www.phparch.com

October 2017
VOLUME 16 - Issue 10

php[architect]

Composing Software

ALSO INSIDE

Learning Machine Learning, Part Two: Building the Model

Education Station: Doctrine Introduction, Part Two

Artisanal: Queueing with Laravel

The Dev Lead Trenches: Project Management Toolbox

Community Corner: The Imminent Release of PHP 7.2

Security Corner: Data Security Lessons from Equifax

finally{}: On Having Unique Ideas

Uncommon Ab(Uses) of Composer

Managing Private Dependencies

Free Sample Article

Building Software that Lasts

PHP[WORLD] 2017

Maude Lemaire
Slack Technologies

Kerri Miller
Nird LLC

Jessica Quinn

Laura Thomson
Mozilla

Andy Ihnatko
Chicago Sun–Times

Meet our Keynotes!

November 15-16, 2017
Washington, D.C.

world.phparch.com

Sponsored by:

RIPSTECH

AUTOMATTIC

nexmo®
The Vonage® API Platform

twilio

php[architect]

CONTENTS

OCTOBER 2017
Volume 16 - Issue 10

Features

Columns

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

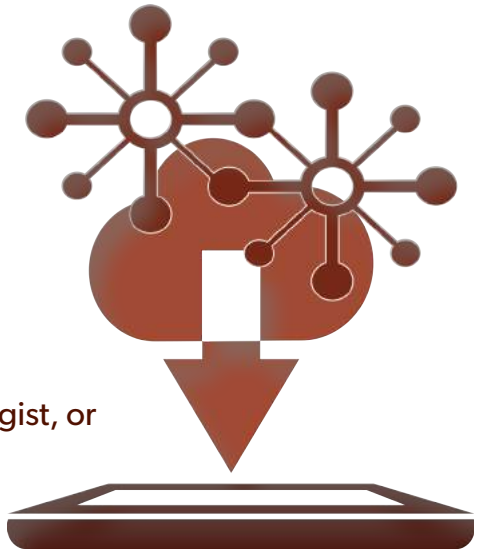
php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Managing Private Dependencies

Andrew Cassell



Using Composer and a private package server is the most efficient way to manage private dependencies with PHP. I recommend using the software as a service, Private Packagist, or Satis, the open source, self-hosted package server. In this article, I'll show you how to easily set this up for your own projects.

Writing software with reusable components is one of the first lessons we learn as programmers. "Don't repeat yourself… Don't repeat yourself… Don't repeat yourself…" is the sacred mantra of all enlightened software developers. Practicing the DRY principle and reusing code are necessary to making software which can be easily modified and maintained.

As good developers, we lean on open source frameworks and libraries to handle the parts of our applications that are not custom business logic. We call these reusable components "dependencies" because our applications depend on them to function. Leveraging frameworks and libraries as loosely coupled dependencies will lead to better application security, better architecture, and less code to write, test, and maintain.

The recent PHP renaissance brought about by the efforts of the people behind Composer[1],

Packagist[2], and the PHP Framework Interop Group—PHP-FIG[3]

—has made development with open source reusable components much easier. If you still live in the "dark ages" and don't use Composer and Packagist, start doing so immediately. See the documentation for more information about installing and setting up Composer.

If a development team is working on multiple applications or decides to break a large monolithic application into smaller microservices, there arises a desire to share common code across separate codebases. You have the choice to use either "copy-and-paste programming" or to find a way to distribute code as a package in a smarter, more efficient manner.

Your packages should be a single class or a collection of classes which serve a sole purpose. It should be used in a minimum of two different codebases to be worth extracting into its own library. The package must also be testable in isolation from the application. However, it is perfectly fine if your private packages rely on other open-source or private dependencies.

1 *Composer:* https://getcomposer.org

2 *Packagist:* https://packagist.org

3 *PHP-FIG:* https://php-fig.org

The next step is to create a new repository for the package and start building. You will want to make sure it is a private repository just like your application. You can accomplish everything in this article using Git, Apache Subversion, GitHub, GitLab, Bitbucket, or your own hosted repository server. This article, however, will only use Git and GitHub to simplify the examples.

You must then create a `composer.json` file in the root of the package's repository. Composer will look for this `composer.json` file to load your package just like it does with open source packages listed on packagist.org. This `composer.json` file is very much like the one in your application or what you see in every open source project which can be installed with Composer.

