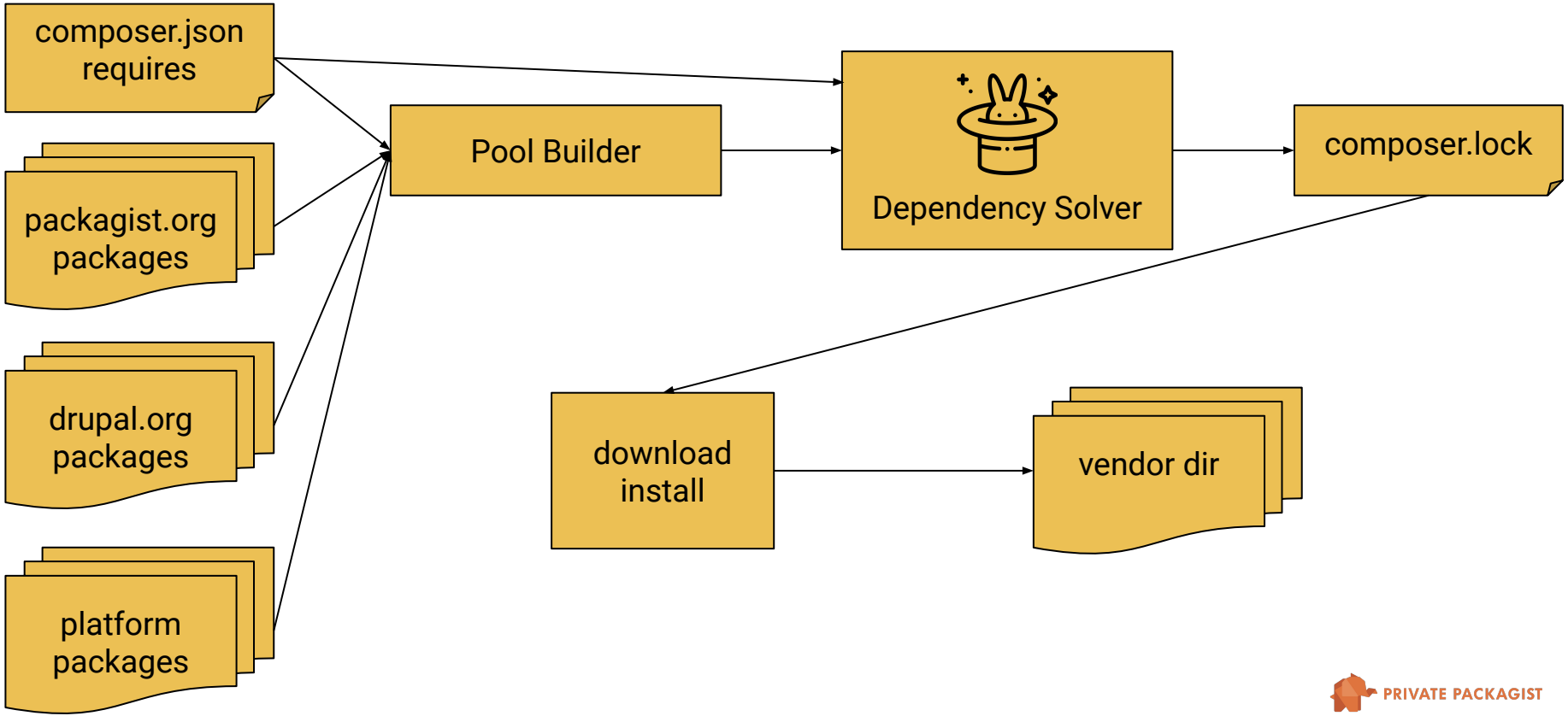
- Idea: Solver only loads what it needs when it gets to that point
  - Solution: **Lazy load packages** while creating memory representation in solver
  - Problems
    - Solver just waits for same info at a later point
    - Impossible to reduce set of packages before generating dependencies
    - Parallelized network access becomes hard to manage

- Idea: Avoid downloading metadata and packages unnecessarily and protect from loss of packages
  - Solution: **use vendor/ and composer.lock metadata in solver**
  - Problems
    - Duplicate metadata
    - Unclear which “version” to use / when to update metadata
    - Confusing results where packages that no longer exist don’t get removed
    - Inconsistent behavior depending on local state

