Developing and Deploying Magento with Composer: Best Practices

Nils Adermann - @naderman - n.adermann@packagist.com
Package Repositories

Third Parties
- Packagist - https://packagist.org
- Individual vendors’ repositories

Private Packages
- Any Git/svn/Mercurial/… repository
- GitHub, Bitbucket, GitLab, …
- Private Packagist - https://packagist.com
Leveraging Open-Source Packages

- Nearly 200k packages on packagist.org
  - Many useful well tested, maintained and secure packages
  - Large amounts of unmaintained, insecure, broken or poorly working PHP code
Leveraging Open-Source Packages

- Evaluate packages every time before you write code yourself

- Selection criteria
  - Quality of documentation (changelogs?)
  - Development activity (commits, issues, PRs)
  - Number of maintainers
  - Installation counts, GitHub stars
  - Complexity

- It’s all trade-offs - no golden rule
Magento Marketplace

- Apply similar criteria as for Open-Source packages

- Additional factors to consider for choosing packages
  - Cost
  - Licenses
  - Reviews / Ratings
  - Extension Quality Program
Using your private code with Composer

- "repositories": [
    {
        "type": "path", "url": "../core"
    }
],
- "repositories": [
    {
        "type": "vcs",
        "url": "https://github.com/naderman/symfony"
    }
],
- "repositories": [
    {
        "type": "composer",
        "url": "https://repo.packagist.com/my-org"
    }
]
Development Environment

Best Practices
Create-project instead of cloning

- composer create-project --repository-url=https://repo.magento.com/magento/project-community-edition <path>
  
  - composer.json will have the correct contents
    - different from forking the community edition

- magento/project-community-edition is a metapackage
  - no code
  - defines dependencies on a number of other packages

- Only clone if you’re trying to contribute to a repository directly
## Managing Updates: Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Example 1.0.0</th>
<th>Example 1.2.3-beta2</th>
<th>Example dev-master</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact Match:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wildcard Range:</td>
<td>1.0.*</td>
<td>2.*</td>
<td></td>
</tr>
<tr>
<td>Hyphen Range:</td>
<td>1.0-2.0</td>
<td>1.0.0 - 2.1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;=1.0.0 &lt;2.1</td>
<td>&gt;=1.0.0 &lt;=2.1.0</td>
<td></td>
</tr>
<tr>
<td>(Unbounded Range:</td>
<td></td>
<td></td>
<td>Bad!</td>
</tr>
<tr>
<td>Next Significant Release</td>
<td>~1.2</td>
<td>~1.2.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;=1.2.0 &lt;2.0.0</td>
<td>&gt;=1.2.3 &lt;1.3.0</td>
<td></td>
</tr>
<tr>
<td>Caret/Semver Operator</td>
<td>^1.2</td>
<td>^1.2.3</td>
<td>Best Choice for Libs</td>
</tr>
<tr>
<td></td>
<td>&gt;=1.2.0 &lt;2.0.0</td>
<td>&gt;=1.2.3 &lt;2.0.0</td>
<td></td>
</tr>
</tbody>
</table>

Operators: “ “ AND, “||” OR
Managing Updates: Stabilities

- **Order**
  
  dev -> alpha -> beta -> RC -> stable

- **Automatically from tags**
  
  1.2.3 -> stable
  1.3.0-beta3 -> beta

- **Automatically from branches**
  
  Branch -> Version (Stability)
  2.0 -> 2.0.x-dev (dev)
  master -> dev-master (dev)
  myfeature -> dev-myfeature (dev)

- **Choosing**
  
  “foo/bar”: “1.3.*@beta”
  “foo/bar”: “2.0.x-dev”

  “minimum-stability”: “alpha”
Managing Updates: Semantic Versioning

x.y.z
(BC-break).(new functionality).(bug fix)

https://semver.org/
Managing Updates: Semantic Versioning

Promise of Compatibility

**X.Y.Z**

- Must be used consistently
  Dare to increment X!
- Only valuable if BC/Compatibility promise formalized
  - Document in Changelog
Updating

- 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
  - Read CHANGELOGs
  - composer outdated
Managing Updates: Unexpected results

- `composer why [--tree] foo/bar`
  mydep/here 1.2.3 requires foo/bar (^1.0.3)

- `composer why-not [--tree] foo/bar ^1.2`
  foo/bar 1.2.3 requires php (>=7.1.0 but 5.6.3 is installed)
Managing Updates: Partial Updates

```json
{
  "name": "zebra/zebra",
  "require": {
    "horse/horse": "^1.0"
  }
}

{
  "name": "giraffe/giraffe",
  "require": {
    "duck/duck": "^1.0"
  }
}
```
Managing Updates: Partial Updates

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

{
    "name": "duck/duck",
    "require": {}
}
```
Managing Updates: Partial Updates

```json
{
    "name": "my-project",
    "require": {
        "zebra/zebra": "^1.0",
        "giraffe/giraffe": "^1.0"
    }
}
```
Managing Updates: Partial Updates

Now each package releases 1.1
Managing Updates: Partial Updates

$ composer update --dry-run zebra/zebra
Updating zebra/zebra (1.0 -> 1.1)
Managing Updates: Partial Updates

$ composer update --dry-run zebra/zebra --with-dependencies
  Updating horse/horse (1.0 -> 1.1)
  Updating zebra/zebra (1.0 -> 1.1)
Managing Updates: Partial Updates

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

Every composer install without a lock file is a catastrophe waiting to happen
The Lock File Will Conflict
Day 0: “Initial Commit”

Project

zebra 1.0
-giraffe 1.0

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
Week 3: Duck 2.0
Week 4: Giraffe evolves, requires duck 2.0

Project

zebra 1.1

- zebra 1.1
- giraffe 1.0
- duck 1.0

master

composer.lock

Project

zebra 1.1

- 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

Project

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

master

composer.lock
- zebra 1.1
- giraffe 1.2
- duck 1.0
- duck 2.0

Merge results in invalid dependencies
Reset composer.lock

git checkout <refspec> -- composer.lock

git checkout master -- composer.lock

DNA-upgrade

composer.lock
- zebra 1.1
- giraffe 1.0
- duck 1.0
Apply the update again

composer update giraffe
   --with-dependencies
Resolving composer.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
Publishing packages

- **composer validate**
  - Will inform you about problems like missing fields and warn about problematic choices like unbound version constraints

- Do not publish multiple packages under the same name, e.g. CE/EE
  - **Names must be unique**
Continuous Integration for Packages

- Multiple runs
  - `composer install` from lock file
  - `composer update` for latest deps
  - `composer update --prefer-lowest --prefer-stable` for oldest (stable) deps

- Potentially multiple composer.json files with different platform configurations
  - `COMPOSER=composer-customer1.json` `php composer.phar update`
  - `COMPOSER=composer-customer1.json` `php composer.phar install`
  - Takes away benefit of “composer install” just working on any PHP project, so avoid this except for testing
Development Tools

- **require-dev** in composer.json
  - These packages won’t be installed if you run `composer install --no-dev`
  - Use for testing tools, code analysis tools, etc.

- **--prefer-source**
  - Clone repositories instead of downloading and extracting zip files
  - Default behavior for dev versions
  - Allows you to push changes back into dependency repos
Deployment

Best Practices
What properties should deployment have?

- Unreliable or slow deployment process
  - You will be scared to deploy
  - You will not enjoy deploying
- Consequence: You will not deploy often
  - Infrequent deploys increase risks
    - You will not be able to spot problems as quickly
    - Problems will fester over time
- Vicious Cycle
  - Reliability and speed are key to breaking it
Composer install performance

- `--prefer-dist`
  - Will always download zip files over cloning repositories

- Store `~/.composer/cache/` between builds
  - How depends on CI product/setup you use
Autoloader Optimization

- composer install --optimize-autoloader
  - composer dump-autoload --optimize

- composer install --optimize-autoloader --classmap-authoritative
  - composer dump-autoload --optimize --classmap-authoritative

- composer install --optimize-autoloader --apcu-autoloader
  - composer dump-autoload --optimize --apcu

https://getcomposer.org/doc/articles/autoloader-optimization.md
Reduce dependence on external services

- **Build process** *(move more into this)*
  - Install dependencies (Composer, npm, ...)
  - Generate assets (Javascript, CSS, ...)
  - Create an artifact with everything in it

- **Deployment process** *(make this as small as possible)*
  - Move the artifact to your production machine
    - sftp, rsync, apt-get install, ...
  - Machine dependent configuration
  - Database modifications
  - Start using new version
Never Deploy Without composer.lock
Reduce dependence on external services

- Composer install loads packages from URLs in composer.lock
  - Packagist.org is metadata only
  - *Open-source dependencies could come from anywhere*

- Solutions to unavailability
  - Composer cache in `~/.composer/cache`
    - Unreliable, not intended for this use
  - Fork every dependency
    - huge maintenance burden
  - Your own Composer repository mirroring all packages
    - e.g. Private Packagist
Summary

Development

- Make a checklist for new dependencies
- composer create-project
- SemVer: Don’t be afraid to increase the major version
- Formalize BC promises for users of your libraries
- composer update [--dry-run] <package>
- git checkout <branch> -- composer.lock
  - replay composer update
- Document changes to dependencies

Deployment

- composer install --prefer-dist --optimize-autoloader --no-dev
- Use a highly available Composer repository (Private Packagist)
- Deploy more frequently
- Focus on reliability and speed of your deployment process
- Deploying should not be scary

Nils Adermann - @naderman - n.adermann@packagist.com
When Deployment goes wrong

- Your site may go down
- You lose orders
- You lose customers
- Customer support has more work
- Developers stressed to get site back up and running
  - More likely to make further mistakes
Typical Deployment Problems

- Manual Error
- Bugs in deployment scripts result in partial deploys
- Inconsistent state across multiple servers
- External services used in the process fail or timeout
  - Required dependencies unavailable for download
- Site unavailable or showing errors during deployment process
Improving your Deployment Process

- **Iterative Improvements**
  - Don’t have to happen in the presented order

- Documenting the current process

- Start automating individual steps

- Change your attitude
  - **Deploy more often**
    - even though it’s scary, it will make deployment less scary
    - to really feel what the pain points are
  - Management buy-in required, this will hurt at first
Improving your Deployment Process

- Continuous Integration
  - Yes PHP projects have a build process

- Staging Environment
  - As close to real production system as possible

- Full Automation
  - Configuration Management

- Continuous Deployment
No-Downtime Database Migrations

- **Adding** database schema element
  1. Add schema element
  2. Update code to fill and then use the new column/table/index/…

- **Removing** database schema element
  1. Update code to stop accessing/using the column/table/index/…
  2. Remove schema element
No-Downtime Database Migrations

- Deployment order (covers adding elements)
  1. Migrate Database Schema
  2. Switch Servers to use new code

- Removing an element requires deploying twice
  1. Deploy without database change
  2. Deploy only the database change with unmodified code

- Migration must keep database operational
Deploying with Symlinks

- /var/www/current -> /var/www/20180321
  /var/www/20180310
  /var/www/20180321
  /var/www/20180418

- ln -sfT /var/www/20180418 /var/www/current

- Problems
  - APC/Opcache do not notice change
    - file is still at /var/www/current/index.php
  - Requests which are executed while the link changes
    - Some code from old version, some from new version
Deploying with Symlinks

- Solutions
  - Restarting fpm on deploy
    - Causes downtime
  - cachetool to clear apc/opcache
    - [https://github.com/gordalina/cachetool](https://github.com/gordalina/cachetool)
  - Nginx: change $document_root to $realpath_root
    - Resolves symlink before passing path to PHP
      => No risk of requests using partial code from new & old versions
  - Apache: [https://github.com/etsy/mod_realdoc](https://github.com/etsy/mod_realdoc)

- Read [https://codeascraft.com/2013/07/01/atomic-deploys-at-etsy/](https://codeascraft.com/2013/07/01/atomic-deploys-at-etsy/)
  (by Rasmus Lerdorf)
Blue-Green Deployments

- Two identical sets of production machines: BLUE & GREEN
- Load balancer sends traffic to one system (BLUE)

- Deployment process
  - Set everything up on unused machines (GREEN)
  - Test functionality on GREEN system
  - Switch all traffic from load balancer to GREEN system
  - BLUE system is now idle, can be used for next deploy
Blue-Green Deployments

Illustration by Martin Fowler [https://martinfowler.com/bliki/BlueGreenDeployment.html]
Blue-Green Deployments

- Advantages
  - No risk of stale cache contents
  - None of the symlink issues
  - Deployment won’t impact live production system
  - Easy rollback (just point the load balancer back)

- Downsides
  - Double the hardware requirements
  - Long running processes may be running on non-live hardware
  - Doesn’t simplify database migrations
Use a PaaS (Platform as a Service) / Cloud provider which handles this for you