The "name" is required, and Composer will use it to load the package. This should be in the format of `name-of-your-organization/package-name`. The `autoload` field should be filled out with the namespace of your package `NameOfYourOrganization\\PackageName\\` so Composer can use it to build the autoloader. In the example below, I chose to

Listing 1

```
{
  "name": "organization-name/example-package",
  "description": "Example Package",
  "require": {
   "symfony/console": "^3.0",
  },
  "require-dev": {
    "phpunit/phpunit": "^6.3"
  },
  "autoload": {
    "psr-4": {
      "OrganizationName\\PackageName\\": "src/"
    }
  }
}
```

place the code in an `src` folder, but it's a matter of personal preference and can be left blank if you place the code in the root of the folder (like many Symfony packages).

The rest of the `composer.json` file is optional and will depend on the individual requirements of your package. If your package requires other dependencies, you can list them in the `require` or `require-dev`. In the example below, this package relies on the open source Symfony Console and PHPUnit for testing.

Figure 1

Listing 1 shows package `composer.json`:

Before you pull your package into your application, you need to version it. If you are still heavily developing the package alongside the application, then this can wait. We can use the "master" branch of the library in our application's `composer.json`. But once you have the package stable and ready for deployment, you should tag a release.

Semantic versioning is one of the most important concepts in developing your own packages. As semantic versioning is the most efficient way to express the intent of your updates to your fellow developers, your releases should always use it, and Composer is built with it in mind.

When applications using your library have to be updated because of a change in the library, this is known as a "breaking change." In semantic versioning, we always increment the "major" version number for a breaking change. When we add features that do not require anyone to change their usage of the library, we change the "minor" version number. When we make bug fixes which don't cause breaking changes, we increment the "patch" version number. Using notations like `"^3.4"` in the `composer.json` file is recommended for our application should continue to function with no changes using versions "3.4.0", "3.4.9", "3.5.19", "3.71.0", or "3.99999.99999".

Composer will automatically pull in the latest compatible version. Don't be afraid to increment the major version. It's a warning sign to your fellow developers to watch out for breaking changes.

There are two ways to bring your packages into your application. The first and simplest way is to have Composer pull the package directly from the repository using Git. However, this will only work if your package has no other dependencies, as Composer will not recursively follow and pull in other dependencies. Composer requires a package server to pull in dependencies of your package automatically. If you are trying to keep things simple, this can be an excellent choice until more complex needs arise.

As a reminder, you must give all the developers and servers read access to the repository if you decide to pull directly from the repository. It is worth noting Composer's documentation recommends you do not use this manner of bringing in dependencies, and instead use a package server like Private Packagist. Also, it will slow your deployments down quite a bit, cloning the repositories instead of downloading the code as a compressed file.

> *Semantic versioning is one of the most important concepts in developing your own packages.*

To add a private repository, you need to let Composer know about the repository by adding `repositories` to the JSON as shown in the example below. In the following example, we are pulling in our "example package" directly from the repository.

Example application `composer.json` is

Listing 2

```
{
  "name": "Example Application",
  "require": {
    "symfony/symfony": "\^3.2",
    "aws/aws-sdk-php": "\^3.19",
    "organization-name/example-package": "\^1.0"
  },
  "require-dev": {
    "phpunit/phpunit": "\^6.3"
  },
  "repositories": [
    {
      "type": "vcs",
      "url": "git@github.com:/organization-name/example-package.git"
    },
  ],
  "autoload": {
    "psr-4": {
      "Organization\\Application\\": "src/"
    }
  }
}
```

shown in Listing 2.

The other option for pulling in your private dependencies is to use a package server like Private Packagist or Satis. They will download the code from the Git repositories and create compressed downloads of the packages for your application to use. Once you point them at the repositories of your private packages, everything works just as easy as the open source packages from packagist.org. They will not be in cloned repositories in your vendor folder but will be downloaded files and folders. Other helpful features include mirroring GitHub downloads to speed up deployments and prevent failure of your deployments by hitting GitHub rate limits, a user interface to view the available versions, and an easier way to manage access permissions.

Private Packagist[4] will host a package server which only your organization can access for a monthly hosting fee. Private Packagist is a great resource if you want to free up more time for developing software, rather than managing and securing servers. It requires very little setup effort after giving it access to your organization's GitHub account, as it automatically looks for `composer.json` files in all of your private repositories.