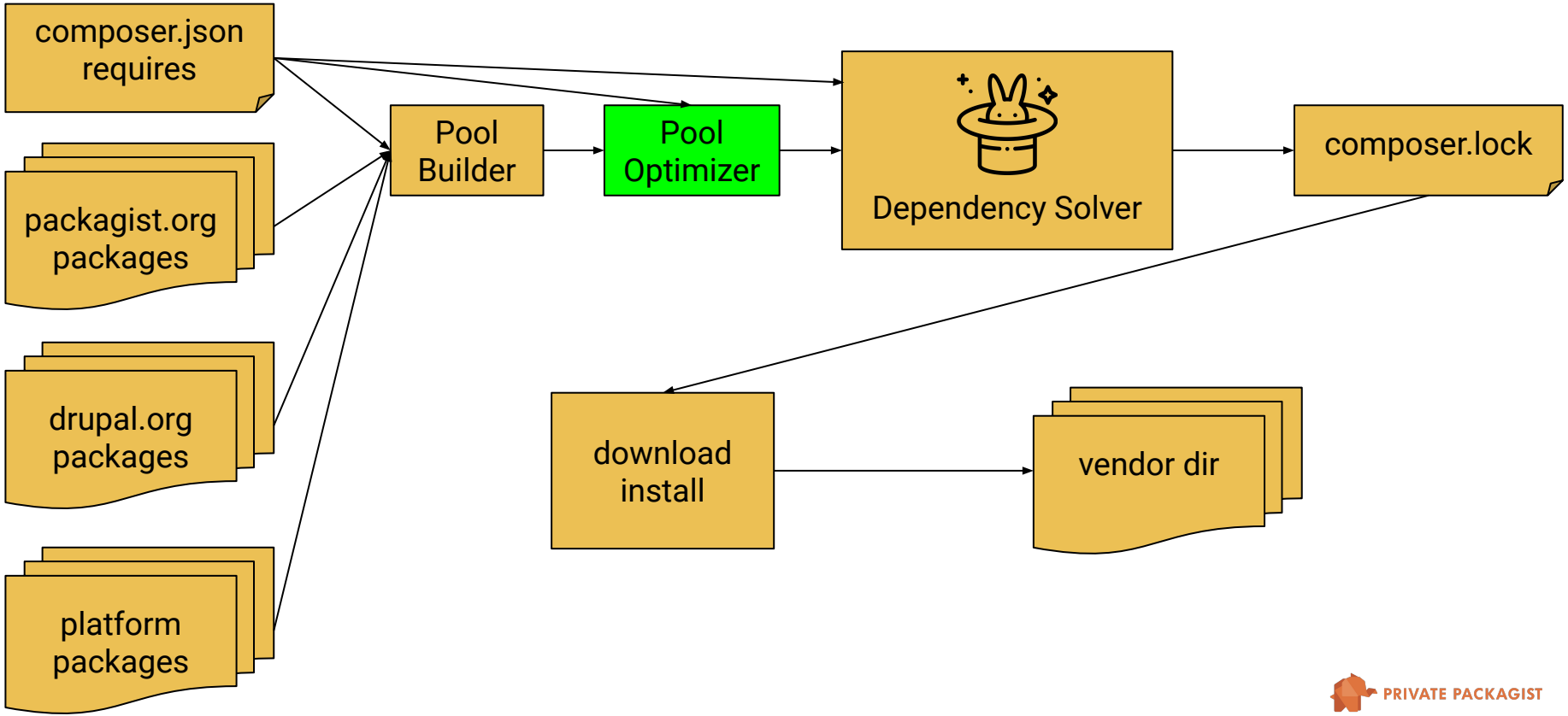
composer update: The idea



composer update: Reality in Composer 2



composer update: Reality in Composer 2.2



composer update: Reality in Composer 2.2

- **Pool**
  - Simple array of all package versions to be passed to the Dependency Solver

- **Pool Builder** collects package metadata from all sources/repositories
  - Takes root composer.json requires into account
  - Avoids loading metadata that is definitely not installable
  - Tries to limit how many versions of a package get loaded by tracking constraints

- **Pool Optimizer**
  - identifies versions with identical constraints and reduces them into one
  - Shout out to Jason Woods / driskell for two additions based on Drupal projects
    - Filters impossible packages out https://github.com/composer/composer/pull/9620/files
    - Do not load replaced targets https://github.com/composer/composer/pull/11449
  - more future improvements possible!

What’s in the Dependency Solver?

And why does reducing loaded package versions matter so much?

Boolean Algebra

- Notation
  - OR: $\vee$
  - AND: $\wedge$
  - NOT: $\neg$
- Laws
  - Associativity: $A \vee (B \vee C) = (A \vee B) \vee C$
  - Commutativity: $A \vee B = B \vee A$
  - Distributivity: $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
  - Absorption: $A \vee (A \wedge B) = A$
  - Complementation 2: $A \vee \neg A = \text{TRUE}$
  - etc.

Conjunctive Normal Form

- $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$
- $(A \vee B)$ is a clause
- A, B, ¬B, C, D, ¬D, E are literals
- A, B, C, D are atoms

Every propositional formula can be converted into an equivalent formula that is in CNF. This transformation is based on rules about logical equivalences: the double negative law, De Morgan's laws, and the distributive law.

What's in the Dependency Solver?

- SAT Solver
  - boolean SATisfiability
  - Is there a set of values for a boolean formula that results in its evaluation to true
  - (A ⋀ B) is satisfiable with A=TRUE and B=TRUE.
  - (A ⋀ B ⋀ ¬A) is not satisfiable because A cannot be both TRUE and FALSE.
- Why a SAT Solver?
  - Port from libzypp / zypper in SUSE back in 2011
  - EDOS project https://www.mancoosi.org/edos/ - Package Installation is NP-Complete
    - https://www.mancoosi.org/edos/algorithmic/#toc15 (For the really interested here you can see someone encode any 3SAT problem as a debian or RPM package installation)

Dependencies as a SAT Problem

- Each version of a package is a literal
  - Package A v1.0.0 should be present: **A-1.0.0**
  - Package A v1.0.0 should not be present: **¬A-1.0.0**

- A-1.0.0 requires B-1.0.0: **(¬A-1.0.0 ∨ B-1.0.0)**
- A-1.0.0 conflicts with B-1.0.0: **(¬A-1.0.0 ∨ ¬B-1.0.0)**
- C-1.0.0 and D-1.0.0 provide B-1.0 and A-1.0 requires B-1.0
  (¬A-1.0.0 ∨ C-1.0.0 ∨ D-1.0.0)
- C-1.0.0 replaces B-1.0 and A-1.0 requires B-1.0
  (¬C-1.0.0 ∨ ¬B-1.0.0) ∧ (¬A-1.0.0 ∨ B-1.0.0 ∨ C-1.0.0)

Fewer packages/versions to analyze? => fewer literals, fewer clauses, less memory

Dependencies as a SAT Problem: Example

project requires A *, A 1.0.0 requires B * and C *, B requires C *

1.		**(A-1.0.0)**	∧ (¬A-1.0.0 ∨ B-1.0.0)	∧ (¬B-1.0.0 ∨ C-1.0.0)	∧ (¬A-1.0.0 ∨ C-1.0.0)
2.	**A-1.0.0=true**	**true**	∧ (**false** ∨ B-1.0.0)	∧ (¬B-1.0.0 ∨ C-1.0.0)	∧ (**false** ∨ C-1.0.0)
3.		true	∧ **(B-1.0.0)**	∧ (¬B-1.0.0 ∨ C-1.0.0)	∧ **(C-1.0.0)**
4.	**B-1.0.0=true**	true	∧ **true**	∧ (**false** ∨ C-1.0.0)	∧ (C-1.0.0)
5.		true	∧ **true**	∧ **(C-1.0.0)**	∧ **(C-1.0.0)**
6.	**C-1.0.0=true**	true	∧ true	∧ **true**	∧ **true**

Solved: Install A 1.0.0, B 1.0.0, C 1.0.0

Dependencies as a SAT Problem: Example

project requires A *, A 1.0.0 requires B * and C *, B **conflicts** with C *

1.		**(A-1.0.0)**	∧ (¬A-1.0.0 ∨ B-1.0.0)	∧ (¬B-1.0.0 ∨ ¬C-1.0.0)	∧ (¬A-1.0.0 ∨ C-1.0.0)
2.	**A-1.0.0=true**	**true**	∧ (**false** ∨ B-1.0.0)	∧ (¬B-1.0.0 ∨ ¬C-1.0.0)	∧ (**false** ∨ C-1.0.0)
3.		true	∧ **(B-1.0.0)**	∧ (¬B-1.0.0 ∨ ¬C-1.0.0)	∧ **(C-1.0.0)**
4.	**B-1.0.0=true**	true	∧ **true**	∧ (**false** ∨ ¬C-1.0.0)	∧ (C-1.0.0)
5.		true	∧ **true**	∧ **(¬C-1.0.0)**	∧ **(C-1.0.0)**
6.	**C-1.0.0=false**	true	∧ true	∧ **true**	∧ **false**

Conflict! A requires C, but B conflicts with C.

Free Choices / Policy

- Policy determines precedence of solution attempts for free choices
  - By default always try the highest version number first
  - Can be altered with flags like --prefer-lowest (reverse)

Dependencies as a SAT Problem: Example with free choice

project requires A *, A 1.0.0 requires B *, B 2.0.0 requires C *

1.		**(A-1.0.0)**	⋀ (¬A-1.0.0 ⋁ B-1.0.0 ⋁ B-2.0.0)	⋀ (¬B-2.0.0 ⋁ C-1.0.0)	
2.	**A-1.0.0=true**	**true**	⋀ (**false** ⋁ B-1.0.0 ⋁ B-2.0.0)	⋀ (¬B-2.0.0 ⋁ C-1.0.0)	
3.		true	⋀ **(B-1.0.0 ⋁ B-2.0.0)**	⋀ (¬B-2.0.0 ⋁ C-1.0.0)	
4.	**B-2.0.0=true**	true	⋀ **(B-1.0.0⋁ true)**	⋀ (**false** ⋁ C-1.0.0)	**[Policy]**
5.		true	⋀ **true**	⋀ **(C-1.0.0)**	
6.	**C-1.0.0=true**	true	⋀ true	⋀ **true**	

Solved: Install A 1.0.0, **B 2.0.0**, C 1.0.0

Implementation

- Each package version object gets an integer id
- \Composer\DependencyResolver\Rule contains an array of literals
  - absolute value is the id, sign is used for negation
- \Composer\DependencyResolver\Solver::solve()
  - generates rules based on package pool and policy
  - finds solution with runSat()
  - returns new lock file state
- \Composer\DependencyResolver\DefaultPolicy
  - implements free choice decisions
  - handles options like --prefer-lowest or --prefer-stable

Representing dependencies/conflicts more efficiently

Regular requirements and conflicts

```
foo/bar 1.0 requires baz/qux ^2.0
foo/bar 1.0 conflicts with baz/qux ^2.0
```

```
(¬foo/bar 1.0 ∨ baz/qux 2.0.0 ∨ baz/qux 2.0.1 ∨ baz/qux 2.1.0)
(¬foo/bar 1.0 ∨ ¬baz/qux 2.0.0) ∧ (¬foo/bar 1.0 ∨ ¬baz/qux 2.0.1) ∧
(¬foo/bar 1.0 ∨ ¬baz/qux 2.1.0)
```

You can only install one version of a package

=> Composer automatically generates a conflict for each pair of versions

```
foo/bar 1.0, 1.1, 1.2
```

```
( ¬foo/bar 1.0 ∨ ¬foo/bar 1.1) ∧ (¬foo/bar 1.0 ∨ ¬foo/bar 1.2) ∧
( ¬foo/bar 1.1 ∨ ¬foo/bar 1.2)
```

Extreme Growth $\binom{n}{2} = \frac{n!}{2(n-2)!}$

				Symfony	
	3 versions	6 versions	100 versions	500 versions	1000 versions
Composer 1	3 rules	15 rules	4,950 rules	**124,750** rules	**499,500** rules
Composer 2	1 rule	1 rule	1 rule	1 rule	1 rule

Composer 2.0 uses a special single multi conflict rule representation for all of these rules

```
foo/bar 1.0, 1.1, 1.2
```

```
oneof(foo/bar 1.0, foo/bar 1.1,foo/bar 1.2)
```

Partial Updates

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

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

Partial Updates

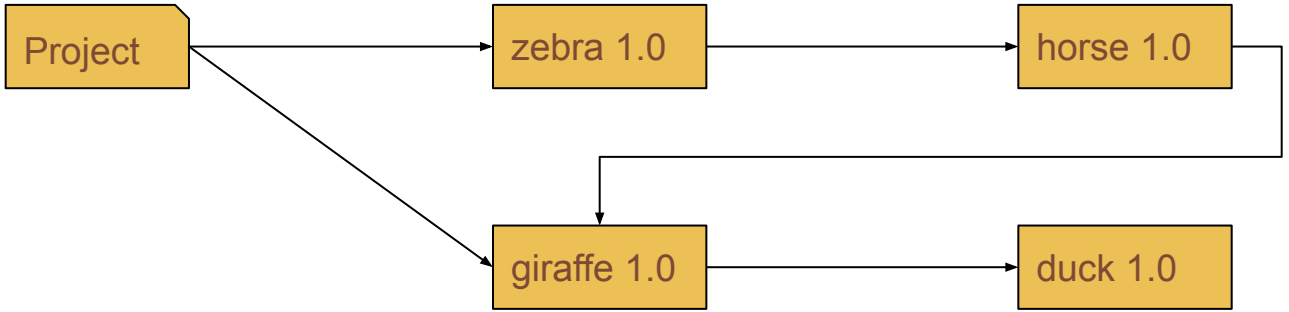
```
{   “name”: “horse/horse”,
    “require”: {
        “giraffe/giraffe”: “^1.0” }}

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

Partial Updates

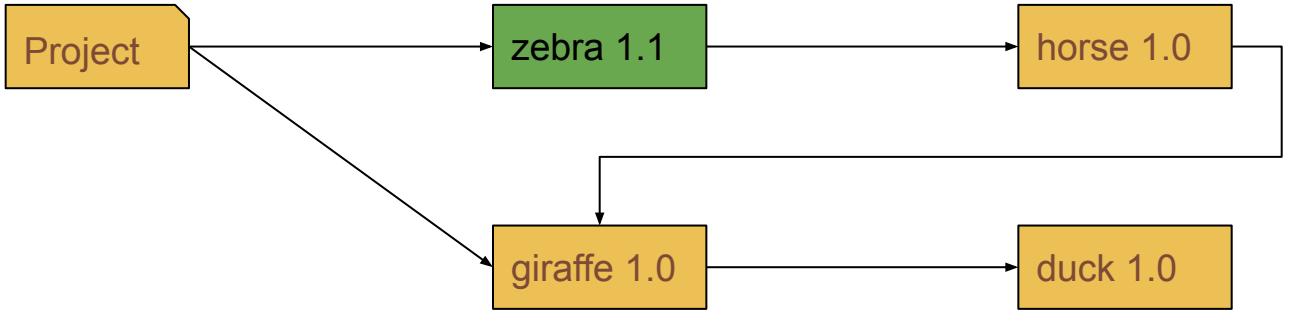
```
{
    “name”: “my-project”,
    “require”: {
        “zebra/zebra”: “^1.0”,
        “giraffe/giraffe”: “^1.0”
    }
}
```

Partial Updates



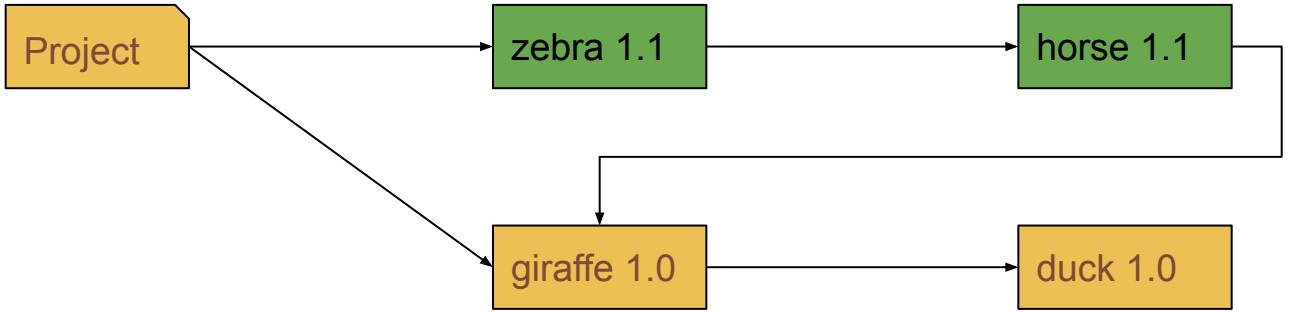
Now each package releases 1.1

Partial Updates



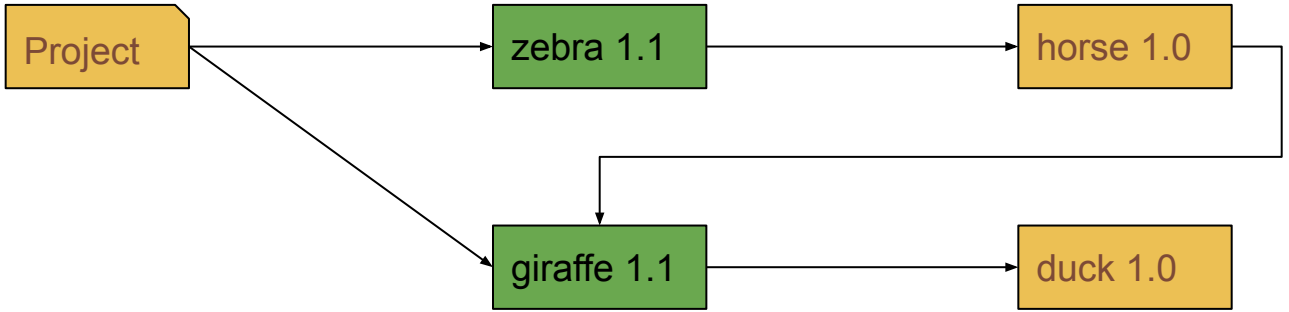

```
$ composer update --dry-run zebra/zebra
    Updating zebra/zebra (1.0 -> 1.1)
```

Partial Updates



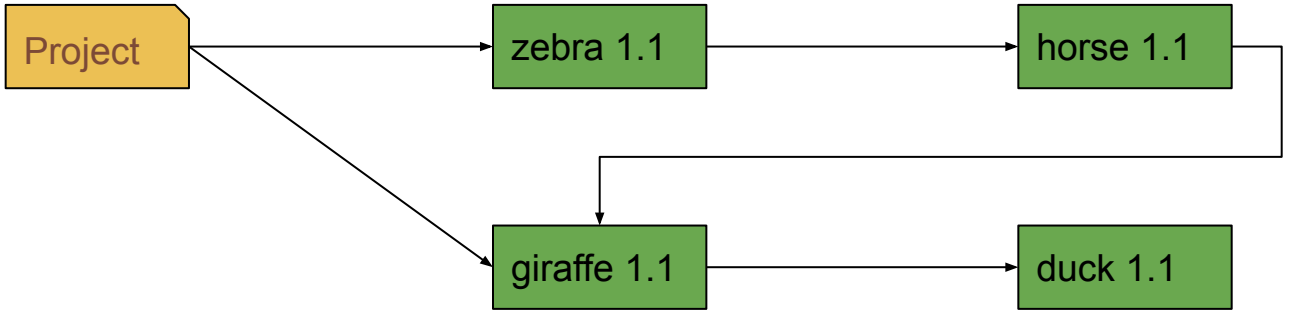

```
$ composer update --dry-run zebra/zebra --with-dependencies
    Updating horse/horse (1.0 -> 1.1)
    Updating zebra/zebra (1.0 -> 1.1)
```

Partial Updates



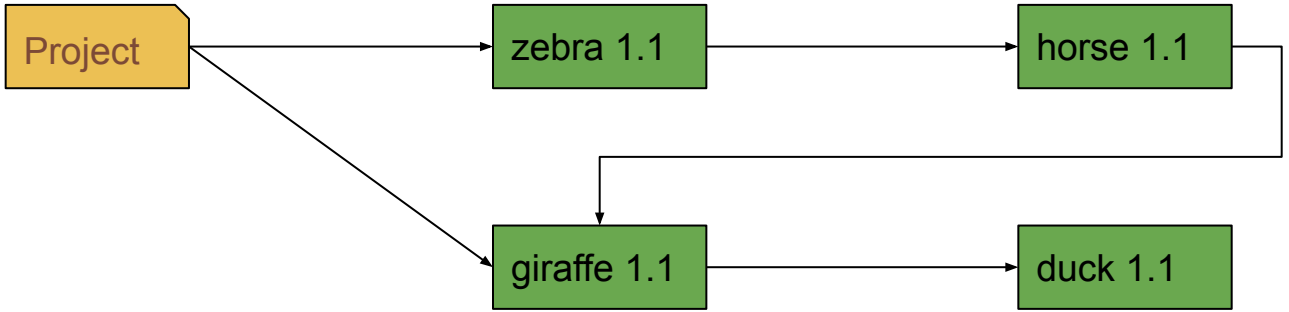

```
$ composer update --dry-run zebra/zebra giraffe/giraffe
    Updating zebra/zebra (1.0 -> 1.1)
    Updating giraffe/giraffe (1.0 -> 1.1)
```

Partial Updates



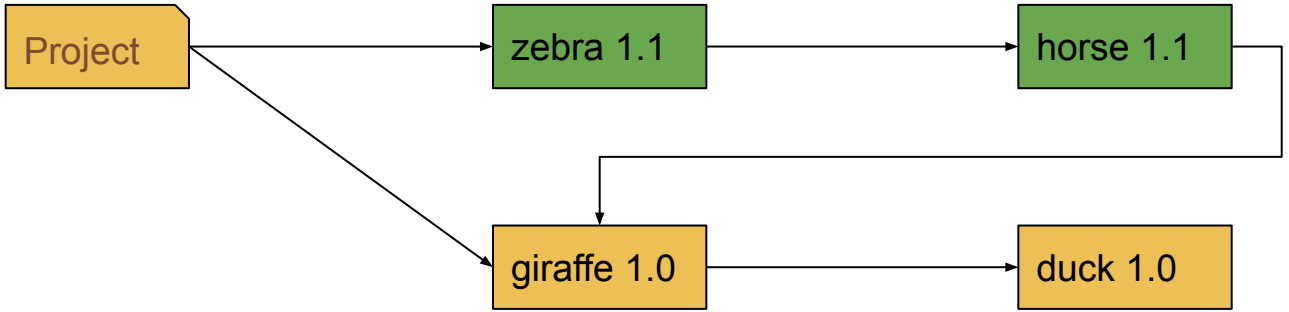

```
$ composer update zebra/zebra giraffe/giraffe --with-dependencies
    Updating duck/duck (1.0 -> 1.1)
    Updating giraffe/giraffe (1.0 -> 1.1)
    Updating horse/horse (1.0 -> 1.1)
    Updating zebra/zebra (1.0 -> 1.1)
```

Partial Updates



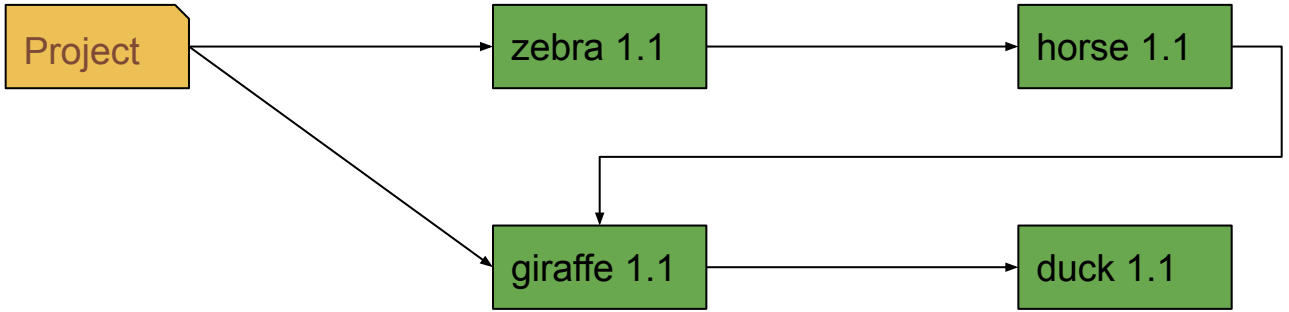

```
$ composer update zebra/zebra --with-all-dependencies
    Updating duck/duck (1.0 -> 1.1)
    Updating giraffe/giraffe (1.0 -> 1.1)
    Updating horse/horse (1.0 -> 1.1)
    Updating zebra/zebra (1.0 -> 1.1)
```

Partial Updates




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

Partial Updates




```
$ composer update zebra/zebra --with-all-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)
```

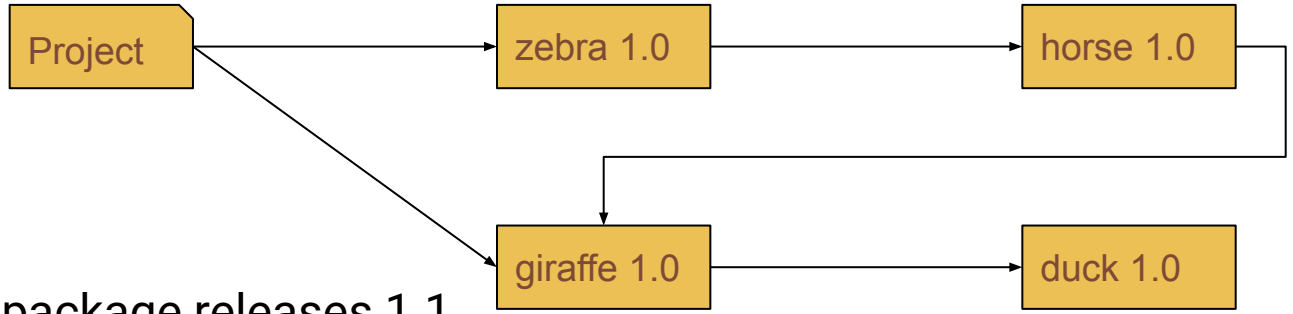
Upcoming Features & Future Plans

- Plans?
  - **Keep things stable and compatible**
  - Composer 1 shut down on packagist.org
    - Feb 1st, 2025: Composer 1 metadata becomes read-only
    - Aug 1st, 2025: Composer 1 metadata becomes unavailable
      - composer update on v1 will error, install from lock keeps working
  - Small improvements based on common workflows
  - Help users improve their projects' security

Small Improvements: An example

- --minimal-changes
  - Since Composer 2.7
  - Problem: I want to update one dependency, but there's a conflict, I need to update more, but I don't want to update everything
  - Solution: Partial updates with dependencies, but keeping them at the same version as the lock file if possible

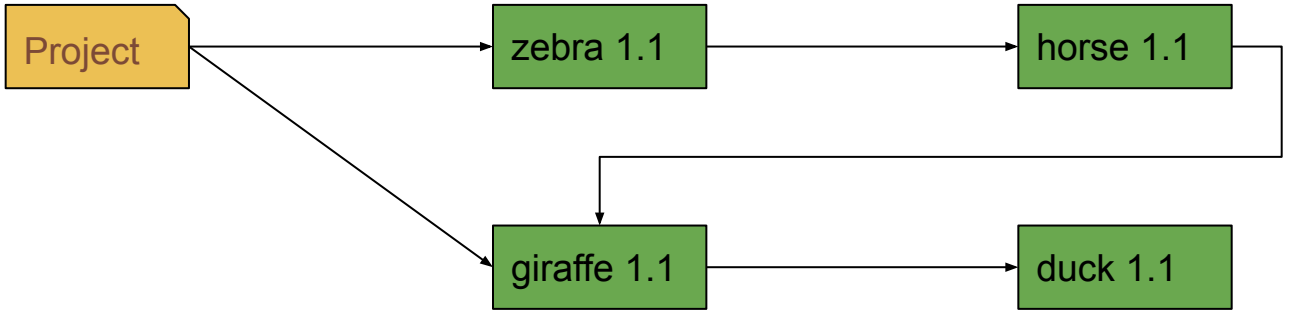
Small Improvements: An example



Now each package releases 1.1

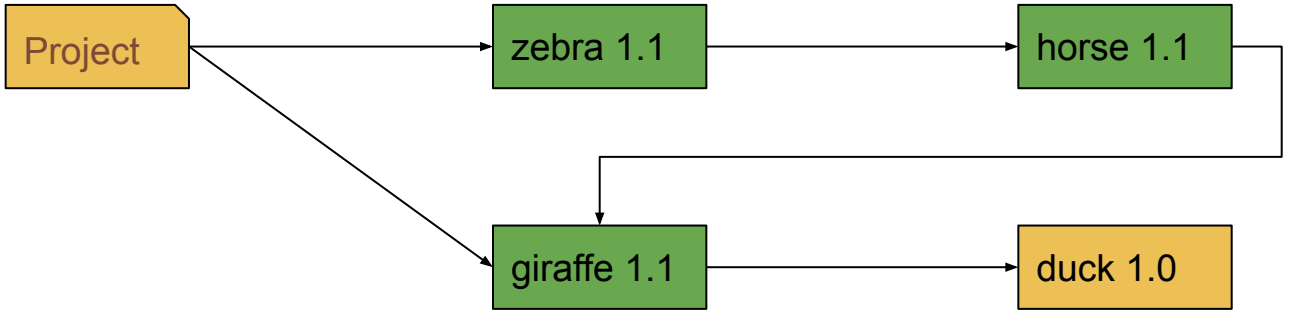
- zebra 1.1 requires horse ^1.1
- horse 1.1 requires giraffe ^1.1
- giraffe 1.1 still requires **duck ^1.0**

Small Improvements: An example




```
$ composer update zebra/zebra --with-all-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)
```

Small Improvements: An example




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

Small Improvements: An example

- --minimal-changes
  - Since Composer 2.7
  - Problem: I want to update one dependency, but there's a conflict, I need to update more, but I don't want to update everything
  - Solution: Partial updates with dependencies, but keeping them at the same version as the lock file if possible

Who could follow the beginning? Any idea how to implement this?

Small Improvements: An example

Who could follow the beginning? Any idea how to implement this?

- Set up the update the same way as if the option wasn't specified
- Make the policy pick locked version numbers before any other versions
- Result
  - Solver will try locked versions first
  - If locked versions are incompatible it will attempt to change versions

https://github.com/composer/composer/pull/11665

Improving Security for Users

- composer audit & packagist.org advisory database API
  - run by default on updates since 2.4 (October 2022)
- Should have the ability to block updating to vulnerable versions
  - currently possible by requiring roave/security-advisories
- Public UI access to & notifications for packagist.org audit/transparency log
- Built in SBOM support, currently only available with plugins

Packagist.org

- Metadata only
- No checksums for GitHub stored packages
  - https://github.com/sansecio/composer-integrity-plugin
- No signatures
- No way to upload code
- Tags can get recreated
- Packagist.org maintainer account takeover
  https://blog.packagist.com/packagist-org-maintainer-account-takeover/
  - Editing of source URLs no longer allowed beyond 50k installs

Improving Security for Users: Signatures

- Problem: GitHub archives not stable (>99% of packagist.org packages)
  - packagist.org may need to host code
    - no longer avoiding category of security issues related to hosting code
    - need to address build attestation
    - have to moderate uploaded content, potentially work intensive
  - sign contents of archives only
    - non-standard, so harder to implement
    - archive metadata may itself contain exploits, need to really know well which parts may be skipped

Improving Security for Users: Signatures

- Observing Drupal's initiative
  - https://www.drupal.org/project/infrastructure/issues/3325040 - Automatic Updates / TUF (The Update Framework)

- But: Signatures are not the holy grail
  - Don't solve important questions like can you even trust the party who signed the package?
  - Doesn't protect you from malicious maintainers (e.g. event-stream backdoor in 2018 or xz/liblzma in 2024)

    https://www.ntousakis.com/es-eurosec.pdf / https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/
    https://en.wikipedia.org/wiki/XZ_Utils_backdoor
  - TLS with GitHub already gives you quite a lot

PIE: PHP Installer for Extensions

- New installer for PHP extensions (replacement for PECL)
  - `pie install apcu/apcu`
- Extension metadata in composer.json served by packagist.org
  - https://packagist.org/extensions
- Funded by the PHP Foundation
  - Goal: shut down hardly maintained pecl.php.net
- Try it today, port your extensions, provide feedback
  - https://github.com/php/pie

Friday 15:15 SensioLabs Track “From Pickles to PIE Sweeten Your PHP Extension Installs” with Andreas Braun

Supply Chain Funding

- It’s your supply chain, fund it!

- Sponsor the PHP Foundation
  - https://thephp.foundation/
- Sponsor Symfony
  - https://symfony.com/sponsor
- Buy a Private Packagist subscription
  - https://packagist.com/

- Run `composer fund`

Some personal ideas / wishes

- Ways to define maintenance/support levels
  - Would be easy to work out when looking at a new project if things need urgent updates if you can check automatically which versions are still maintained
  - Could work well as an addition to composer audit
  - Would help in prioritizing updates or selecting automated updates

  - https://github.com/composer/composer/issues/8272 open since Aug 2019, help welcome!
    - unfortunately not as easy as it seems on a first look

Some personal ideas / wishes

- Improved support for tools
  - Problem: Dev tools, e.g. phpunit, have requirements
    - can potentially conflict with your own requirements for the same packages
    - can result in your project using lower/higher versions than you would otherwise use
  - Idea: Separate requirements for tools
    - Problems: some tools need to be resolved together some independently
    - Some tools must run in same scope, how do we make multiple versions of same package work?
  - Current workaround / alternative
    - PHPScoper + phar files, e.g. phpstan

Some personal ideas / wishes

- Better support for patches
  - Most widely used as cweagans/composer-patches
  - Please only use with locally versioned files
  - Currently bypasses Composer concepts like repositories
    - impossible to override
    - impossible to mirror and verify by tools like Private Packagist
  - At least uses Composer download mechanism now to support the same proxy settings
  - See https://github.com/cweagans/composer-patches/issues/358

Some personal ideas / wishes

- Conductor by Private Packagist
  - Full Composer support (plugins, scripts)
  - Built with PHP in mind, great default rules, full understanding of Composer capabilities
  - Interested? Talk to me at the conference, I'd love to hear
    - ideas you have
    - what problems you face with existing solutions
    - what do you use? did you build something yourself?
    - what are you dissatisfied with?
  - Sign up for waitlist at https://packagist.com/features/conductor

PRIVATE PACKAGIST

Questions / Feedback?

Private Packagist
https://packagist.com

E-Mail: n.adermann@packagist.com
Bluesky: @naderman.de
X: @naderman
Mastodon: @naderman@phpc.social