Satis is a free, open source tool you install, run, and manage on your own. Setup is more complex than Private Packagist but is still rather straightforward. Installation documentation is available in the repository[5]. With Satis, you have to manage a JSON file containing a list of all of your package repositories. The Satis server will also need read access to all of those repositories. Configuration is also required to make Satis automatically resolve and make available for download all of the dependencies for your projects. When you update your packages, it is necessary to refresh its information via a cron of the rebuild script, use Git hooks, use GitHub webhooks, or a manual rebuild. And you must set up a method for authentication between your application servers, development machines, and Satis server.

Satis will generate downloadable files for all of your private package repositories, so it is important to secure the Satis server from unauthorized users. To use Satis securely, you should use it over SSH or use the HTTPS version with a free SSL certificate from Let's

Listing 3

```
{
  "name": "Organization Package Server",
  "homepage": "https://satis.example.org",
  "repositories": [
    {
      "type": "vcs",
      "url":
"https://github.com/organization-name/example-package.git"
    },
    {
      "type": "vcs",
      "url": "https://github.com/organization-name/package-two.git"
    },
    {
      "type": "vcs",
      "url": " user@gitserver.example.org:/git/package-three.git"
    },
  ],
  "require-all": true,
  "archive": {
    "directory": "dist",
    "format": "tar",
    "prefix-url": "https://cdn.example.org/S3cr3tH4shF0ld3R",
    "skip-dev": true
  }
}
```

Listing 4

```
{
  "name": "Example Application",
  "require": {
    "symfony/symfony": "^3.2",
    "aws/aws-sdk-php": "^3.19",
    "organization-name/example-package": "^1.0"
  },
  "require-dev": {
    "phpunit/phpunit": "^6.3"
  },
  "repositories": [
    {
      "type": "composer",
      "url": "https://satis.example.org/"
    }
  ],
  "autoload": {
    "psr-4": {
      "Organization\\Application\\": "src/"
    }
  }
}
```


4 *Private Packagist:* https://packagist.com

5 *repository:* http://phpa.me/composer-satis

Encrypt[6] while using an HTTP Header field for token authentication. If possible, you should use a firewall or VPN to make the Satis server accessible to only your developers and servers or host your downloads on a secure Amazon AWS S3 bucket or similar CDN. It is also possible to generate the Satis files and downloads to a hashed folder to provide a little bit of security through obscurity.

Refer to Listing 3 for an example `satis.json` configuration file:

You will then have to update your application's `composer.json` file to point to the new Private Packagist or Satis server. Satis is used in the example below.

Example application `composer.json` pointing at Satis server (see Listing 4).Once you have Private Packagist or Satis working for you, you will eventually need to make changes to your package. The best way to do this is to switch the package to a branch instead of using the versioned release tag and work on the package as a cloned repository while it resides inside the application. When you are done making changes, switch back to a versioned release. To accomplish this, use the following steps:

1. Edit your application's `composer.json` and switch from a semantically versioned release (e.g., `"^1.0"`) to `"dev-master"` and run `composer update`. This will cause Composer to switch to a cloned copy of the package's repository.
2. Make your changes to the package and test its operation within your application. You should also run and update any unit tests you have in the package.
3. After your tests have passed, you can now version the package. You will have to change your working directory to make it the path of your package to commit, tag a release, and push changes for the package.
4. If you are using Satis, you must now rebuild. Remember, it is possible to automate the rebuild using cron, Git hooks, or GitHub webhooks.
5. Update your application's `composer.json` file and point it to your newly released version (e.g., `"^2.0"`).
6. Run `composer update` and Composer will switch back from a cloned branch to a downloaded package.
7. Commit both the application's `composer.json` and `composer.lock` files to your application's repository.

Last, I want to thank everyone who has contributed to Composer, Packagist, and Satis. Jordi Bogianno, Nils Adermann, Rob Bast, and many others who have improved the PHP community and coding practices more than many of us could ever hope to do.

6 *Let's Encrypt:* https://letsencrypt.org

Andrew Cassell is a full-stack web application developer and designer in Herndon, Virginia. Andrew is an employee of the non-profit Marine Spill Response Corporation, the largest dedicated oil spill and emergency response organization in the United States. He works on their website and internal web applications. @alc277

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe