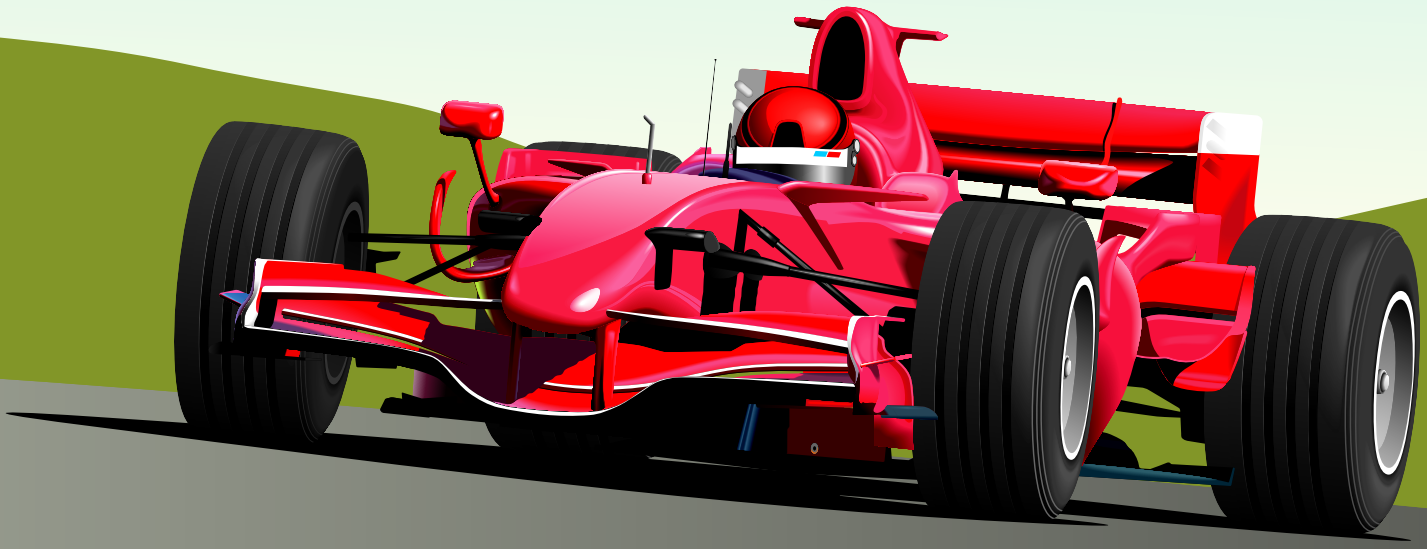
Full Speed Ahead



Community Corner:
Gratitude

Security Corner:
Keeping Credentials Safe

Education Station:
Directing Requests with FastRoute

Leveling Up:
You Had One Job

finally{}:
Hindsight & Planning

WE'RE HIRING

QA

SOFTWARE QUALITY ASSURANCE ENGINEER

test
develop
collaborate
improve

self starter
security minded
experienced
attention to detail

APPLY	nexc.es/218WJ0Y

NEXCESS™
beyond hosting.

NEXCESS.NET | USA 1.866.639.2377 | UK 0.808.120.7609 | AU 1.800.765.472

php[architect]

CONTENTS

MAY 2016
Volume 15 - Issue 5

Full Speed Ahead

Columns

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, Eli White

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

Full Speed Ahead

A constant pressure in our line of work is the need to do more and more with less and less. We've all been there, under pressure to deliver more code sooner with an ever present temptation to cut corners (I mean, who *really* needs tests?)

If your in an agency, projects have deadlines and budgets to meet and often times you're juggling more than one project at a time. If your company is developing a product, you have to balance adding new features with maintaining your codebase. Open-source project maintainers, from the smallest library to the largest framework, can't just churn out *new* features but also have to triage bug reports, update documentation, and promote your project to other developers.

Once you have a few projects under your belt—and it really doesn't take that many—you'll quickly realize that churning out code is not a viable solution. Besides accumulating technical depth, you're probably on the road to burn out if you're not careful. Not every new service, framework, task tracker, or management fad will pan out. More importantly, there's not enough time to learn them all. I urge you to cultivate a healthy sense of skepticism when the next shiny thing comes around.

In this issue, we'll take a look at some things that can save you time and effort in your coding endeavors. First, Jason McCreary shares his experiences in *Building Laravel Shift*, both technical and otherwise. Laravel Shift is his service than can save you time by automating upgrades to new versions of the Laravel framework. In *Mastering OAuth 2.0*, Ben Ramsey shows how to use the league/oauth2-client library to connect to Instagram. OAuth is now a de facto standard for connecting your application to web services, and this article is a step-by-step example explaining how it all works. Marco Pivetta gives us *An Introduction to Doctrine ORM Best Practices* packed with excellent advice on how to structure your Entity and related classes. You're sure to find a number of approaches you can adapt to your own persistence layer to make it easier to understand, less error-prone, and simpler to maintain.

Also this month, Edward Barnard starts a three-part series with *Learn from the Enemy: Securing Your Web Services, Part One*. In this part, he'll show you why your web site and web service should be treated differently when talking about security. In *Education Station*, Matthew Setter writes about *Directing Requests with FastRoute*. FastRoute is a library which brings fast performance to handling request routing. In *Leveling Up: You Had One Job*, David Stockton explores what it means to be a senior developers. He shares how he defines different roles and the ensuing expectations for each at his workplace. In this month's *Community Corner*, Cal Evans rants about the lack of gratitude that is all-to-common sometimes. Chris Cornutt provides advice on *Keeping Credentials Safe* in *Security Corner*. Your application has to track at least one (if not multiple) credentials for local and external services, are you protecting them adequately?

To round out the issue, Eli White shares his thoughts on *Hindsight & Planning* in *finally{}*. When we look back at our own code, or even at another project, it's easy to pick the decisions made apart but one should first understand why they were made at the time.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

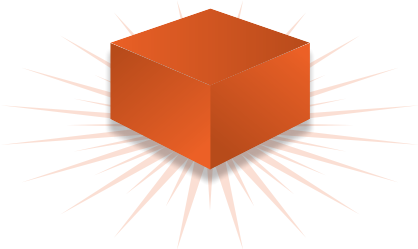
Stay in Touch

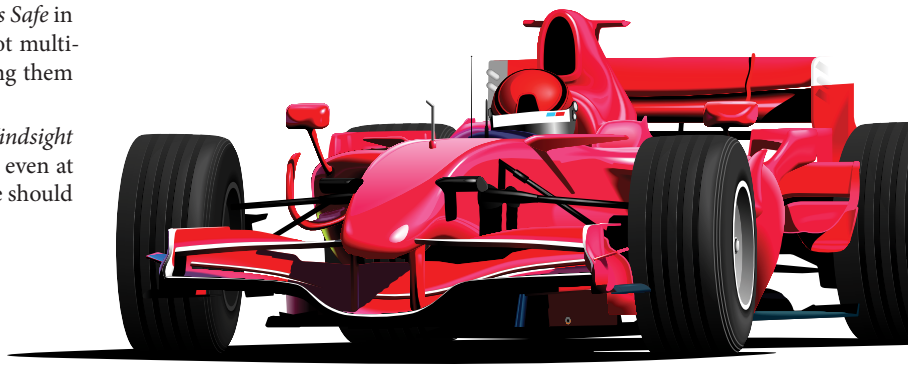
Don't miss out on conference, book, and special announcments. Make sure you're connected with us via email, twitter, and facebook.

- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this issue's code package:

http://phpa.me/May2016_code





Building Laravel Shift

Jason McCreary

Laravel is one of the popular PHP frameworks of our time. And much like other projects in the PHP community, it has adopted a more rapid release cycle. This can make it difficult for developers to stay current. Wouldn't it be great if there was a tool that automatically upgraded your Laravel application for you? I thought so, too. Which is why I built *Laravel Shift*—the automated way to upgrade Laravel applications.

Like all software, the Laravel framework is upgraded over time. It's easy to upgrade the framework. Just update the dependency and run `composer install`. But what about your application code? You have to go through the tedious exercise of reviewing the docs, testing your search-and-replace skills, and moving files. I'll be honest, I don't want to do that work. That's what a computer is for. That's what Shift is for.

I want to share the story of building Laravel Shift. To do so, I've broken this article into three parts. First, I want to explain not only how this project came about, but its goal. Let me be clear—this is not a sales pitch. Rather, it's the process of turning a project into a product. Second, I want to dig into the main technical challenges and how I solved them. Third, I'll close with the future plans for Shift.

Let's Begin at the Beginning

This past fall, I gave a presentation at php[world][1] titled *All Aboard for Laravel 5.1*[2]. While researching the topic, I was surprised at the lack of resources for upgrading Laravel. There were a few blog posts and, of course, the official *Laravel Upgrade Guide*[3], but I didn't find any tools. I thought, "I should build one for Laravel." Fortuitously, Taylor Otwell (the creator of Laravel) was at the conference. I asked him if he was aware of anything. He said, "No. But I'd use that…" So during the Hack Night, I started *Laravel Shift*.

The Laravel Community

I believe something that has helped popularize Laravel is its vibrant community. Specifically, its services like Forge[4], Envoyer[5], and Laracasts[6]. Interestingly enough, most of these are paid services. People may disagree with mixing paid services and open-source projects. I don't think *open source* automatically means *free* (as in beer). But let's not get into a philosophical debate. Quite simply, I believe these services provide value. I felt Shift could be one of these services. It provides value by saving you time, and time is money.

Breaking Ground

Before getting started, I thought about other upgrade tools I had used in the past. I remember years ago CakePHP offered a shell script which upgraded projects between major versions. Magento does something similar with their security patches. So while this was an ambitious project, I knew it could be done. But how did I want to do it?

One of the hardest things in programming is just getting started. Much like an writer, we get writer's block. Sometimes you just have to start typing. Even if it's bad, just start writing code. This is exactly what I did. I started writing shell scripts, PHP snippets, even a custom *sniff* for PHP Code Sniffer. It got pretty complex pretty quick. So I went back to Taylor and he suggested starting with the upgrade from 5.0 to 5.1, as it had fewer changes. When he said that, it reminded me of the *lean process*.

Go Lean or Go Home

It is interesting because I work on an extreme programming (XP) team. We're all about lean process. But in the excitement of creating Shift, I strayed from the path. So I backtracked. I took Taylor's advice. After all, one Shift was the minimum viable product (MVP). Furthermore, from a timing perspective, Laravel 4.2 was over a year old—maybe developers had already upgraded. I felt the Laravel 5.1 Shift was also more relevant. Especially with Laravel 5.2 scheduled to be released soon. So I planned to make the 5.1 Shift, then the 5.2 Shift, then, if the market existed, go back and make the 5.0 Shift.

This iterative and measured approach has become the foundation for developing Shift. It only took a few days to finish the 5.1 Shift. Around the same time, I also shared a post on reddit to gauge interest. In doing so, nearly all the feedback expressed interest in upgrading from 4.2 to 5.0. So I pivoted and built the 5.0 Shift. With the foundation from the 5.1 Shift, it only took about a week. Once I had those two Shifts, I decided to build the site.

From inception to launch, Laravel Shift only took about a month. I was able to time its release with the release of Laravel 5.2. And as such, wrote the 5.2 Shift. It really couldn't have worked out better. I created a service that could upgrade a Laravel application from 4.2 all the way to the latest version of Laravel. I attribute that speed to the lean process.

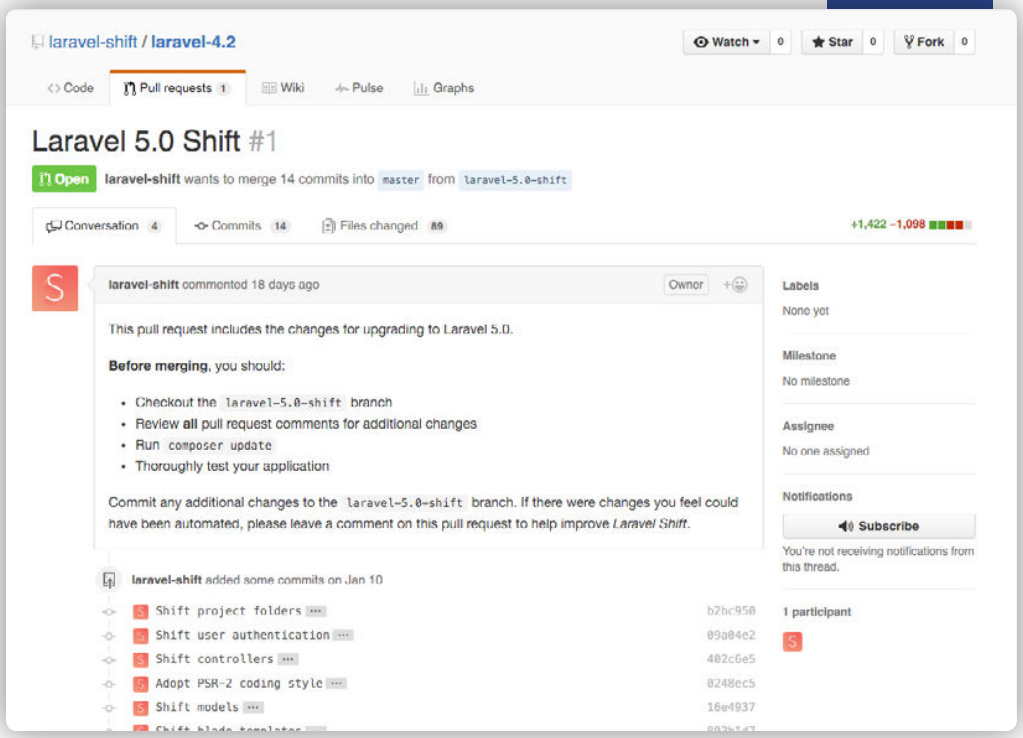
From a Project to a Product

I wandered off the lean path for a few weeks after the launch. I was excited. Every Shift that ran, I checked the logs. I was making adjustments every night. It was the Wild West. I was going to either burn out or smother the project. So I took a step back. Now that Shift had launched, it needed to be treated like a product. I couldn't just change whatever I wanted. I needed to pay attention to usage. I still wanted to actively develop, but I started categorizing work strictly between critical bugs and enhancements. I continued to fix critical bugs immediately. But anything else became part of the next *release*. I set milestones for each release. At 100 Shifts I would release

1 *php[world]:* https://world.phparch.com
2 *All Aboard for Laravel 5.1:* http://phpa.me/all-aboard-L51
3 *Laravel Upgrade Guide:* https://laravel.com/docs/master/upgrade
4 *Forge:* https://forge.laravel.com
5 *Envoyer:* https://envoyer.io
6 *Laracasts:* https://laracasts.com

the next set of enhancements. The next milestone was 250 Shifts, then 500, then 1,000. Doing this has provided structure and forces me to consider decisions carefully. No need to get fancy. In my case, a simple Todo markdown file was enough.

FIGURE 1



Product Overview

Before getting into the technical specification, allow me to talk you through Laravel Shift. To start, you sign in with your Bitbucket, GitHub, or GitLab account to allow access to your repository. Then you place your order by choosing a Shift and providing a few details. Once payment is received, the *Laravel Shift* is added as a collaborator to your repository and your Shift is started. When your Shift is finished, a Pull Request (PR) with detailed comments is opened for you to review the changes.

It's important to note that the entire process is automated. I didn't want users to have to bounce back and forth between Laravel Shift and other services. You sign in, choose your Shift, and you get a PR. You can upgrade your Laravel application in minutes with just few clicks, all from http://laravelshift.com.

Technical Specifications

Now that you have a product overview, let's get into the technical details. Any project I work on I try to find the right tool for the job. This can be tough. I don't pressure myself to find it immediately. Instead, I keep an open mind and allow myself to pivot if a better solution presents itself. Laravel Shift is a mashup of technologies broken into two main parts: the website and Shift. The website handles authentication, checkout, and payment. Shift handles cloning the project, upgrading the code, and submitting a pull request.

The Website

I wanted to build the site in Laravel. I first tried to see if *Laravel Spark*[7] was an option. I remembered Taylor demoing it at Laracon last August. For those not familiar, Laravel Spark is essentially a SaaS platform for Laravel applications. Unfortunately, it was still in alpha. So I stuck with Laravel. After all, it has all the base components Laravel Spark uses. To authenticate with GitHub and Bitbucket, I used Socialite. I started to use Cashier, but realized that was really only for subscriptions. Shifts were one-time purchases. At first, I made the mistake of rebuilding Cashier for one-time purchases. However, I didn't need that. So I stayed disciplined, practicing XP, and called YAGNI (you aren't gonna need it). I ripped it all out and used the Stripe API directly to charge credit cards.

With authentication and payment done, I needed a way to process the Shift. Since Shifts might be long-running, I knew I needed to run them in the background. I thought about using a cron job, but didn't like the needless polling. That is to say, I wanted the process to start immediately and on demand. Laravel provided what I needed again with Job Queues. I created a job that passed the order data to Shift. From there, Laravel handled running the job asynchronously. If the job failed, Laravel attempted to rerun the Job. If it fails a third time, I pop it off the queue and send an email notification to the user. This could have been a very complex piece, and Laravel made it very easy.

Shift

Shift itself is outside of Laravel. Laravel hands off the order for processing by passing several arguments to a PHP script. This PHP script manages the underlying technologies that perform the upgrade and submit the Pull Request. To stick with the lean process, I made myself write code naively. I'd write very basic code until I had a requirement to make it more complex. This has worked out really well.

For example, at first I thought Shift could be a shell script. After all, most upgrades were just moving files and text changes. All of which could be done with basic Unix commands. I did this at first. However, this was pretty low-level and I felt like I was writing a lot of code. So I migrated to PHP. Yet, I didn't abandon the Unix commands. In most cases, they were still the right tool for the job. For example, merging directory structures was a one-liner with `rsync`. No reason to recreate that in PHP. However, manipulating text was more straightforward in PHP—`sed` and `awk` look like line noise after a while. The snippet in Listing 1 shows how I've wrapped Unix commands in a PHP library.

Another example is the Git integration. At first, I only integrated with GitHub and Bitbucket. The code was very basic. I just used inline `if` statements to switch between the different API calls. However, I believe in the *Rull of Three*[8]. So once I added GitLab authentication, I refactored this code into objects sharing a common interface. This allowed me to encapsulate a lot of that code,

7 *Laravel Spark:* https://spark.laravel.com/

8 *Rule of Three:* http://phpa.me/wiki-rule-of-three

and instead call methods like `addComment()` or `openPullRequest()`.

Where possible, I used third-party libraries to integrate with other systems and avoid reinventing the wheel. Some of the packages that Shift depends on include:

- squizlabs/php_codesniffer
- knplabs/github-api
- m4tthumphrey/php-gitlab-api
- composer/semver
- vlucas/phpdotenv

LISTING 1

```
$output = Util::run_system_command('find app -maxdepth 1 -mindepth 1 -type f -print');
if (!empty($output)) {
    $remaining_app_files = preg_split('/\R/', trim($output));
    foreach ($remaining_app_files as $file) {
        Util::run_system_command('mv ' . $file . ' ' . $unshifted_app_path);
    }

    $comment = 'Laravel 5 only has model files in the top-level of the app folder. '
             . 'Your Laravel 4.2 app contained the following files within the app '
             . 'folder. You should move these files into another folder.';
    $commentBag->add(new Comment($comment, $remaining_app_files));
}
```

String Manipulation or the Token Analysis

Figuring out context has been the largest technical challenge. Since PHP is a dynamic language, it's difficult to know things like the type of a variable or how it's being used. Context helps infer such things. For example, if the string `"class"` is at the beginning of the line, it's likely a `class` declaration. Or if a string were followed by an opening parenthesis, it might be a function call.

Two things have helped determine context. First, Shift assumes a PSR-2 code style. This was a safe assumption since Laravel itself adopted PSR-2 in version 5. Based on this assumption, the code format can provide some clues about context. In other cases, I needed to go deeper, and for that I use `token_get_all()`. This gives me the PHP language tokens, which provides even more context. This is still proving to be a challenge. Most of the bugs are related to context issues. I think tokenization is the right approach, but a better way may present itself.

Leveraging Git

Shift also interacts with Git. Git proved to be an excellent foundation for several reasons:

- it allows immediate access to the code
- it serves a backup of the code
- it separates the changes for review
- it provides transparency

This last one was important. As developers, we can be territorial about our code. We don't want others messing with our code. So I didn't want Shift to be this black box that made all these changes to your code. So Shift makes changes incrementally, with full commit messages that reference to the docs so you have a complete history.

An area where I really leveraged Git was applying *patches*. I borrowed this concept from Magento. If you look at their security patches, they're actually shell scripts. This script contains several `patch` commands. Each patch contains instructions for changing code. Its syntax is rich and it took a few blog posts to understand. We've all seen it, though. Run `git diff` the next time you make some changes to a Git repository. That's exactly it. Git tracks changes with patches. As such, Shift sometimes talks to Git directly by applying patch commands. In these cases, I don't have to determine context or run extra commands, I just have a patch file that can be applied directly to the code. For example, Listing 2 shows a patch for upgrading bindings in Laravel 5.1.

LISTING 2

```
--- a/app/Http/Kernel.php
+++ b/app/Http/Kernel.php
@@ -8,25 +8,25 @@ class Kernel extends HttpKernel
      * The application's global HTTP middleware stack.
      * @var array
      */
-    protected $middleware = ['Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
+    protected $middleware = [\Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
         \App\Http\Middleware\EncryptCookies::class,
-        'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
-        'Illuminate\Session\Middleware\StartSession',
-        'Illuminate\View\Middleware\ShareErrorsFromSession',
+        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
+        \Illuminate\Session\Middleware\StartSession::class,
+        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
-        //'App\Http\Middleware\VerifyCsrfToken',
-        'App\Http\Middleware\SetReferralUrl',
+        //\App\Http\Middleware\VerifyCsrfToken::class,
+        \App\Http\Middleware\SetReferralUrl::class,
     ];
```

The Server

I want to speak a bit about the server Shift runs on. People often think you need some complex, redundant collection of segregated instances. In the interest of staying lean, I challenge that thinking. Shift runs on a single small instance. But this requires you to be lean all the way through. The website leverages Laravel route and view caching. Shift, while potentially CPU intensive, only runs for a few seconds. So it works. I think the thing that gives me confidence about this setup is treating instances as disposable. I have a script that can completely rebuild an instance for Shift. So if something were to go wrong, I can just fire up another instance. I'm not a sysadmin. I don't want needless complexity. When there are enough Shifts coming in to require scale, I'll consider that a high-quality problem. Until then, I'm willing to defer that problem to tomorrow.

Good Enough

Shift isn't perfect. It won't upgrade everything for every project. As we discussed, between the dynamic nature of PHP and the varying ways to use Laravel, upgrading everything is just shy of impossible. It boils down to being "good enough." If Shift can complete 90% of the upgrade for 90% of projects, I consider that a great success. This is acceptable for two reasons. First, I want Shift to be opinionated. If you do things the "Laravel way," Shift is more

likely to upgrade your project completely. Second, it's diminishing returns. From a product perspective, I could spend a lot of time to get a little improvement for a few highly customized applications. But I think that time is better spent fixing bugs or developing new Shifts.

Dammit Jim, I'm a Programmer, Not a Marketer!

The biggest challenge with this project has been the social aspect. People are skeptical. It's an ambitious product and I think people doubt it can work. So they'd rather do it themselves. On top of which, people underestimate the amount of work that goes into upgrading. It's easy to think it'll take *15 minutes*. Then hours later you realize it's a big job. So you cut corners and now your project is in some *hybrid state* between versions.

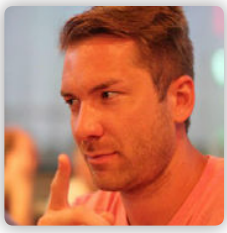
That's really what Shift comes down to for me—time value. Sure, you could do a better job than Shift. You know your code better and you can intuit things Shift can't. But to do that would take hours. Or, Shift can do most of that for you in a few minutes. Now you can spend that saved time refactoring your code to take advantage of new features. That's a far better use of your time. When you value your time, it's easy to see the value in Shift.

The Future of Shift

Shift has moved out of alpha and beta and I recently released the first set of micro-Shifts. The next milestone is 500 Shifts. This will likely include a significant rewrite of the Shift code to resolve some of the persistent bugs. I've also had people Shifting from 4.2 all the way to the latest version. I may rework the payment where you can buy Shifts in bulk at a discount. I also need to write the 5.3 Shift, which will need to be ready for the Laravel 5.3 release sometime in June.

In addition, many of the Laravel Shifts apply to PHP as well. So I'm finalizing PHP Shift[9]. The MVP products will include a PSR-2 Shift to automatically format your PHP code with this popular coding style, as well as a PSR-4 Shift that will upgrade PSR-0 namespaces to PSR-4 namespaces. Shortly after, I plan to release a *MySQL Shift*, which converts the deprecated `mysql_` functions to their `mysqli_` equivalents in an effort to help projects move to newer versions of PHP. All of these will build toward the ultimate goal of upgrade Shifts between PHP versions.

The sky is the limit. If I can solve some of these context issues, I may be able to create an API for building Shifts. Then I could open up a marketplace where developers could create their own Shifts. In the end, Shift's success is tied to its usage. That seems true for any product, but more so for Shift. The more people that use it, the more I can improve it. And the more code that gets upgraded. Which helps keep the Laravel and PHP communities progressing.



Jason McCreary, or JMac, is a PHP & iOS application development. His current focus is working on Laravel Shift, PHP Shift. He loves being a "geek" and interacts with the community through mentoring and running the Louisville Software Engineering Meetup. You should reach out to him on Twitter. @gonedark.

9 *PHP Shift: https://php-shift.com*

Mastering OAuth 2.0

Ben Ramsey

OAuth 2.0 is the de facto standard for authenticating users with third-party websites. If you want access to a user's data in Google or Facebook, for example, OAuth 2.0 is what you use. But, let's face it: OAuth 2.0 is not easy, and to make matters worse, it seems everyone has a slightly different implementation, making interoperability a nightmare. Fortunately, the *PHP League of Extraordinary Packages* has released version 1 of the league/oauth2-client library. Aiming for simplicity and ease-of-use, league/oauth2-client provides a common interface for accessing many OAuth 2.0 providers.

OAuth solves a specific problem: it minimizes exposure to credentials. It achieves this through *authorization grants*. You grant a website access to your account information on another website. The grantee website then uses temporary credentials called *access tokens* to access your information on the grantor website, usually through the grantor's API. In this way, you only use your username and password to authenticate with the service where your data lives and not anywhere else.

Let's Jump In

But that all sounds confusing and wordy. To understand OAuth, it's best to see it in action, and we'll use Instagram for our example since Instagram is an OAuth 2.0 provider.

We want to create a website that pulls a user's photos from Instagram and shows them to the user on our website. To do this, the user has to grant permission to let us request their photos from Instagram, but they don't need to give us their username and password. Those are secret and sacred credentials, known only to them and Instagram. Instead, we'll use OAuth 2.0 to request permission to access their photos, keeping the user's credentials safe.

> *You can view the complete code example from this article at https://github.com/ramsey/oauth2-phparch*

To build a quick example application illustrating OAuth 2.0 concepts, we'll use the Laravel framework[1] with the league/oauth2-client[2] library. However, the league/oauth2-client library may be used with any framework or standalone project.

> ***Note*** *If you have Composer installed, great! If not, go to https://getcomposer.org and read the "Getting Started" section to find out how to install it on your system. Once you have Composer installed, you'll be ready to complete the rest of this exercise.*

Open your terminal application and run the following command to use Composer to create a new Laravel project. This command will create a directory named `oauth2-phparch`.

```
composer create-project --prefer-dist laravel/laravel \
oauth2-phparch
```

In the `oauth2-phparch` directory, we'll want to run a few commands to set up our new application, so use `cd` to change directory to `oauth2-phparch`. Then, set up the Laravel scaffolding for authentication with the following command:

```
php artisan make:auth
```

It's not important right now to understand everything this command does; in a nutshell, it sets up all the routes and views that Laravel needs to make authentication work.

Next, we'll run a few commands to set up a simple database connection using SQLite.

```
sed -i 's/DB_CONNECTION=mysql/DB_CONNECTION=sqlite/' .env
sed -i 's/DB_DATABASE=homestead/DB_DATABASE=database\
/database.sqlite/' .env
touch database/database.sqlite
php artisan migrate
```

The first two commands update the .env file that the Composer `create-project` command created for us. They tell Laravel to use SQLite as the database. The second command creates an empty SQLite database file, and the third command runs Laravel migrations to set up authentication.

> ***Tip*** *If your system doesn't have the* `sed` *or* `touch` *programs, or these commands don't work, you may open the* `.env` *file in your favorite text editor and make these changes manually. You may also use your text editor to create an empty SQLite database at* `database/database.sqlite`.

Now, we're ready to run the built-in PHP web server, and we can do so with the following command:

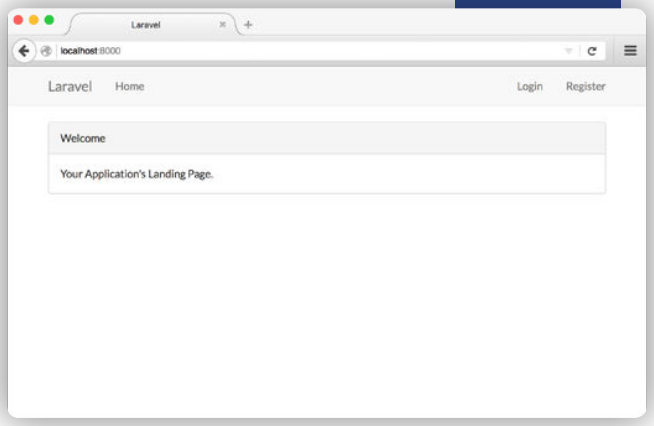
```
php -S localhost:8000 server.php
```

1 *Laravel framework: https://laravel.com*

2 *league/oauth2-client: http://oauth2-client.thephpleague.com*

If all went well, when you go to http://localhost:8000 in your web browser, you should see Figure 1.

Laravel Application Landing Page **FIGURE 1**



Preparing for OAuth

Now, let's start working on our OAuth 2.0 integration with Instagram. I've created a package for Laravel to make this easier. Let's require it with Composer.

```
composer require ramsey/laravel-oauth2-instagram
```

Open `config/app.php` in your favorite editor and modify the `providers` and `aliases` arrays with the values shown in Listing 1.

Our Instagram OAuth 2.0 service provider is now set up for use within Laravel and we can run the `artisan` command to publish it, which copies around some configuration files.

```
php artisan vendor:publish
```

Now it's time to set up two accounts: one in our Laravel application and one on Instagram. With the PHP built-in web server running our application, go to http://localhost:8000/register and register for an account. You may want to familiarize yourself with the Instagram Developer Documentation[3] or have it handy for reference.

Afterwards, go to https://instagram.com to create an Instagram account (if you don't have one) and then to https://instagram.com/developer/clients/register/ to sign up as a developer and register an Instagram client. We'll use these client credentials in our Laravel application. When registering a client, feel free to use any values you wish, but one of the *Valid redirect URIs* must be the value:

```
http://localhost:8000/instagram
```

We'll use this URL in our Laravel application.

After registering an Instagram client, we'll configure our Laravel application with the *client ID* and *client secret* provided by Instagram. Open the .env file in a text editor and add the following values, replacing the X's with the Instagram client values.

```
INSTAGRAM_CLIENT_ID=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
INSTAGRAM_CLIENT_SECRET=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
INSTAGRAM_REDIRECT_URI=http://localhost:8000/instagram
```

3 *Instagram Developer Documentation: https://www.instagram.com/developer*

Add the service provider and facade alias. **LISTING 1**

```
<?php
return [
    /* ... */

    'providers' => [
        /* ... */

        Ramsey\Laravel\OAuth2\Instagram\InstagramServiceProvider::class,
    ],

    'aliases' => [
        /* ... */

        'Instagram' => Ramsey\Laravel\OAuth2\Instagram\Facades\Instagram::class,
    ],
];
```

app/Http/Controllers/HomeController.php **LISTING 2**

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use Illuminate\Http\Request;
use Instagram;

class HomeController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function index(Request $request)
    {
        $instagramUser = null;
        $instagramFeed = null;

        if ($request->session()->has('instagramToken')) {
            $instagramToken = $request->session()->get('instagramToken');

            $instagramUser = Instagram::getResourceOwner($instagramToken);

            $feedRequest = Instagram::getAuthenticatedRequest(
                'GET',
                'https://api.instagram.com/v1/users/self/feed',
                $instagramToken
            );

            $client = new \GuzzleHttp\Client();
            $feedResponse = $client->send($feedRequest);
            $instagramFeed = json_decode($feedResponse->getBody()->getContents());
        }

        $redirectionHandler = function ($url, $provider) use ($request) {
            $request->session()->put(
                'instagramState',
                $provider->getState()
            );

            return $url;
        };

        $authUrl = Instagram::authorize([], $redirectionHandler);

        return view('home', [
            'instagramAuthUrl' => $authUrl,
            'instagramUser' => $instagramUser,
            'instagramFeed' => $instagramFeed,
        ]);
    }
```

LISTING 2 (CONT'D)

```
54.
55.     public function instagram(Request $request)
56.     {
57.         if ($request->session()->has('instagramToken')) {
58.             return redirect()->action('HomeController@index');
59.         }
60.
61.         if (!$request->has('state')
62.             || $request->state !== $request->session()->get('instagramState')
63.         ) {
64.             abort(400, 'Invalid state');
65.         }
66.
67.         if (!$request->has('code')) {
68.             abort(400, 'Authorization code not available');
69.         }
70.
71.         $token = Instagram::getAccessToken('authorization_code', [
72.             'code' => $request->code,
73.         ]);
74.
75.         $request->session()->put('instagramToken', $token);
76.
77.         return redirect()->action('HomeController@index');
78.     }
79.
80.     public function forgetInstagram(Request $request)
81.     {
82.         $request->session()->forget('instagramToken');
83.
84.         return redirect()->action('HomeController@index');
85.     }
86. }
```

Integrating with Instagram

Until now, everything has been preliminary setup, getting us to the point where we can begin integrating with Instagram as an OAuth 2.0 provider. Every OAuth 2.0 provider has similar account setup and configuration steps necessary to provide unique client ID and secret values to any client application wanting to integrate with the provider.

Now that we have our client credentials, we're ready to begin writing the code. Listing 2 shows our completed `HomeController`.

Listing 3 shows our modified home view, but we'll step through each main concept to explain what's going on.

We also need to add the following routes to `app/Http/routes.php` to activate the new routes in `HomeController`:

```
Route::get('/instagram', 'HomeController@instagram');
Route::get('/forget-instagram',
          'HomeController@forgetInstagram');
```

resources/views/home.blade.php **LISTING 3**

```
01. @extends('layouts.app')
02.
03. @section('content')
04. <div class="container spark-screen">
05.     <div class="row">
06.         <div class="col-md-10 col-md-offset-1">
07.             <div class="panel panel-default">
08.                 <div class="panel-heading">Dashboard</div>
09.
10.                 <div class="panel-body">
11.                     You are logged in!
12.
13.                     @if ($instagramUser)
14.
15.                         <h1>Hello, {{{ $instagramUser->getName() }}}</h1>
16.                         <p>{{{ $instagramUser->getDescription() }}}</p>
17.                         <p><a href="/forget-instagram">Forget Instagram token</a></p>
18.
19.                         <h2>Your Instagram Feed</h2>
20.                         @forelse ($instagramFeed->data as $item)
21.                             <div style="float: left; padding: 10px;">
22.                                 <a href="{{{ $item->link }}}">
23.                                     <img src="{{{ $item->images->thumbnail->url }}}">
24.                                 </a>
25.                             </div>
26.                         @empty
27.                         <p>No photos found in feed. Follow some friends in Instagram.</p>
28.                         @endforelse
29.
30.                     @else
31.                         <p>
32.                             <a href="{{{ $instagramAuthUrl }}}">
33.                                 Click here to authorize with Instagram
34.                             </a>
35.                         </p>
36.                     @endif
37.                 </div>
38.             </div>
39.         </div>
40.     </div>
41. </div>
42. @endsection
```

Authorization Request

The first main OAuth 2.0 concept we need to implement is the *authorization request*. We do this by generating an authorization request URL and either redirecting a user to it or asking them to click a link or a button. The league/oauth2-instagram[4] client library helps with the background logic for generating this URL.

```
$authUrl = Instagram::authorize([], $redirectionHandler);
```

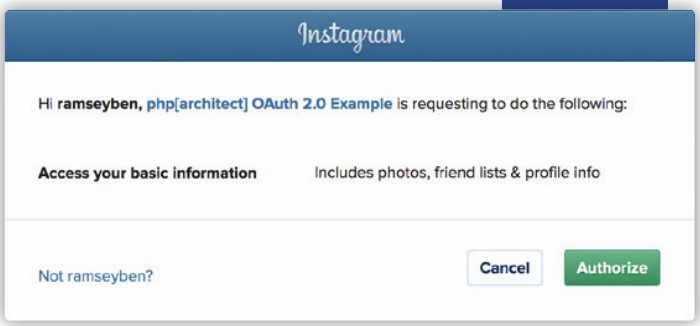
Notice how we pass a `$redirectionHandler` callback to `Instagram::authorize()`. This isn't necessary, but it helps us set the *state* parameter in the user's session. This state value is included in the authorization request URL, and when the provider redirects back to us, we'll check the state parameter they send against the one stored in the session. If they don't match, then we know something has gone wrong, and we shouldn't trust the provider response.

```
$redirectionHandler = function ($url, $provider) use ($request) {
    $request->session()->put(
        'instagramState',
        $provider->getState()
    );

    return $url;
};
```

When the user is redirected or clicks a link to the Instagram authorization request URL, they will see a page similar to that shown in Figure 2.

Instagram Authorization Request **FIGURE 2**



In the example application we've set up, we can see how this works by going to http://localhost:8000/home and clicking the *Click here to authorize with Instagram* link.

Redirection Endpoint

After the user provides their authorization for our client application, Instagram will redirect them to the redirect URL we defined (`http://localhost:8000/instagram`), including as query string parameters an *authorization code* and the same state we sent in the authorization request. Note how we check the state received against the one stored in the session to ensure they match (see the `instagram()` method in Listing 2, lines 61-65).

The next thing the redirection endpoint does is exchange the authorization code for an access token.

```
$token = Instagram::getAccessToken('authorization_code', [
    'code' => $request->code,
]);

$request->session()->put('instagramToken', $token);
```

The `getAccessToken()` method on the league/oauth2-client provider makes a request to Instagram, sending the authorization code and our client credentials. Instagram then sends back an *access token*.

The access token may be stored and reused, so that a new token does not need to be requested each time a user wants to access their data. In this example, we're storing the token to the user's session, but we could also store it in a database or other storage repository.

Expiring & Refreshing Tokens

Most providers include expiration information with the access token, and many include a *refresh token*, as well. The league/oauth2-client library provides functionality for determining whether an access token has expired and for refreshing it.

```
if ($token->hasExpired() && $token->getRefreshToken()) {
    $newToken = $provider->getAccessToken('refresh_token', [
        'refresh_token' => $token->getRefreshToken(),
    ]);
}
```

Instagram does not provide access token expiration information or refresh tokens, so we do not need to use this functionality in our application.

Using Access Tokens

Access tokens are used to request a user's authenticated information without needing their username or password. After Instagram redirects to our redirection URL, our `instagram()` route uses the authorization code to request an access token from Instagram. It stores the access token in the user's session—–again, this could also be a database, etc.—and it redirects the user back to the `index()` route on the `HomeController`.

From the `index()` action, we check whether the token is present in the session. If it is, we retrieve it.

```
$instagramToken = $request->session()->get('instagramToken');
```

It's all downhill from here. The league/oauth2-client library provides a convenience method to request details about the resource owner (the authenticated user).

```
$instagramUser = Instagram::getResourceOwner($instagramToken);
```

The library also provides an easy way to create an authenticated request using the access token. The request returned implements `Psr\Http\Message\RequestInterface`, so it may be used with any PSR-7[5] compliant HTTP client. Here, we use Guzzle[6]; see Listing 4.

LISTING 4

```
01. $feedRequest = Instagram::getAuthenticatedRequest(
02.     'GET',
03.     'https://api.instagram.com/v1/users/self/feed',
04.     $instagramToken
05. );
06.
07. $client = new \GuzzleHttp\Client();
08. $feedResponse = $client->send($feedRequest);
09. $instagramFeed = json_decode(
10.     $feedResponse->getBody()->getContents()
11. );
```

4 *league/oauth2-instagram: https://github.com/thephpleague/oauth2-instagram*

5 *PSR-7: http://www.php-fig.org/psr/psr-7*

6 *Guzzle: http://guzzlephp.org*

The `$instagramFeed` variable contains an object representing the items in the user's Instagram feed. When returning the view (lines 48-52 of Listing 2), we pass this as a view variable, which we use in our view to display the user's feed images, as seen in Figure 3.

```
@forelse ($instagramFeed->data as $item)
    <div style="float: left; padding: 10px;">
        <a href="{{{ $item->link }}}">
          <img src="{{{ $item->images->thumbnail->url }}}">
        </a>
    </div>
@empty
    <p>No photos found in feed. Follow some friends
       in Instagram.</p>
@endforelse
```

At this point, the OAuth 2.0 flow is over, and we're making requests to the Instagram API using the access token and HTTP client of our choice.

This seems like an awful lot of work just to let a user access their Instagram data from our website. OAuth 2.0 can appear cumbersome and complex. It is not a protocol. Instead, it is a *framework* for authorization. One example where this trips up many developers is right here in the Instagram implementation. Instagram does not support the use of the `Authorization` header. When developing the league/oauth2-instagram provider library, we needed to diverge from the generic implementation in league/oauth2-client (which uses the `Authorization` header) and simply add the `access_token` parameter to the query string on each request, as Instagram requires.

So, why all the fuss? Why all the hoops to jump through? Why not just pass the user's username and password as API parameters over HTTPS?

A Brief History of Web Authorization

Long, long ago, in the age of Web 1.0, we didn't worry much about this. Most websites were silos of information, and few shared data across domains. Web 2.0 changed all of this with an explosion of APIs. For the first time, a user could sign in to one website and access their data on other, and we did some pretty cool mash-ups with this data. We generated tag clouds, analyzed social standing among peers, and much more. The only problem was users had to trust us with their usernames and passwords. This was the only way we could access their data through the APIs.

The key word here is *trust*. Users began trusting us with their usernames and passwords, and we facilitated it.

This became known as the *password anti-pattern*[7]. Users learned a bad habit. They learned that by giving up very sensitive information—their username and password—they could see some cool visualizations of their data. Usernames and passwords, while protecting important information, became a currency of sorts, and attackers realized they could take advantage of many easy targets.

Most people do not use secure passwords, and they tend to use the same username and password for all their accounts—social networking, as well as banking. Attackers exploited this by creating compelling mash-ups. They didn't need to spend expensive cycles with brute-force attacks. We willingly gave them our credentials to see cool things!

As the Web matured, it became clear that we needed some form of delegated authorization to protect users, while allowing them to give other websites access to their data. Thus, OAuth was born.

What is OAuth 2.0?

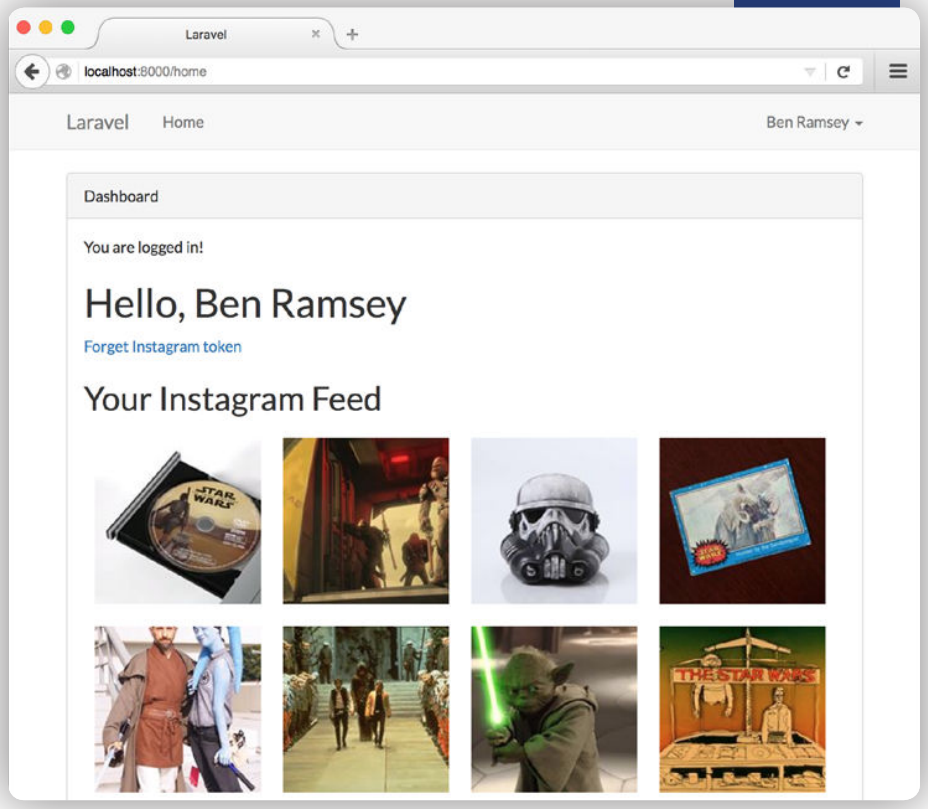
OAuth 2.0 is codified in RFC 6749[8], where it is referred to as an "authorization framework," rather than a protocol. That is, it defines a framework for authorization, along with common conventions, but it intentionally leaves many of the specifics open for interpretation.

Section 1.8 of RFC 6749 states:

> *However, as a rich and highly extensible framework with many optional components, on its own, this specification is likely to produce a wide range of non-interoperable implementations.*

This leads to many implementations, all familiar but slightly varied, and this is the problem that league/oauth2-client aims to solve.

Instagram Feed On Our Website **FIGURE 3**



7 *The Password Anti-Pattern:* https://agentile.com/the-password-anti-pattern

8 *OAuth 2.0 Specification:* https://tools.ietf.org/html/rfc6749

The OAuth 2.0 framework defines four roles and four authorization grant types. Understanding these is crucial to understanding OAuth.

The **resource owner** is the person or entity who is capable of granting permission to a protected resource. Usually, this is a user who has data that a website wants to access. For example, if a website wants to access a user's Instagram photos, the user who can grant access to their photos is the resource owner.

The **resource server** is the server hosting the protected resource(s). This server is able to accept and respond to resource requests using access tokens. In the case of the Instagram example, the Instagram API acts as the resource server.

The **client** refers to any application that uses a resource owner's authorization to make requests of a resource server for protected resources. If a website wants access to an Instagram user's account, that website is the client.

Finally, the **authorization server** is the server that grants access tokens to the client after authenticating with the resource owner. Sometimes, this is the same as the resource server, but it may be separate. Instagram uses the same domain as their API; the URL to access their authorization server is `https://api.instagram.com/oauth/authorize`. A client requesting access to protected resources (i.e., photos in an Instagram account) will redirect the resource owner to this URL, asking them to authenticate with Instagram and grant access.

An authorization grant allows a client to request access to protected resources with the owner's authorization. The four authorization grant types provide flexibility to OAuth implementations, and OAuth allows additional grant types to be specified for future extension.

For many providers, the `GenericProvider` in the league/oauth2-client library is enough to support most OAuth 2.0 grant types. Just specify a handful of configuration parameters:

```
$provider = new \League\OAuth2\Client\Provider\GenericProvider([
    'clientId'=>'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
    'clientSecret'=>'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
    'redirectUri'=>'https://your.example.com/redirect-url',
    'urlAuthorize'=>'https://their.example.net/authorize',
    'urlAccessToken'=>'https://their.example.net/token',
    'urlResourceOwnerDetails'=>'https://their.example.net/api/me'
]);
```

Some providers require additional handling of parameters and requests. In these cases, extend `League\OAuth2\Client\Provider\AbstractProvider` to provide the extra functionality, or take a look at one of the many third-party providers that extend the league/oauth2-client library.

Authorization Code

The **authorization code** grant type is perhaps the most common grant type. We used it in the Instagram example. It's what's commonly referred to as *three-legged* OAuth, since there are three parties involved: resource owner, authorization server, and client. Figure 4 illustrates the authorization code grant flow.

In step 1, the resource owner asks the client to access a protected resource they own. The client then redirects the resource owner to the authorization server (step 2). The resource owner uses their credentials to authenticate and grant access to the client, and the authorization server redirects the resource owner back to the client with an access token (step 3). Now, the client may use the access token to request a protected resource from the resource server (step 4).

Add the service provider and facade alias. LISTING 5

```
$authorizationUrl = $provider->getAuthorizationUrl();

/* Redirection dance */

$accessToken = $provider->getAccessToken('authorization_code', [
    'code' => $_GET['code']
]);

$request = $provider->getAuthenticatedRequest(
    'GET',
    'https://their.example.net/api/',
    $accessToken
);

$client = new \GuzzleHttp\Client();
$response = $client->send($request);
```

We covered the authorization code grant type in depth for the Instagram example, but to recap using the league/oauth2-client library, it works like this:

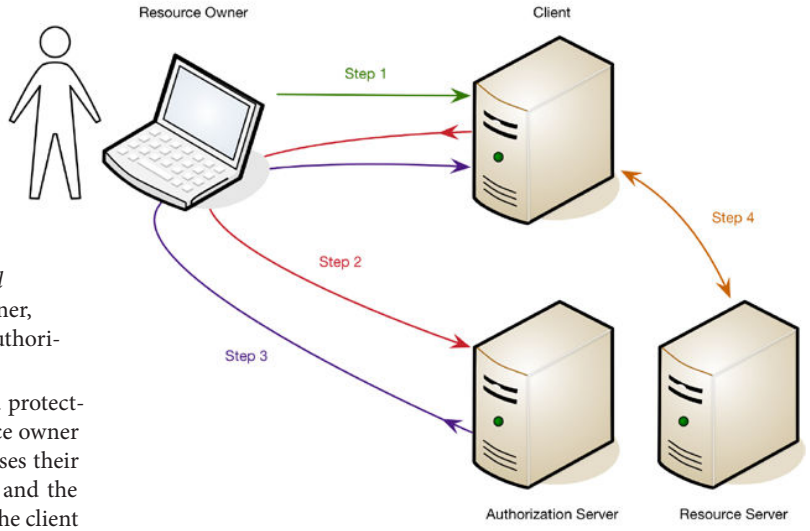
1. Redirect the user to the provider
2. Provider redirects back to our `redirectUri`
3. Exchange the authorization code for an access token
4. Use the access token to make authenticated requests

Resource Owner Password Credentials

The **resource owner password credentials** grant type involves a high degree of trust being placed in the client. In this grant type, the resource owner gives their username and password to the client, and the client sends these to the authorization server once, exchanging them for an access token.

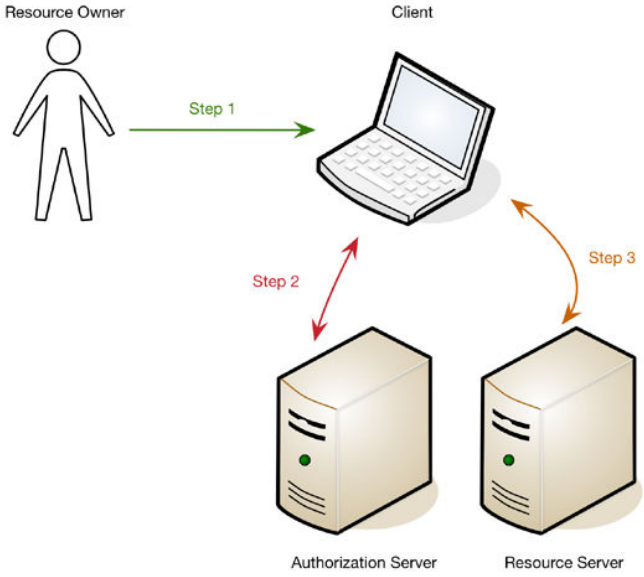
A good example of where the resource owner password credentials grant type is acceptable is when the client is an operating system or a trusted application running on a device owned by the resource owner. **Use this grant type with extreme caution.**

Authorization Code Grant FIGURE 4



As Figure 5 illustrates, the resource owner gives their username and password to the client in step 1. In step 2, the client exchanges the username and password for an access token.

Resource Owner Password Credentials Grant **FIGURE 5**



The access token may then be used in step 3 to request authenticated data from the resource server.

For providers that support this grant type, the league/oauth2-client library makes it possible to use this flow in your PHP applications. Again, do so with extreme care and caution, as this grant type can perpetuate the password anti-pattern.

```
$accessToken = $provider->getAccessToken('password', [
    'username'=>'demouser',
    'password'=>'testpass'
]);
```

Client Credentials

Sometimes the client itself is the resource owner, and the goal is to show users data from a third-party source. This is common, for example, with APIs dealing with weather or map data. The **client credentials** grant type may be used in this case.

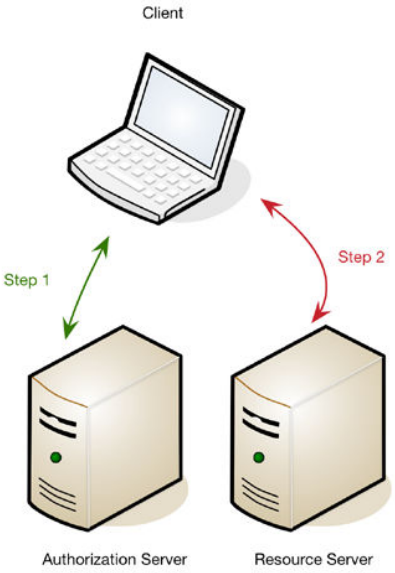
This grant type works very similarly to the resource owner password credentials grant type, but we don't ask the user for any credentials. We are using the client's own credentials, which are stored on the server.

In Figure 6, we see that the user does not provide any credentials. Instead, the client sends its own credentials (client ID and secret) to the authorization server (step 1) in exchange for an access token, which it may then use (step 2) to request authenticated data from the API.

Since the client ID and secret are passed into the provider's constructor in the league/oauth2-client library, the provider object already has the credentials, and we simply exchange them for an access token:

```
$accessToken = $provider->getAccessToken('client_credentials');
```

Client Credentials Grant **FIGURE 6**



Implicit

The final type is the **implicit** grant type. This grant type is optimized for client-side applications. It does not use a client secret, and it takes place wholly within the browser. This means the league/oauth2-client library cannot support this grant type, so we won't cover it in detail.

In short, the implicit grant relies on a known redirection URL for the client identifier. The client redirects to the authorization server, specifying its client ID and redirection URL. The authorization server checks the redirection URL for the client ID against known redirection URLs for that ID. If all checks out, the user authenticates, and the authorization server redirects back to the client with an access token.

Toward a More Secure Web

OAuth 2.0 provides a framework for web authorization. It is a step along our journey to creating a friendlier, more secure Web. While it is difficult to show and explain in concise code snippets, I hope the exercises and examples here have led to a better understanding of the most common OAuth 2.0 flow, the authorization code grant, the reason why OAuth was created, and how to use the league/oauth2-client library.

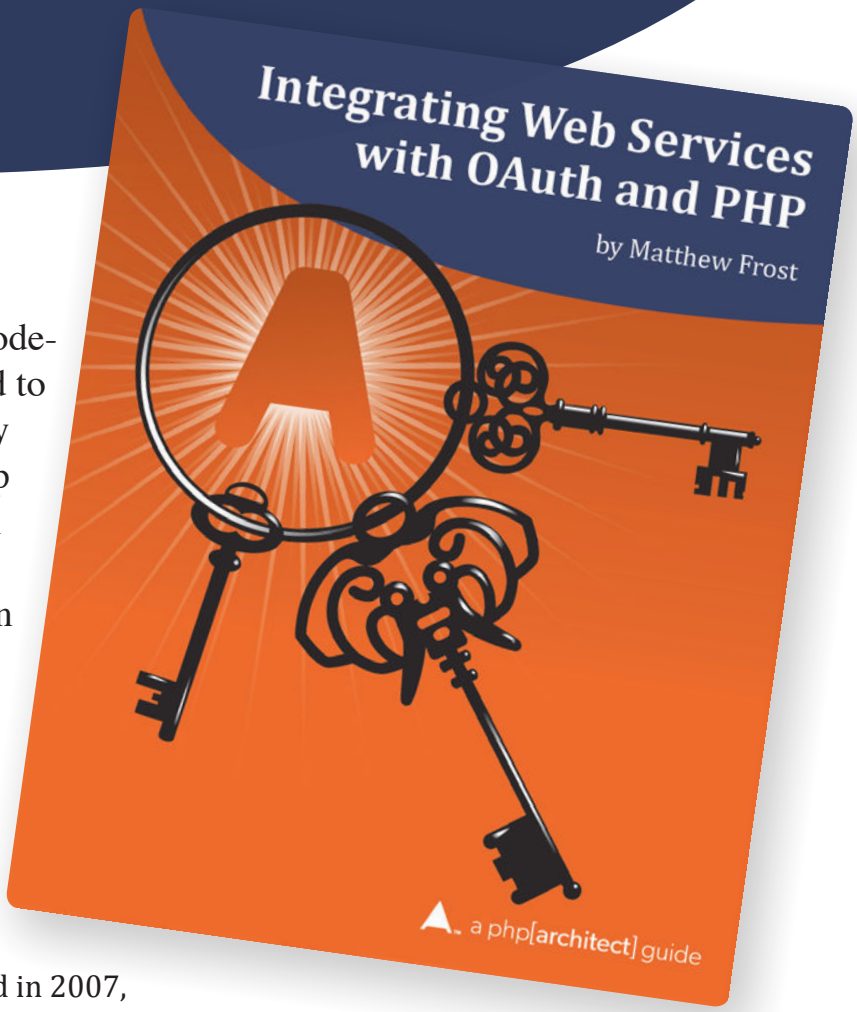
For a deeper dive into OAuth concepts, complete with working examples integrating with popular services like Twitter, Tumblr, and more, check out Matthew Frost's *Integrating Web Services with OAuth and PHP*[9] published by *php[architect]*.

Ben Ramsey is a web craftsman, author, and speaker. He is a software architect at ShootProof, https://www.shootproof.com, where he builds a platform for professional photographers. The maintainer of the league/oauth2-client and ramsey/uuid libraries, he enjoys organizing user groups and contributing to open source software. Ben blogs at benramsey.com and is @ramsey on Twitter.

9 Integrating Web Services with OAuth and PHP: http://phpa.me/oauthbook

Integrating Web Services with OAuth and PHP

by Matthew Frost



Modern web applications are no longer standalone, monolithic codebases. Instead, they are expected to integrate with external, 3rd party applications to allow users to tap into new features, integrate with their social networks, and to easily migrate their data between systems. Many services afford these integrations by building web services that use the OAuth standard to authenticate users and allow "secure delegated access" on their behalf.

There are two versions of OAuth. Version 1.0 as introduced introduced in 2007, and OAuth 2.0 was released in 2012. **Integrating Web Services with OAuth and PHP** describes the differences between the two versions, explains the jargon associated with each, and–most importantly–provides working PHP examples for integrating with popular web services such as Twitter, Tumblr, Instagram, and others. This book also includes a primer on the HTTP protocol, highlights open-source resources for OAuth clients and servers, and discusses issues with OAuth and application security.

Written by PHP professional Matt Frost, this book is an indispensable resource for any PHP developer that builds or integrates with online applications.

Purchase

http://phpa.me/oauthbook

Learn from the Enemy: Securing Your Web Services, Part One

Edward Barnard

Knowing how to secure your web**site** does not translate into knowing how to secure your web **service**. Your website is friendly to humans. You can fend off attacks with CAPTCHA and other ways of detecting and rejecting automated traffic. Your web services, by contrast, are to be consumed by non-humans. If you have a flagship mobile app, it's not a human. It's an app! You therefore need to take a far different approach to securing your web services. I'll show you my experiences and the attitude you need to protect your own.

It Happens

On September 14, 2015, Business Wire announced[1]:

> *Kim Kardashian West, Khloé Kardashian, Kendall Jenner and Kylie Jenner today launched new Personal Media Apps – and websites – allowing them to connect more directly with their fans and provide a unique and personal look into their lives.*

Two days later, on September 16, 2015, TechCrunch published an article *Kardashian Website Security Issue Exposes Names, Email of Over Half a Million Subscribers, Payment Info Safe*[2]. The article describes the discoveries of 19-year-old software developer Alaxic Smith.

Alaxic Smith reported that the Kardashian app had a JavaScript file providing client (app) access to the website API. So long as he was logged into the website himself, he could get the web services to respond with information on the 663,270 people who had signed up for the site. The other sisters' sites behaved identically.

It happens!

Getting the Attitude

Fifteen years ago I wrote a series of articles, *How to Hack a Paysite: What the Good Guys Need to Know*, after spending time among hackers and crackers. These days I wouldn't recommend the "dark web" to anyone, not with the rise of organized crime online. But back then, I was rated "Master Exploiter" by my putative peers, was allowed in the more private "Sploiters" forums, and was made an admin of one of the larger boards.

Publishing those articles made certain people unhappy. On the other hand, a couple of billing companies changed their code as a result, and thanked me.

My own security interest stems from November 1988, when the Morris Worm[3] was released into the wild. I was teaching Cray Supercomputer operating system internals in assembly language and octal. Several of my students that week were from the government labs being hit by the worm. They were the system gurus, and as such they kept being called out of class to get on the phone.

I had virtually a ringside seat to the breaking of the Internet. It literally was torn apart, with backbones isolated from each other for a few days to stomp the worm.

Robert Morris and his Worm taught us that relatively minor mischief can cause major havoc. Learn from the experience.

Learn from the Enemy

There's good PHP security information available online. See *Additional Reading* at the end of this article. There's also information out there that's not so good. That's not necessarily the fault of the author. "Security" is a continuous battle. Techniques and needs evolve.

My best advice concerning web services comes from *Ender's Game* by Orson Scott Card:

> *You will be about to lose, Ender, but you will win. You will learn to defeat the enemy. He will teach you how.*

Your first step in securing your web services is understanding your adversarial relationship. No doubt your website is usable, friendly, inviting. That's great.

That is *not* how to view your web services. Your web services need to be *prickly and distrusting*. Don't you want to be friendly and inviting? No, you do *not!* This is the fundamental difference between your human-visible website and your invisible web services.

Guide your humans in successfully navigating your website. When errors occur—and they will—provide your humans the information they need to complete their task.

Your web services, by contrast, are aimed at computer software that already know precisely how to use your web services API.

1 *Busines Wire: The Kardashian/Jenner Sisters Launch Individual Personal Media Apps:* http://phpa.me/kardashian-apps

2 *TechCrunch: Kardashian Website Security Issue:* http://wp.me/p1FaB8-54RC

3 *Morris Worm:* https://en.wikipedia.org/wiki/Morris_worm

Client software does not need your guidance. Extra information would just get in the way.

Instead, remember that there is one *other* consumer of your web services. Your enemy, whoever that might be, is *also* consuming your web services. Do you want to *help* your enemy hack you? Of course not!

As we'll see, there may be no way to be sure if any given web service request is legitimate or an attack from the enemy. You *must* be distrusting. You *must* assume that your web services operate in a hostile environment. That's because they *do!*

Suppose the web service request is partly correct. For example, it's a properly formed request except that one parameter is out of range. Do you provide help? No. When it comes to web services, your role is to be *prickly and distrusting.* Your role is to assume you have an attacker who has almost figured you out.

In *Ender's Game,* Ender explains:

> *I've been through a lot of fights in my life, sometimes games, sometimes—not games. Every time, I've won because I could understand the way my enemy thought. From what they did. I could tell what they thought I was doing, how they wanted the battle to take shape. And I played off of that. I'm very good at that. Understanding how other people think.*

Are we planning to play games with our attackers? Absolutely not. We don't have time for that nonsense! Our strategy is to tip the odds in our favor. Once the effort to attack far outweighs the possible reward, we are far less likely to come under attack at all.

What might *your* attackers' motivations be? Cash, glory, free content downloads? Draw from your experience to date and form your own threat analysis. The attack *method* can be different via web services, but the attack *motivations* will likely remain the same.

> **Threat Modeling:** *What motivates your attacker? What might he, she, or they be after? What might be their intrusion vector? This is threat modeling. See Threat Modeling: Designing for Security by Adam Shostack under Additional Reading.*

Web Services are Different

We've all seen "brute force" attacks before. Someone hits your website login page many times with different user name & password combinations. Alarms go off, you block a few dozen IP addresses, and it's "game over" for your attacker of the moment.

We all use CAPTCHA images during brute-force attacks to distinguish and shut down the "bots." Everything works; we've been through this before.

Do you see the attitude here? It's just another brute force attack, no big deal. Detect the bot and shut it down. It's hardly worth mentioning.

Therein lies the problem.

We need to back up and take a moment to think about how we got here. Bear with me; this is the fundamental change in thinking you need to understand.

Web Services Mirror the Site

Web services are a good thing. If you have created a well-structured RESTful API powered by PHP, you have a good thing. You have an infinitely expressive portal into your server, your business operations, your reason for existence. You'd best assume that your enemy understands this as well as you do.

We created an app for Android and iOS mobile devices. Members can now use our site through either their web browser or via the native app.

We created web services which allow our app to have the same functionality as our browser-based website. Our web services are *only* consumed by our own app. We don't publish a public API. Since nobody knows where to find our web service end points, that should make us relatively safe.

No, it does not! Learn from your enemy.

Suppose, for example, you notice that a single IP address has an excessive number of failed login attempts. By capturing the transactions (web service requests and responses for that IP address), you realize that each request has a *different* user name / password combination. The login requests are formatted correctly.

What has our enemy taught us? That he or she has our API figured out. Our enemy is able to counterfeit our web services requests. We do not know *how* our enemy figured this out. Our enemy has taught us that our web services protocol is known, inside and out.

Our Focus

There's lots of good information out there about website security and web services security. By all means, do your homework and practice the fundamentals!

The problem is that your enemy won't feel constrained to follow *your* rules. You need to work with what your enemy *does* rather than stay within the security rules. The experts will provide you help, but only the enemy can teach you how to defeat the enemy.

Security is a continuous give-and-take. Learning is continuous; you can't "do it" and be done.

This article doesn't cover the fundamentals. Yes, the fundamentals are important, and they must be your starting point, but you won't find them here. Instead we focus on learning from the attacks, from our enemy.

An App is a Bot

Here is where the first of the problems comes in. Here is where we need to begin changing our thinking.

Much of your website security is based on telling the difference between a human attacking us and a bot attacking us.

You need to recognize *why* you might come under attack. Are you simply a target of opportunity? And if so, opportunity for what? Are you a high-profile site? Might someone attack for the glory of beating you? Can someone gain free downloads?

With a bot, it's often a series of repeated attempts. For example, if someone is running a password list, we see thousands (or millions) of login attempts. If it's blatant, we simply block it. If it's merely questionable, we use something such as a CAPTCHA to distinguish between human and bot.

By "bot," short for "robot," I mean any sort of automated mechanism for interacting with our website.

The fundamental problem is that your own app is a bot. It's a program, not a human, and by definition a bot. More to the point, any of your "are you a human?" tests will fail. It's not a human.

You may well need to "white list" your web services. That is, anything you have in place for bot detection, brute-force attacks, etc., will block normal app usage.

The problem is that an attacker can format and send an HTTPS request to your web services API which looks *exactly* like a legitimate request coming from your app. The headers are the same. There is no "secret handshake" telling you that "this is your app talking" and "this is not your app talking."

Your enemy can spoof your app. This is a startling realization. You *must* understand this! This may mean that your firewall won't help, because your firewall can't tell the difference between legitimate app traffic and your enemy's attack traffic. You need to think differently.

As developers, we blithely assume that the usual server-level protections are in place. After all, it's the same server! It's the same code base, the same load balancer, the same firewall configuration.

Assuming you use HTTPS for all web services, *and* have it correctly configured, you're covered. Right? Wrong! Your enemy will be only too happy to show you what you've missed.

> **Lesson:** *How do you distinguish between normal users and attackers? With web services, you generally can't. That's why it's so easy for your attacker to appear as "a wolf in sheep's clothing." Attacks can go unnoticed.*

Web Services Need to be Efficient

You put your app in app stores so that people will install and use it. If a million people have your app, and they all use it, that amounts to a potential Distributed Denial of Service attack coming from a million different places. That is a widespread attack, and that is precisely the problem that we all hope to have!

Under the covers, of course, each copy of the app is making GET and POST requests to your server via HTTPS.

The web services code can be far more efficient than a normal web page load. The web services don't need to worry about HTML rendering or navigation bars. For example, you don't need to check the member mailbox if the mailbox content is not part of the current web service response.

RESTful web services are stateless. Generally speaking, the outcome of one web service request should not depend on the outcome of the previous web service request, or the next one. You probably don't even need the standard PHP session when implementing your web services.

This means we are all able to make our web service responses **fast**. Our web services have a far lighter load on our databases. We only hit the database for what we specifically need with this request.

This is all a good thing, right? Our enemy will show us otherwise. Keep this "efficiency" in mind as we look at a common example, *running the password list*.

Running the Password List

Taking a concrete example, the answer is obvious (after the fact).

When an attacker runs a password list against the main website, most large sites detect it rather quickly. The site administrators see a run of failed logins from the same IP address or series of proxies. DevOps is likewise aware of other attack possibilities and watches for them.

With web services, it's different. You expect a lot of traffic from the web service. Because legitimate app traffic looks just like bot traffic, the normal bot-detection approaches don't work.

Are members allowed to log in to your site via the app? That is, do your web services support member login? Remember that your web services are designed to be a *lot* faster than the main site pages.

Putting it together, this means that your attacker can run through their password list a *lot* faster when attacking via your web services. Your enemy will teach you that this is *not* a good thing! With the web services being so much more lightweight and efficient, your enemy can do a lot of damage before you know anything is wrong.

How do you protect your login web service from someone running a password list? We'll cover that in Part Two of this series.

An Open Portal

Continuing our example, say somebody ran a password list against your login web service. You learn to block the attack and move on. Your own efforts at writing efficient code worked against you. That's the nature of the game. What's the big deal?

Our enemy has more to teach us.

Your web services are as stateless and lightweight as you can make them. This means that a lot of what we've learned about PHP "security in depth" simply does not apply. The principles remain, so that means we need to find different ways to achieve those objectives.

One principle is to protect data by keeping it server-side. Browser cookies, for example, can be manipulated by an attacker. We would normally use the PHP session for maintaining state, but we try not to with the web services. You might cache non-sensitive information (such as which offers the user has already completed) in the app and keep everything in your database.

On the main site, a given database query might be five levels deep in the code, with input parameters long since checked, sanitized, and validated. When a web service makes a direct call to that same function, it won't be obvious what protections need to be in place.

There are a number of solutions to this "direct access" issue, such as a "bridge" which centralizes the web service requests and provides validation. Those details don't matter here. What's important is the attitude. We need to consider any such "hot path" a direct path for the enemy.

In any event, we have two (or more) paths to the same functionality. The main site has the functionality, and the web services expose that same functionality. All of this happens naturally. It's normal. You already had a website, and you later expanded your reach by creating the web services.

As we add a web services layer to expose that same functionality to the app (or AJAX or whatever), we're likely dealing with code that came before our time. "It's a trap!" (Admiral Ackbar, *Return of the Jedi*) As you add efficient access to old code, you may be unintentionally losing security that was "bolted on" years ago.

Observing HTTPS Traffic

You should force all of your web services to use HTTPS protocol. That means requests and responses are sent in encrypted form.

Incorrect HTTPS configuration is a common vulnerability. *Get proof of your correct configuration. See the OWASP SSL/TLS Cheat Sheet[4] for a good overview.*

This should mean that even passwords can be sent in plain text across HTTPS and be safe, right? Your enemy will show you otherwise.

The problem is that your app is "out there" in the wild. Your attacker can download and install your app just like anyone else. The app can be decompiled. All copies downloaded are identical (until you update with a new app version). The app can be installed by the enemy on a test bed of their choice.

Free tools exist to capture and display encrypted web traffic. I use Fiddler by Telerik[5] for my own web services development. It allows me to see my own HTTPS app traffic, decrypted and nicely formatted.

This is one way your enemy can learn to precisely mimic your app. You can't distinguish a legitimate web service request, coming from your app, from an attack, when not one byte is different.

Do you have security tokens? Of course! But your attacker can probably harvest a live token from the current Fiddler session and use it.

Learn from the Master

What does "learn from the enemy" mean for you? Bruce Lee, possibly the greatest martial artist in living memory, stated, "Be like water, my friend." Water instantly adapts to its environment.

Bruce Lee, quoted in *The Warrior Within* by John Little[6] describes his own self-expression:

Jeet kune do is training and discipline toward the ultimate reality in combat. The ultimate reality is simple, direct, and free. A true jeet kune do man never opposes force or gives way completely. He is pliable as a spring and complements his opponent's strength. He uses his opponent's technique to create his own. You should respond to any circumstance without pre-arrangement; your action should be as fast as a shadow adapting to a moving object.

Bruce's son Brandon Lee explained in the same book,

[The master] always talks about teaching "jeet kune do concepts." In other words, teaching someone the concepts, a certain way of thinking about the martial arts, as opposed to teaching them techniques. To me, that kind of illustrates the difference between giving someone a fish and teaching them how to fish. You could teach someone a certain block, and then they have that certain block; or you can teach someone the concept behind such a block, and then you have given them an entire area of thinking that they can grow and evolve in themselves. They can say: "Oh, I see—if that's the concept, then you could probably also perform it this way or that way and still remain true to the concept."

In other words, one does not "do" web service security. There

4 *OWASP SSL/TLS Cheat Sheet:* http://phpa.me/owasp-tlp

5 *Fiddler:* http://www.telerik.com/fiddler

6 *The Warrior Within:* http://www.amazon.com/dp/0809231948

is no particular way to establish as the "right" way. The right way is whatever keeps your attacker at bay—for now. As your enemy grows and matures, of course, so must you.

Looking Forward

In this part, *Learn from the Enemy,* we learned that we dare not think of web *service* security the same as web*site* security. The enemy does not follow "the rules," whatever they might be. We must therefore directly learn from the enemy how to block the enemy.

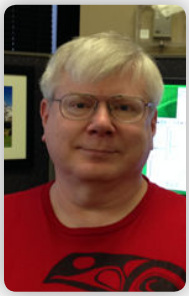
Part Two, *Security Architecture,* teaches you to meet the enemy. You've heard of Authentication and Authorization. We'll show why they do *not* work with web services. Our enemy has challenged us; we'll meet that challenge.

Part Three, *Implementing Encryption,* sounds simple. It is! The trouble is that encryption is extremely difficult to get right. In fact it's a great way to grab news headlines when you get it spectacularly wrong. We'll give you a concrete place to begin. We'll cover randomness, and how to encrypt and decrypt a string.

Additional Reading

This article serves as an introduction to securing your web services. For more advice and guidance, consult the sources collected below.

1. *Survive The Deep End: PHP Security* by Padraic Brady. Excellent survey of what you need to know about PHP security. This short online book is a good starting point. http://phpsecurity.readthedocs.org/en/latest/
2. *PHP Security Cheat Sheet* by The Open Web Application Security Project (OWASP). I include the OWASP page to point out that you should be long past dealing with these basic website security issues. But if you *are* new to PHP security, this is a good reference. https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet
3. *Web Service Security Cheat Sheet* by OWASP. Checklists are valuable. Visit this cheat sheet from time to time to ensure you still have the right things covered. https://www.owasp.org/index.php/Web_Service_Security_Cheat_Sheet
4. *Information Security* at Stack Exchange. I find the *Information Security* folks to be friendly, helpful, authoritative, and thorough. Learn to ask questions correctly and you'll be delighted with the responses. Don't be shy, but show that you've thought things through before typing out the question. http://security.stackexchange.com
5. *How to Hack a Paysite: What the Good Guys Need to Know* by Ed Barnard. This article series is old, but my exploration of attitude and motivation remains relevant. http://otscripts.com/how-to-hack-a-paysite-articles/
6. *The Art of War: Complete Text and Commentaries* by Sun Tzu, translated by Thomas Cleary. Various Twitter accounts quote this two-thousand-year-old classic, including @battlemachinne. One line at a time, this can help you retain that all-important security attitude. http://www.amazon.com/gp/product/1590300548
7. *Threat Modeling: Designing for Security* by Adam Shostack. This is the "big picture" look at formally anticipating security threats to your software. It's a tough row to hoe. But if you don't, who will? http://www.amazon.com/gp/product/1118809998
8. *Web Security: A WhiteHat Perspective*, by Hanqing Wu and Liz Zhao. This one is tough to read but worth the energy expended. I believe there were two editions of the book published, one in Chinese and one in English. A former hacker himself, the author brings a useful perspective and solid information. http://www.amazon.com/gp/product/1466592613
9. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd Edition, by Ross J. Anderson. This thousand-page monster won't be read in one sitting. Like *Threat Modeling*, this "big picture" book will give you perspective and strategies you won't find elsewhere. http://www.amazon.com/gp/product/0470068523
10. *Cryptography Engineering: Design Principles and Practical Applications* by Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. I saved the best for last. If you're planning to write security-related code, read this book first. It's a good and surprisingly fast read. You'll come away with a far better understanding of how things hold together and why. http://www.amazon.com/gp/product/0470474246



Ed Barnard has been programming computers since keypunches were in common use. He's been interested in codes and secret writing, not to mention having built a binary adder, since grade school. These days he does PHP and MySQL for InboxDollars.com. He believes software craftsmanship is as much about sharing your experience with others, as it is about gaining the experience yourself. The surest route to thorough knowledge of a subject is to teach it. @ewbarnard

An Introduction to Doctrine ORM Best Practices

Marco Pivetta

Persistence of data is a concern that involves most of our applications. How can we keep our persistence-related code easy to maintain and understand, yet make it fail-safe and future proof? Doctrine ORM is a powerful tool, and we will review a few best practices that should make using it easier, safer, and better.

Where to Start

This article should be helpful to most readers, but it assumes that you already have some basic knowledge of how Doctrine ORM[1] works.

Most examples should be comprehensible to everyone; a read of the Object-Relational mapping (ORM) documentation is recommended nonetheless.

For those not familiar with how the ORM works: It is a library that turns generic PHP objects into relational SQL database rows; see Listing 1.

LISTING 1

```
01. <?php
02. class User
03. {
04.     public $username;
05.     public $passwordHash;
06. }
07.
08. $user = new User;
09.
10. $user->username = 'Ocramius';
11. $user->passwordHash = password_hash('super-secret', \PASSWORD_BCRYPT);
12.
13. $orm->persist($user);
14. $orm->flush();
```

When is an ORM the Appropriate Tool?

Object-Relational mapping is often described as *The Vietnam of Computer Science*[2].

Mapping database table rows to objects can lead to both amazing results and terrible performance. Relational databases were never meant to solve the problem of object graphs, yet that is what it has come to.

Although recent development in the fields of graph databases and event stores spawns new hope for alternate approaches to object persistence, transactional SQL-based RDBMSs (Relational Database Management Systems) remain the best fit due to their impressive reliability.

In general, you may want to use an ORM when:

- You value data consistency, relational integrity, and transactional safety.
- You want to express business interactions with an object graph.
- Your business interactions are limited in scope, and each interaction can be wrapped in a serialized RDBMS transaction. This is typical of OLTP (online transaction processing) applications, such as e-commerce carts, message boards,and CRMs.
- You want to make quick iterations on the relational data structure, where you change your database schema often, driven by your OO-level decisions.
- You want to keep your domain logic as distant and unaware as possible from your persistence logic.

An ORM becomes a major hindrance when you start using it for purposes it is not intended to deal with:

- Online analytical processing application (OLAP), or anything that often relies on map/reduce operations, data aggregation over millions of records, and so on. These reporting concerns are out of the scope.
- Schema-less data: storing mixed data into fields usually does not play nicely with relational databases, their indexing strategies, and their performance and data-integrity features in general.

1 *Doctrine ORM:* http://www.doctrine-project.org

2 *The Vietnam of Computer Science:* http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

Doctrine logo used with permission of the Doctrine Project.
Phu Son Nguyen (http://www.phuson.com) - designer of the Doctrine logo

Where to Start

When working with a data-mapper, you should focus on the target of the mapping first. In Doctrine ORM, that would be Entities.

An Entity is, by definition, a "thing" in your domain with an assigned identifier.

A good exercise when modeling your domain is to separate concepts that require an assigned identifier from concepts that can exist without a given ID.

An entity's responsibility is to encapsulate all the behavior bound to a certain domain concept.

For instance, a `User` object would expose the API to:

- Retrieve its username.
- Fetch the full name of the person.
- Ban the person from interacting with the system.
- Handle authentication.
- Change the password used for authentication.

A Practical Example, to Warm Up

Listing 2 is an example `User` API, expressed as a class:

As you can see, this example includes no properties or implementation details. That is done on purpose, as we want to focus only on what our domain model should convey as publicly available API interactions. Also note that there is no interface for this class, as the `User` concept in our domain is a fixed concept and is not really exchangeable with anything else.

No ORM interactions have been coded so far because our model should be as unaware of the persistence concerns as possible. We should keep it this way until we have finished implementing all the little details of our code.

Coding data-first (basing the code on the properties/state of the object) would have terrible implications, as we'd end up coupling API design with encapsulated state, which is exactly what OO-design was supposed to protect us from.

Once our API has taken shape, we can actually implement its details. Although this article has nothing to do with TDD, this is the perfect chance for you to apply it in practice. We will skip to the implementation example in Listing 3.

LISTING 2

```
01. <?php
02. class User
03. {
04.     // ...
05.
06.     public function toUsername() : string {
07.         // ...
08.     }
09.
10.     public function toLegalName() : string {
11.         // ...
12.     }
13.
14.     public function ban() : void {
15.         // ...
16.     }
17.
18.     public function changePassword(string $newPassword,
19.                                    callable $hashingFunction) : void {
20.         // ...
21.     }
22.
23.     public function authenticate(string $password,
24.                                  callable $hashComparisonFunction) : void {
25.         // ...
26.     }
27. }
```

LISTING 3

```
01. <?php
02. class User
03. {
04.     private $firstName;
05.     private $lastName;
06.     private $username;
07.     private $passwordHash = '';
08.     private $isBanned = false;
09.
10.     public function toUsername() : string {
11.         return $this->username;
12.     }
13.
14.     public function toLegalName() : string {
15.         return $this->firstName . ' ' . $this->lastName;
16.     }
17.
18.     public function ban() : void {
19.         $this->isBanned = true;
20.     }
21.
22.     public function changePassword(string $newPassword,  callable $hashingFunction) : void {
23.         $this->passwordHash = $hashingFunction($newPassword);
24.     }
25.
26.     public function authenticate(string $password,  callable $hashComparisonFunction) : void {
27.         return $hashComparisonFunction($this->passwordHash, $password)
28.                 && ! $this->isBanned;
29.     }
30. }
```

Validation

After designing the previous piece of code, those who applied TDD will already have noticed a big problem related to this API: there is no real way to get data into our object.

In addition, any invalid state may cause our public API to fail.

There is a simple solution to that: keep the moving parts as tiny as possible by making state immutable and validating our object when it is built (see Listing 4):

By enforcing our object to always have required parameters at construct time, we reduce problems around validity of our object. The reason is extremely simple: because our object has the responsibility of encapsulating its internal state, it should always contain only valid state.

There is no such thing as an "invalid entity," as that would also go against the database-level constraints that we defined. Having objects with invalid state floating in our system means that we just delay bugs by making them happen after our interactions with them have already taken place.

Another few rules also help when trying to keep the state of your entities valid:

Avoid Explicit State Mutations

`public` properties are to be avoided at all costs.

The `public` keyword may seem to introduce simplifications when interacting with your data, but that also misses the point: you are **not** interacting with data but with domain models that expose meaningful API.

The same is to be said about setters: a state transition doesn't usually happen on its own but is instead caused by some more complex business interaction, i.e., hiring a job candidate is different from setting his or her `$state` to `"hired"`: a wide array of details are part of the hiring process.

Avoid Exposing State When Not Necessary

Getters should be carefully evaluated: Are they strictly required, or are you inadvertently moving some of the business logic of your domain out of the domain model in question?

In addition, is your getter exposing the state safely? Take the example in Listing 5.

Preventing this sort of mistake is actually really simple:

- Avoid exposing state, when possible.
- When required to expose state, de-reference it by cloning/copying it.
- When receiving state that may be referenced, de-reference it.
- Prefer immutable implementations, when depending on given state (use `DateTimeImmutable` over `DateTime`).

Missing any of the above leads to broken state encapsulation, which defeats the purpose of choosing an object-oriented API in first place.

LISTING 4

```php
<?php
class User
{
    private $firstName;
    private $lastName;
    private $username;
    private $passwordHash = '';
    private $isBanned = false;

    public function __construct(string $firstName, string $lastName, string $username)
    {
        if ('' === $firstName) {
            throw new \InvalidArgumentException('$firstName is required');
        }

        if ('' === $lastName) {
            throw new \InvalidArgumentException('$lastName is required');
        }

        if ('' === $username) {
            throw new \InvalidArgumentException('$username is required');
        }

        $this->firstName = $firstName;
        $this->lastName  = $lastName;
        $this->username  = $username;
    }

    // ...
}
```

LISTING 5

```php
<?php
class BankTransaction
{
    private $amount;
    private $recipient;
    private $date;

    public function __construct(Money $amount, Recipient $recipient, DateTime $date) {
        $this->amount    = $amount;
        $this->recipient = $recipient;
        $this->date      = $date;
    }

    public function getTransactionDate() : Date {
        return $this->date;
    }
}

$transaction = new BankTransaction(
    new Money('eur', 123), new Recipient('ocramius'), new DateTime()
);

$date = $transaction->getTransactionDate();

$date->modify('+1 day'); // spooky action at a distance!
```

Feedback to the Requirements

This mindset also helps us model our domain because we prevent transitions to invalid state. For instance, the following questions arise when addressing state mutations inside our `User` object:

- Is the username mutable at all? Can it be used as an identifier?
- Are first/last name allowed to change over time?
- Can a user who was banned ever be allowed to log into the system again?
- Do we want to track when a user was banned?

By reducing the number of state mutations within our implementation, we have raised many questions about details that may have been missed when first thinking about the requirements of our application.

Do not jump forward to implement solutions for these questions though, because that will just lead to discovering further details, and you may also misinterpret the requirements. Stay focused on the current functionality.

Wiring Everything Together with the ORM

We can now put everything together with doctrine mappings.

For our `User` object to be a valid entity, it must be given an identifier. Because we didn't yet determine whether the username is a valid identifier, we need to create a surrogate identifier to save our entity in a database.

On Auto-Generated Identifiers

Most developers would instinctively jump at `AUTO_INCREMENT` or `SERIAL` for such an identifier, but here we hit another big issue: we break the "validity" rules that we just set. In fact, an entity with an assigned identifier generated on the database side is not in a valid state until it is actually saved in the database.

Therefore, using an auto-incremental column for an identifier is usually considered an issue.

Fear not! There is an extremely simple solution to this problem, which is to generate the identifier when the entity is actually created. That ensures consistency, as the entity never reaches an invalid state. To do so, we can simply use the ramsey/uuid[3] library, which provides a simple way to generate v4 UUIDs (see Listing 6).

If you are not familiar with UUIDs (universally unique identifiers), then it should suffice to know that having two generated V4 UUIDs with the same value is less likely than is an asteroid hitting this planet. Should an asteroid hit this planet, then we will have bigger problems than fixing entity identifiers.

Mapping

Now that our entity is complete, we can finally start wiring it together with the ORM. To do that, I strongly recommend using XML mappings because we want to keep mappings separated from our entity and easy to validate via XSD (XML Schema Definition) as shown in Listing 7.

Do not worry too much about precision: you can fine tune the schema when it is time to do so.

ORM Independence?

As you can see, the `User` object works with or without the ORM, which should be your goal. Keeping it ORM-independent will be an ongoing effort, but you should never give it up, as it will allow you to get rid of the tool should it become obsolete. What matters is that the interactions that you implemented fit your needs.

As a rule of thumb, try implementing your domain models to make them work in such a way that `serialize()` and `unserialize()` can be used for saving them. If you manage to do so, then you will not be dependent on the persistence layer inside your domain logic.

LISTING 6

```
01. <?php
02. use Ramsey\Uuid\Uuid;
03.
04. class User
05. {
06.     private $id;
07.
08.     // ...
09.
10.     public function __construct(string $firstName, string $lastName, string $username) {
11.         // ...
12.
13.         $this->id = (string) Uuid::uuid4();
14.     }
15.
16.     // ...
17. }
```

LISTING 7

```
01. <?xml version="1.0"?>
02. <doctrine-mapping
03.     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
04.     xsi="http://www.w3.org/2001/XMLSchema-instance"
05.     schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd"
06. >
07.   <entity name="User">
08.       <!-- don't map UUIDs as strings: it's a waste of performance! -->
09.       <id name="id" type="guid">
10.           <generator strategy="NONE"/>
11.       </id>
12.       <field name="firstName" type="string"/>
13.       <field name="lastName" type="string"/>
14.       <field name="username" type="string"/>
15.       <field name="passwordHash" type="string"/>
16.       <field name="isBanned" type="boolean"/>
17.   </entity>
18. </doctrine-mapping>
```

3 *ramsey/uuid: https://github.com/ramsey/uuid*

Saving and Retrieving User Entries

We now need to solve a few problems related to fetching and saving entries.

In terms of fetching records, your immediate instinct would be to use the `EntityManager#find()` or `EntityManager#getRepository()` API. Repositories are actually a concept that is really near your domain because you will customize how to find a `User` depending on your needs.

Domain Repositories

We have to step back from directly using the ORM API and instead code the behavior we want. Listing 8 is an example of a repository API that suits our needs.

We can go further and split this interface into single functions (see Listing 9).

LISTING 8

```php
<?php
use Ramsey\Uuid\UuidInterface;

interface UserRepository
{
    /**
     * @throws UserNotFound
     */
    public function getByUsername(string $username) : User;

    /**
     * @throws UserNotFound
     */
    public function getById(Uuid $id) : User;

    public function persist(User $user) : void;
}
```

LISTING 9

```php
<?php
use Ramsey\Uuid\UuidInterface;

interface GetUserByUsername
{
    /**
     * @throws UserNotFound
     */
    public function __invoke(string $username) : User;
}

interface GetUserById
{
    /**
     * @throws UserNotFound
     */
    public function __invoke(Uuid $id) : User;
}

interface PersistUser
{
    public function __invoke(User $user) : void;
}
```

Having functions separated like this will simplify optimization details when we want to switch between ORM APIs and raw SQL-level operations, which rely on different dependencies.

Let's go on with implementing them; see Listing 10.

These are easy to test and, more importantly, easy to use!

LISTING 10

```php
<?php
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\Common\Persistence\ObjectRepository;
use Ramsey\Uuid\UuidInterface;

final class DoctrineGetUserByUsername implements GetUserByUsername
{
    /** @var ObjectRepository */
    private $users;

    public function __construct(ObjectRepository $users) {
        $this->users = $users;
    }

    public function __invoke(string $username) : User {
        $user = $this->users->findOneBy(['username' => $username]);

        if (!$user instanceof User) {
            throw new UserNotFound::fromIdentifier($id);
        }

        return $user;
    }
}

final class DoctrineGetUserById implements GetUserById
{
    /** @var ObjectRepository */
    private $users;

    public function __construct(ObjectRepository $users) {
        $this->users = $users;
    }

    public function __invoke(Uuid $id) : User {
        $user = $this->users->findOneBy(['id' => (string) $id]);

        if (! $user instanceof User) {
            throw new UserNotFound::fromIdentifier($id);
        }

        return $user;
    }
}

final class DoctrinePersistUser implements PersistUser
{
    /** @var ObjectManager */
    private $objectManager;

    public function __construct(ObjectManager $objectManager) {
        $this->objectManager = $objectManager;
    }

    public function __invoke(User $user) : void {
        $this->objectManager->persist($user);
    }
}
```

Domain Interactions

You may have noticed that we do not `flush()` inside our repository. We will delegate this functionality to a thin service layer that is responsible for mediating communication between the application layer and the actual business logic.

In recently developed applications, this mediation is handled by a command bus. For simplicity and clarity, we will code it directly in our application layer. The code in Listing 11 is oversimplified for the purpose of the example and doesn't include any validation:

In this example, we don't even need to use `EntityManager#flush()` ourselves because the `EntityManagerInterface#transactional()` already does that for us.

Wrapping Up

As you can see, there is much to be said about your persistence layer. This article just scratched the surface, allowing you to have an overview on how to cleanly separate the tool (Doctrine ORM) from your relevant bits of business logic and your application layer.

Always strive for separation of concerns and adapters to external libraries because that introduces great flexibility in your application, allowing you to freely move parts around.

Marco "Ocramius" Pivetta is a software consultant at Roave. With over a decade of experience with PHP, he is part of the Zend Framework CR team, Doctrine core team, and is also active in the community as a mentor and supporter. When not coding for work, he usually hacks together new concepts and open source libraries, or simply provides Q&A on IRC.

LISTING 11

```php
<?php
use Doctrine\ORM\EntityManagerInterface;

class ChangeUserPassword
{
    /** @var Authenticate */
    private $authenticate;
    /** @var GetUserById */
    private $getUser;
    /** @var PersistUser */
    private $persistUser;
    /** @var EntityManagerInterface */
    private $entityManager;

    public function __construct(
        Authenticate $authenticate,
        GetUserById $getUser,
        PersistUser $persistUser,
        EntityManagerInterface $entityManager
    ) {
        $this->authenticate  = $authenticate;
        $this->getUser       = $getUser;
        $this->persistUser   = $persistUser;
        $this->entityManager = $entityManager;
    }

    public function __invoke($request, $response) : Response {
        $this->entityManager->transactional(
            function () use ($request) {
                $user = ($this->getUser)(($this->authenticate)($request));

                $user->changePassword($request->getPost('newPassword'));

                ($this->persistUser)($user);
            }
        );

        return new RedirectResponse('/my-profile');
    }
}
```

Directing Requests with FastRoute

Matthew Setter

A fundamental aspect of modern web-based applications is routing, as the routing engine can be critical to the application's performance. This month we look at a library which is gaining significant traction in the PHP community—one which provides blazingly fast performance to applications both large and small. It's called FastRoute.

What is FastRoute?

As part of maintaining a blog about all things Zend Framework, I came across FastRoute[1] when I started investigating Zend Expressive, as it's the default routing package. I'd not heard of it at the time, and was only familiar with the Zend-MVC router.

But when I experimented with the installer to see what Zend Expressive had to offer, I saw FastRoute as the default and decided to give it a go. To be honest, I was more than pleasantly surprised—even after only a short time using it.

The main reason is that, unlike previous iterations of Zend Framework, I didn't have to create different route types to handle different situations. The route definition provides all the required information.

I could create simple routes, such as `/`, which point to the homepage of a site. Or I could create ones with parameters, such as:

```
/users/{type:(?:all|worker|management)}
```

And these aren't even the most complex or sophisticated of routes.

But I digress. What is FastRoute? The official definition from the repository is this:

> **FastRoute** - *Fast request router for PHP. This library provides a fast implementation of a regular expression based router.*

As you can see, and as the route examples highlighted, routing in FastRoute relies on regular expressions. At this point, you might feel reluctant to use it, as even the mention of the term *regular expressions* seems to evoke fear in many people.

I've worked with developers who'd rather write copious lines of code than consider writing even the simplest of regular expressions, citing all sorts of half-baked excuses as reasons for justifying their reluctance.

But it needn't be that way. I hope by the end of this column that, in addition to having a basic understanding of FastRoute, you'll also have, at least, a basic understanding of regular expressions.

Anyway, back to FastRoute. FastRoute was initially developed by Nikita Popov as an experiment, after perusing another PHP routing library and believing that their approach was all wrong. He goes into great detail in a post he wrote[2] explaining how FastRoute works.

Here's an excerpt:

> *…after a cursory look at the code I had the strong suspicion that the library was optimizing the wrong parts of the routing process and I could easily get better performance without resorting to a C extension. This suspicion was confirmed when I took a look at the benchmarking code and discovered that only the extremely realistic case of a single route was tested.*

So what began as an experiment eventuated as a blazingly fast routing library. I love situations such as these! Given that the package relies so heavily on regular expressions to use it even modestly, not much code is required.

Specifically, there are three steps. These are:

1. Define a route
2. Marshal the data
3. Handle the request

There's no complicated overhead to go through, no series of hoops to jump through. Just these three steps.

Installation

But before we can do any of these, we have to install it. Like most modern PHP libraries, FastRoute is available via Composer. To install it, whether in a new project or an existing one, running the following from the terminal, in the root of the project, will add it as a dependency.

```
composer require nikic/fast-route
```

With that done, the library will be available in the project's `vendor` directory, and the autoloader will be updated to provide it. Let's look at the basics.

As I mentioned earlier, to use FastRoute, there are three key components involved. The first is defining a route, the second is marshaling the data, the third is handling the request. Let's step through each of these so that you see how it works, starting from the top.

Defining a Route

In the code sample below, you can see a basic routing table, composed of one route. There are a number of working parts involved, which need to be considered.

```
$dispatcher = FastRoute\simpleDispatcher(
    function (FastRoute\RouteCollector $r) {
        $r->addRoute('GET', '/', new homePageHandler());
    }
);
```

1 *FastRoute: https://github.com/nikic/FastRoute*

2 *Fast request routing using regular expressions: http://phpa.me/nikic-fast-routing*

First, the dispatcher is an instance of `simpleDispatcher`. FastRoute provides two dispatcher classes, `simpleDispatcher` and `cachedDispatcher`. The key difference, as you may have inferred, is that `cachedDispatcher` can construct the dispatcher from a cached copy of the dispatcher, whereas `simpleDispatcher` cannot.

Then there's the callable supplied to `simpleDispatcher`, which takes a `RouteCollector` object. As the name implies, it is responsible for collecting a series of routes together into a routing table.

It conversely has two key methods: `addRoute()`, which adds a route to the routing table, and `getData()`, which returns the routing table data. We next add a route to the home page by calling the addRoute method, and pass in three parameters. These are:

- The accepted route methods
- The route regex
- A route handler

The route methods, which can be one or more of the HTTP methods, can be supplied individually as a string, or as an array of the method names.

The route handler can be either the name of a function or a callable object. With all that in mind, you can see that the top-level route `/` can only be accessed with a `GET` request and that it's handled by a `homePageHandler` object's `__invoke` method (since no method was specified).

Here's the definition of `homePageHandler`:

```
class homePageHandler
{
    public function __invoke()
    {
        return "This is the home page";
    }
}
```

It's nothing special, but enough to get the job done, returning a string showing you've landed on the home page. In a real application, you'd likely use a templating engine like *Blade*, *Zend*, or *Twig* and return the rendered results of a template file. But this suits our needs for now.

Marshaling the Data and Dispatching

Next comes marshaling the request data. This snippet, taken verbatim from the FastRoute documentation, provides an example of how to pull the required information from the environment which the dispatcher needs, that being the requested HTTP method and URI.

```
$httpMethod = $_SERVER['REQUEST_METHOD'];
$uri = $_SERVER['REQUEST_URI'];

// Strip query string (?foo=bar) and decode URI
if (false !== $pos = strpos($uri, '?')) {
    $uri = substr($uri, 0, $pos);
}

$uri = rawurldecode($uri);
$routeInfo = $dispatcher->dispatch($httpMethod, $uri);
```

Focusing on the dispatch method, depending on the routing table and the request, it will return one of the following three arrays:

```
[FastRoute\Dispatcher::NOT_FOUND]
[FastRoute\Dispatcher::METHOD_NOT_ALLOWED,
                ['GET', 'OTHER_ALLOWED_METHODS']]
[FastRoute\Dispatcher::FOUND, $handler,
                ['varName' => 'value', ...]]
```

Stepping through those, if the route is not found, then `NOT_FOUND` is returned. If the route is available but the requested method is not one of the allowed ones for that route, then it will return a `METHOD_NOT_ALLOWED` response, along with an array of the allowed methods for that route.

Finally, if the route is valid and the requested method is allowed, then it will return the handler, along with any extracted route parameters in an associative array. Now let's look at the dispatch handler.

Handling Requests

In Listing 1 you can see a simple switch statement, also taken from the FastRoute documentation. It has a case to handle all of the three possible return values from the dispatch method. If the route is not found, then it prints the string `"Not Found"`.

LISTING 1

```
01. switch ($routeInfo[0]) {
02.     case FastRoute\Dispatcher::NOT_FOUND:
03.         print "Not Found";
04.         break;
05.     case FastRoute\Dispatcher::METHOD_NOT_ALLOWED:
06.         $allowedMethods = $routeInfo[1];
07.         printf("Method Not Allowed on this route. Allowed methods are: %s",
08.             implode(', ', $allowedMethods));
09.         break;
10.     case FastRoute\Dispatcher::FOUND:
11.         $handler = $routeInfo[1];
12.         print new $handler();
13.         break;
14. }
```

If the method used isn't allowed for that route, then it prints that the method used wasn't allowed, along with a concatenated string of the allowed methods. Finally, if the route was found and an allowed method used, then the handler is retrieved and invoked.

This case serves to illustrate what is going on. In a more sophisticated application, you'd likely implement the route handling using a factory or abstract factory pattern, so that the route is handled properly, especially as some routes may take arguments, some may not. Now let's build up in complexity and look at how to handle route arguments.

Routes with Arguments

Let's now say that we need to expand our application and add a route to handle rendering email stats for a user. Let's assume that we have a fictitious application which provides a service such as Mailchimp[3], or Mailgun[4].

3 Mailchimp: http://mailchimp.com

4 Mailgun: http://www.mailgun.com

When the user logs in, they have the option to view the number of incoming, outgoing, and bounced emails from a regular campaign to their mailing list. Let's add a route to handle that functionality (see Listing 2).

In this code, I've added a second route which accepts two route arguments. The first, `type`, can be one of `all`, `incoming`, `outgoing`, or `bounced`. The second, `limit`, is a numeric value. This way, the user can view a specific email type, or all of them, and they can also limit the number of records, instead of perhaps being deluged with information.

Now let's update the code to handle both routes. I'll first implement a function, called `dispatchRoute`, which takes two variables. The first will be the route's handler, and the second is an array of route variables extracted from the request, based on the route definition.

Here it checks which type `$handler` is an instance of, and prints the results of invoking it. This is OK, as both objects' `__invoke` methods return a string. The first accepts no arguments, the second does. To that, the values for `type` and `limit` are supplied.

Next comes the definition for the new `emailCampaignHandler` class. The class' `__invoke` method prints a string showing the number of emails of the type requested, based on the arguments supplied. I wanted to keep it simple. So if type is set to `all`, the sentence won't be grammatically correct. But for a simplistic example, is that really such a crime?

```
case FastRoute\Dispatcher::FOUND:
    list(, $handler, $vars) = $routeInfo;
    dispatchRoute($handler, $vars);
    break;
```

LISTING 2

```
01. <?php
02.
03. $dispatcher = FastRoute\simpleDispatcher(function (FastRoute\RouteCollector $r) {
04.     $r->addRoute('GET', '/', new homePageHandler());
05.     $r->addRoute(
06.       'GET',
07.       '/users/email/campaigns/{type:(?:all|incoming|outgoing|bounced)}/{limit:\d+}',
08.       new emailCampaignHandler()
09.     );
10. });
```

LISTING 3

```
01. function dispatchRoute($handler, $vars) {
02.     if ($handler instanceof homePageHandler) {
03.         print $handler();
04.     }
05.
06.     if ($handler instanceof emailCampaignHandler) {
07.         print $handler($vars['type'], $vars['limit']);
08.     }
09. }
```

LISTING 4

```
01. <?php
02. class emailCampaignHandler
03. {
04.     public function __invoke($emailType, $recordLimit)
05.     {
06.         return sprintf(
07.           "Retrieving %d %s campaign emails",
08.           $recordLimit,
09.           $emailType
10.         );
11.     }
12. }
```

Finally, I've updated the FOUND case to use the new `dispatchRoute` function. And in essence, that's all that's required to make use of FastRoute. If you carry out those three steps, then you're off and running.

However, perhaps you want something more sophisticated, or even simpler to use. If so, before you consider writing a wrapper around it, consider the following three libraries, which all build upon it:

- Route, http://route.thephpleague.com
- Air-PHP, https://github.com/air-php/fastroute
- Zend Expressive FastRoute, https://github.com/zendframework/zend-expressive-fastroute
- SimpleRoute, https://github.com/AndrewCarterUK/SimpleRoute

I've some experience with Route, by *The League of Extraordinary Packages*, and *Zend Expressive FastRoute*. But I can't personally vouch for Air-PHP and SimpleRoute. Both of the aforementioned libraries do a great job of providing more sophistication. But they're not the subject of this month's column.

FIGURE 1

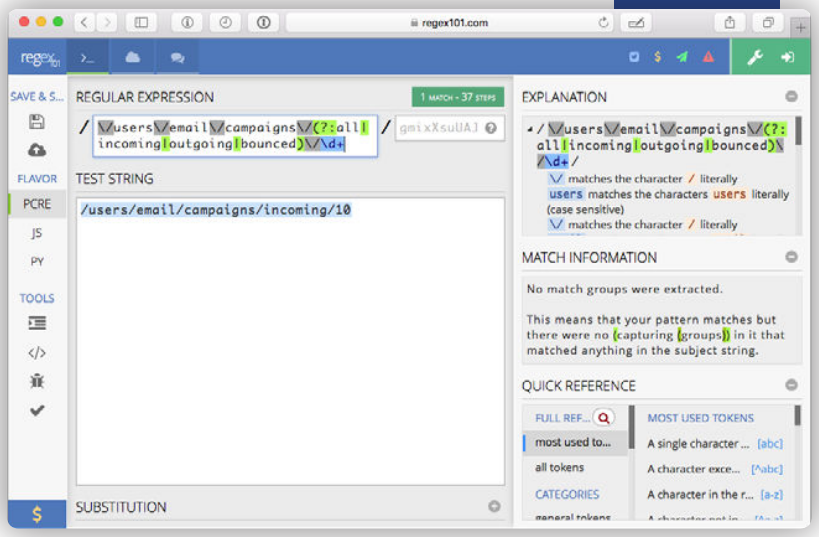
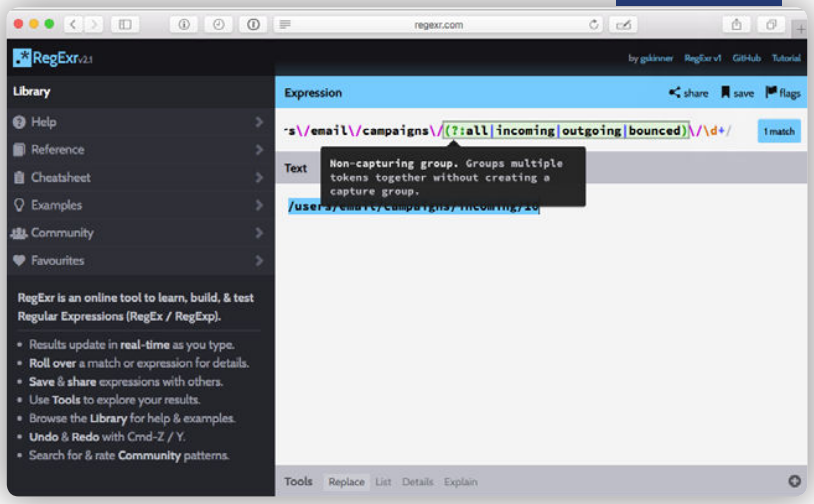
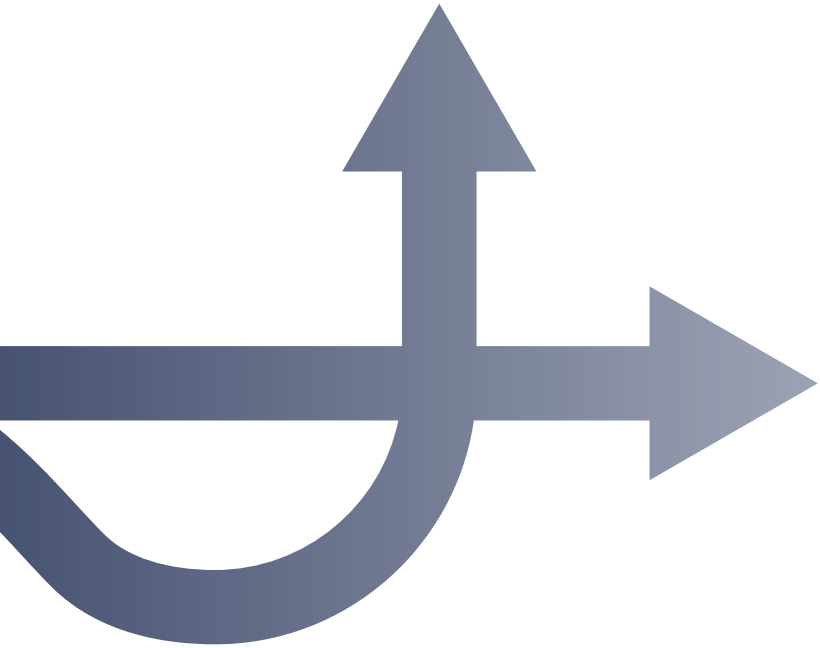


FIGURE 2





Helpful Regular Expression Resources

If you've stuck this far with me, thank you. I respect your time and patience, even if regular expressions seem like a scary concept, or one only for "rocket scientists." Because I love regular expressions, and believe that if you build even an average knowledge of them, that FastRoute will be an ideal routing package, here's a couple of suggestions to help you build them more manageably.

Refex101.com

My first recommendation is Regex101.com[5], which you can see in Figure 1. It's an excellent resource which guides you through the process of building regular expressions, from the simple to the sophisticated. You can see that I've taken the regex from the second route and pared it back to just the essentials.

On the right-hand side, you see how the regex matched the supplied test string. It provides a host of information in the quick reference section, so you can expand on the route, as well as an excellent debugging tool so that you can make the regex more efficient. Check it out—especially as it's free.

Regexr

The next suggestion is Regexr[6], which you can see in Figure 2. I feel it's likely equally as powerful, but the interface isn't as polished as regex101.com. That said, it provides an easy a way of building and testing regular expressions, along with excellent resources in the form of cheatsheets, online help, and regular expression reference documentation.

Regular-Expressions.info

Thirdly, there's Regular-Expressions.info[7]. This site is the granddaddy of regular expression resources. It has everything, and I mean everything, you need to know. From the basics, right up to such complex topics as recursion and balancing groups.

They have a native application, called RegexBuddy[8], which I believe is only for Windows. But if you're on Windows, and want both a reference and a powerful application, definitely check that site out.

And That's a Wrap

I hope this has whet your appetite enough to check out FastRoute, and that you've not been put off by the fact that it's built around regular expressions. It's an excellent package, one I'm loving using in my Zend Expressive applications, and one I'm keen to make much greater use of. I hope that you are as well.

Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, the podcast about the business of freelancing as a software developer and technical writer, and editor of Master Zend Framework, dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com.

5 *Regex101.com: http://regex101.com*
6 *Regexr: http://regexr.com*
7 *Regular-Expressions.info: http://www.regular-expressions.info*
8 *RegexBuddy: https://www.regexbuddy.com*

You Had One Job

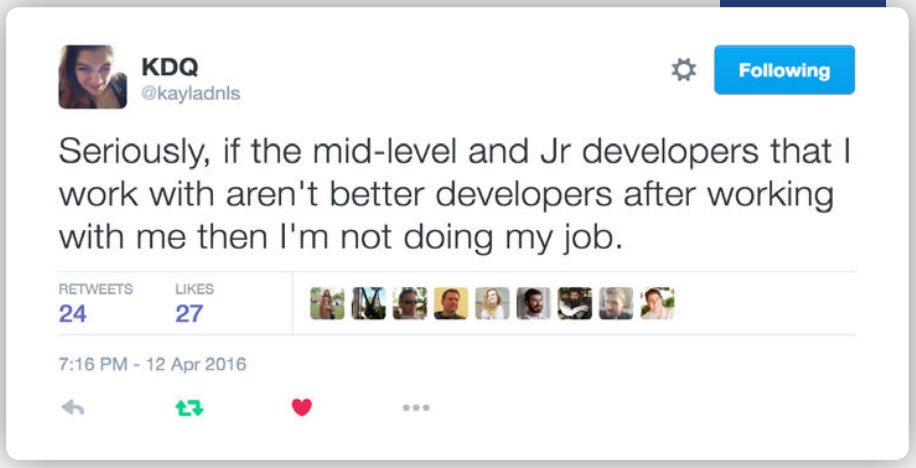
David Stockton

I don't think it's much of a stretch to say that the majority of people reading this column are senior developers (or beyond) or are quickly progressing to be senior engineers. This column is primarily directed toward the senior developers or managers of senior developers. It's about how you can level up yourself, but more than that, it's about the responsibility you have in leveling up those around you. As a senior engineer, your job responsibility doesn't end at producing loads of code or being the subject matter expert on your projects.

Why This Topic?

Every month for the past 17 months or so, either I come up with a topic to write about or Oscar suggests a few and I pick from something that sounds interesting. It's not always easy to keep choosing topics and keep writing, month after month. This was one of those months until a couple of days ago, when I saw this tweet from Kayla Daniels (See Figure 1).

FIGURE 1



There's so much wisdom and truth to that. If you're working as a senior developer and you're only focused on advancing yourself and your own skills, you're doing yourself, your company, and your coworkers a disservice.

What Does a Senior Developer Do?

Clearly, part of the role of a senior developer is to develop code. But that's not all. In my previous position, I worked my way up to a senior developer position. I wrote a lot of code, thought I was pretty good, and ended up as the subject matter expert on nearly every aspect of the projects I was working on. I had a few coworkers and tried to ensure that everything I was doing was clear and understood by my fellow developers.

At my current company, I came over as Director of Software Engineering, being recommended by one of the people that I had worked with previously. One of the things I'd noticed as I started interviewing candidates was the enormous range of skill level in the developers who came in to interview, even when they had the same title. Specifically, it seemed that most places promoted developers to the "Senior Developer" or "Senior Engineer" title based primarily on a certain number of years of service or experience, rather than any specific skills or responsibility they displayed. I decided early on that I wanted to establish some standards for job titles and what that meant.

My first stop was Google, trying to see if anyone else had written and shared anything regarding job titles, responsibilities, and expectations and what each level meant. I came up with nothing. Either no one had done this before, or if they did, they didn't want to share. So I decided to write my own and align my developers into the levels. By writing it all out and sharing it, the idea was that developers had a very clear indication of what it takes to move from one level to the next, of what's expected. It did mean that some people we had or that we would hire might be coming in at a "lower" job title level than they had at another company, but I wanted to ensure that at least within my company, the titles meant something.

I haven't shared this list outside of my company until now, but I feel that sharing it with you might help you out, help out your company, and help you level up. At my company, developers are expected to be performing at the level they want to be promoted to in order to be considered for promotion. We don't promote and hope that you'll eventually grow into the title; we expect you to expand your skills and responsibilities and then we upgrade your title to match what you do. The following is the list of titles and responsibilities that I came up with.

You will notice a theme among these descriptions: quality, mentoring, design, and architecture. As a developer progresses through the ranks, he or she should be producing higher quality code. Typically this translates into less code that is more flexible and has fewer bugs. This is the quality aspect. He or she should be working toward increasing the skills of their peers through teaching and helping. This is the mentoring aspect. Typically junior level developers aren't going to be asked or expected to build an application from the ground up. However, as a developer progresses, there will be a greater expectation that he or she will be able to not only understand how an application is built, but also design and architect new applications and products. Additionally, it is expected that the ability to successfully build working, maintainable software becomes more repeatable as the developer builds his or her skill set.

Software Engineering Job Levels and Responsibilities

At all levels of Software Engineering, it is expected that individuals are striving to learn and improve their craft. Suggestions for process improvements and changes that benefit the component, team, and the company are always welcome. Engineers at the lower levels are expected to learn from more senior engineers and senior engineers are expected to mentor junior engineers. Progression through the levels is not limited to certain times of year. It is expected that engineers who wish to be promoted exhibit many, if not all, of the skills and responsibilities required in the higher levels. Engineers should strive to improve themselves and their skills.

Software Development Apprentice

As a software development apprentice, you are expected to be able build simple functions, classes, and methods with supervision and heavy review. This level is reserved for developers new to development and with no real practical, real-world experience.

The developer is expected to be able to complete simple functions and methods with moderate supervision and direction. Learning about the SDLC and unit testing is also expected.

Software Developer

The Software Developer role is for those who are just starting out in the software development field. This role should be writing simple classes, methods, and functions and learning about the SDLC, including reviewing code, unit testing, documentation, and coding styles and source control. At this level, coding may be more of a "code by coincidence" than coding on purpose, but it is expected that an individual will move out of this level quickly.

The key to this level is that there is a lot of learning taking place and the individual should strive to learn as much as they can to progress to Software Engineer. Individuals at this level may have only some familiarity with a few of the components of the system they are working on.

Software Engineer

As a Software Engineer, coding is now significantly more deliberate and less "code by coincidence." The individual should be comfortable with basic unit testing concepts and be able to write simple classes, methods, and functions with some instruction but without much supervision. The individual should be paying more attention to details and beginning to emphasize development testing of their own code to eliminate all classes of obvious problems (syntax errors, invalid function calls, etc.), and should be working to increase the value their tests provide to the application. Software Engineers at this level should be quite familiar with several of the code components and possibly have advanced understanding of a few.

Senior Software Engineer

The Senior Software Engineer has an advanced knowledge of many of the components in the system and has likely written or been a key developer on several, and may be the expert in some of the components. This individual can design and develop classes, methods, and functions in a repeatable fashion and rarely writes code by coincidence. The Senior Engineer will program unit tests that are valuable in ensuring that the system is functioning properly. The Senior Engineer is developing with an emphasis on building and maintaining quality software.

The Senior Software Engineer should be able to create designs and plan for more complex interactions between classes, modules, and functions, but may still need help and supervision and review of some aspects of the design. The Senior Software Engineer will be heavily involved in reviewing other developers' code and making suggestions to ensure the quality of the final product. At this level, the engineer can take the lead role in design and development of small to medium components of the system, with the expectation that they are reviewed by peers and more senior members of the team. At this level, mentoring occurs for more junior level team members.

Staff Software Engineer

As a Staff Software Engineer, the individual will be mentoring junior developers. The focus on quality has increased beyond that of a Senior Software Engineer. Advanced unit testing, including a good understanding of mock objects and other testing concepts, is expected. The Staff Software Engineer should be able to identify and fix anti-patterns, poor development practices, and other issues that would lead to increasing technical debt.

The Staff Software Engineer is able to design and develop modules of moderate to high levels of complexity with minimal supervision. The ability to design and think through how software works is critical at this level. Staff Software Engineers are likely a lead on every project they are working on, or are involved with ensuring that Senior Software Engineers are properly leading the projects. They are responsible for ensuring quality for their team for the modules that are assigned to them.

The Staff Software Engineer has a thorough understanding of the entire system or application and is at expert level on many of the components.

Principal Software Engineer

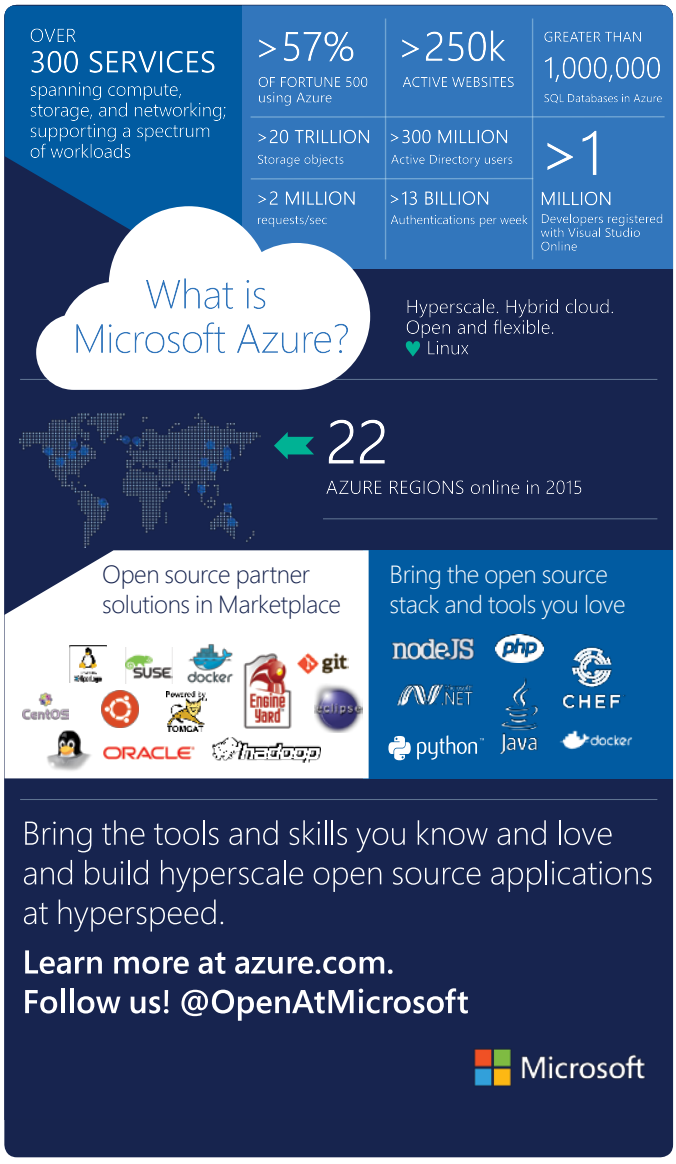
At the Principal Software Engineer level, quality is of utmost importance. Engineers at this level are likely practicing and encouraging test-driven development with the rest of their team. They are able to plan, design, implement, and lead a team through the full Software Development Life Cycle. Unit testing is a given and identifying incorrect and useless tests by review is expected. Suggesting and implementing valuable testing procedures is required. Suggestions for problem solving and process improvement are also expected at this level. Mentoring should be continuing and Principal Software Engineers should be mentoring other mentors as well.

Researching, finding, and suggesting new software engineering practices and tools is expected. Principal Software Engineers are consistently successful in the delivery of projects and software components, as well as successfully motivating the troops. This individual is critical to the success of the projects to which they are assigned; they will also likely play an important role in design and decision making on other projects and products to which they may not be directly assigned.

Beyond Senior

As you've probably noticed, I went ahead and included two levels that go beyond the senior level, placing more emphasis on quality, mentoring, design, and architecture. I wanted to provide a path for people who are good at tech, really like doing it, and don't feel that management is the career path they want to head down. It's not uncommon at larger software companies to have an advanced tech path, and I wanted to provide it at my company as well. Too many employers see senior developers as future managers, even though they may not be good at it and may not have any interest or aptitude for management.

FIGURE 2



With Great Power...

I've outlined the responsibilities, actions, and characteristics that I would expect to see from a developer at each level. Starting out, the emphasis is on learning and improving one's own skills, and understanding how to build good software that is maintainable. Once we reach the level of Senior Software Engineer, the focus starts to shift outward, with the Senior Engineer mentoring the more junior members of the team. As I mentioned, Kayla's tweet was what gave me the idea for this article, but also deserving of credit was this tweet (see Figure 2) in reply from Laura Thompson.

Before I continue, I want to say, if you're not following Kayla (@kayladnls) or Laura (@lxt), you should be.

If you're in the role of Senior Developer or Senior Engineer or something higher, your responsibility, and your challenge, is to bring up those around you. It may mean that your total output in terms of code may diminish, but that's OK. Suppose you're the Senior with eight other developers. If you produce only half the code you would have, but each of those other developers improves by 10%, it's a huge win. And the reality is that a 10% improvement of a junior developer who has a good, attentive mentor is on the low end. It's probably closer to 20–50% improvement, if not even higher. This means that mentoring is a huge win for everyone involved, from the junior mentees, the mentors, the team, and the company.

How Can You Do This?

Again, it's hard to see your code output decrease. It happens. You may be seen as your team's "rock star" developer, the go-to engineer that everyone relies on in order to get things done. It can feel good to be that lynchpin, but by holding onto knowledge, becoming that island of getting things done, you're doing yourself and others a disservice. There's a well-known term in the industry for how many (or few) people hold the critical knowledge about how the system or software works. It's called the "bus factor," and it essentially refers to how many of your developers being "hit by a bus" would bring progress to a halt.

While being a bit macabre, the "bus factor" is an important concept that needs to be understood, not just by the company, but by the developers. Being hit by a bus could just as well refer to a developer getting sick, leaving the company for another opportunity, or just taking a vacation. If you're getting a phone call when you're on the beach about how to get the message queueing system you wrote to start back up, or how to stop emails from sending because a customer messed something up, you *are* the bus factor.

Additionally, the "bus factor" can be triggered by burnout. This is typical when projects are poorly managed and result in "death marches," excessive and continued overtime, and a feeling of helplessness and despair. I wrote more about this topic in the July 2015 issue of php[architect][1], and I'd recommend you check it out if you haven't already. Burnout can become a bus factor even if the developer hasn't left the team—the quality of their code declines to the point where they are essentially making negative contributions to the project.

Some developers, sysadmins, and other IT workers like to build software in a way that's hard to understand, obfuscated, or just plain weird because they feel that if they are the only person who knows how to do something, they've got job security. The reality is that they've put themselves in the position of being the bus factor and have likely ensured they are so indispensable that they lose the chance of promotions or other career advancing opportunities, because the company cannot afford to lose them. At least for the time being. Eventually another engineer will come along and figure out what was done, why, and how to fix it, which means the original engineer will no longer be able to hold the company hostage.

Instead, I recommend working to make yourself replaceable. If you've developed new processes, code, systems, and software, and documented them well, trained up fellow developers on how everything works so that any of them can do all of it, you've opened the possibility of promotions, new projects, new opportunities, and more interesting work. And in seeing how you handle this, the developers that you're training and mentoring will see you as an example of how things ought to be done, and hopefully follow in those footsteps, training the next generation of developers to understand those systems and the new ones they create.

It's About the Mentoring

Whether you got into this industry following a degree in Computer Science or similar, or didn't go the college route and taught yourself to code, or some other path, you've got skills and ideas that others can learn and benefit from, regardless of what level of developer you are. Sharing this knowledge and teaching others is a great way to increase your own understanding of the topic. If you're in a senior role or above, you should be doing this already with your coworkers.

If you're not doing it already, I'd encourage instituting "lunch and learn" sessions, where a developer can teach the others about something in software, whether it's a new technique or a better explanation of the basics. If you don't have anyone who wants to present, Cal Evans offers discounts on his Nomad PHP or Day Camp 4 Developers (DC4D) video licenses for teams to help facilitate these learning sessions. Since sharing your knowledge can help others improve, you can do this even outside your company. Get involved with your local user group and give a talk. Submit a few topics you're interested in to a conference and speak about it there. You can record screen casts or Google Hangouts on Air, explaining topics and offering them for other developers to view and learn from. Blog about things you know to share your knowledge.

On freenode IRC in #phpmentoring and at https://phpmentoring.org, you can find a great group of people in the PHP community who have a goal of connecting mentors and mentees. Some members have these relationships in both directions, being mentored in some areas by people who are more skilled and have more knowledge in some areas, and at the same time sharing their knowledge and mentoring others in their areas of expertise.

Sometimes managers are hesitant to advocate for training for their people. I feel this is a huge mistake. Learning is vital. The following is cliché, but I feel it's completely true. The story goes that a manager is trying to justify a training budget for her developers. Her manager argues that by providing training, the developers would have better skills and be more likely to leave or be poached by recruiters. She argues back that while that is true, what happens if they never learn new things and they stay. Developers who are challenged and continuing to learn are more likely to stay in a position, all other things being equal.

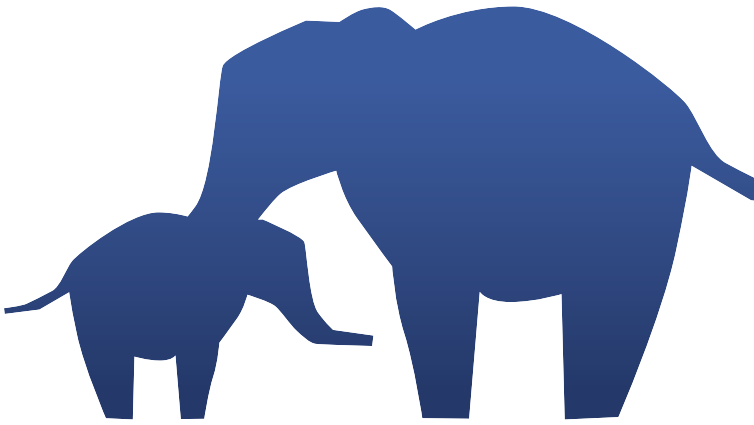
Climbing By Pulling Others Up

While some individuals climb by stepping on others and pushing them down in order to gain higher ground, a much better way is to pull others up. Share your knowledge, train others, and learn new concepts with the goal of sharing that knowledge. The reason many people are senior level developers or higher is not because they've done everything right, but because they've made more mistakes and know how to recover. I'm able to help people at work sort out problems and issues quickly because chances are I've made the same mistakes in the past and can help them identify the problem and gain knowledge more quickly by avoiding those same mistakes.

Closing

Build others up, share your knowledge, present at user groups and conferences, and improve others. As a senior developer, you should be helping make others into senior developers. It will help them, it will help you, and it will improve your company, your community, and your career. See you next month.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.



1 *July 2015. Leveling Up: How to Burn the Candle at Both Ends: https://www.phparch.com/magazine/2015-2/july/*

Gratitude

Cal Evans

WARNING: This is an Old Man Rant (OMR). Feel free to skip this.

Sometimes I think that we as a community–not just the PHP community but also most web developers and possibly society as a whole–have lost the ability to be grateful. People do something nice for us and instead of simply saying thank you, we point out publicly that they did not do it in the right way. Let me give you an example. It's a fairly specific one but serves to make my point.

Recently, but far enough in the past that most of you won't remember, I watched this scenario play out. Someone created a list of their favorite people in a community. In this particular case, they took the time to do some research into the people and even had nice caricatures of each person created for the article. On the day the article was released, at least one person went on a Twitter rant. (A Twitter rant is defined as five or more tweets on a given subject in a row.) The author of the article was berated for creating such a divisive article, for setting some people up on pedestals, and even for daring to create caricatures of people without their express permission.

So let me get this straight. Some poor soul dared to call out twenty-five people as the ones that he or she looks up to the most, the ones that have been an inspiration to the author, twenty-five people who have had an impact on the author's career, and the best one of them can do is berate the author? Something is seriously messed up here.

In the past ten years of my career, I've been included on more of these lists than I can count. Even so, I've been left off more lists than I've been included on. Here's something that a lot of people don't know. For the first twenty-five years of my career I wasn't included on a single one. That's right, for twenty-five years I struggled in anonymity.

For those twenty-five years, I struggled, I coded, I read and researched, and when I came across a list of people that an author took the time to put together, I read it with interest and looked to see if there were people I should be listening to; sometimes there were, sometimes not. I was grateful to the people on the list, but also to the author for taking the time to put together the list. After email became common, I would write the author and thank him or her. (Yes, my career started before email was common[1]. I told you this was an OLD man rant.)

These days, when it comes to PHP, yes, I am on a lot of these lists. Every time someone takes the time to list me in a Top 10/Top 25/Top X list, I take the time to thank them. It doesn't matter what I think of the list or the author, they took the time to single me out and lift me up to others as someone who has influenced them. Gratitude is the order of the day.

I don't berate them. I don't publicly call them out for dividing the community. I don't explain to them that every developer is worthy of praise. I simply say thank you. Depending on the list, I also may help the author promote it so that the other people on the list get additional promotion.

It is sad to me that we as a community have lost the ability to say thank you. Even if you are the one being added to the list, sometimes it's not about you. Sometimes it's not about how you think things should be done. Sometimes you just need to swallow your angst over how this list will divide the community—it won't—and just say thank you.

Saying thank you to the author of the list and being gracious about the praise that is being heaped on you will say more about you and your character than being listed to begin with. Of course, your gratitude shouldn't be limited to lists of notables. Show gratitude to that coworker who helped you find the cause of that tricky bug, thank a developer who writes a library/module/plugin that you use every day, and tell your local user group organizer that you appreciate the work he or she does.

The rant is over now. By the way, recently my good friend Ms. Samantha Quiñones was interviewed by Cloudways hosting. In it she said some very nice things about me. I thanked Samantha privately when the interview came out. However, please allow me to take this chance to thank her publicly for her kind words. I am privileged to have friends like Samantha.

1 *Editors note: Ray Tomlinson is credited with inventing email and using the @ symbol as part of the address in 1972. Email became popular in the mid 1990s with the rise of AOL, Prodigy, Compuserve, and—later—Hotmail.*

Past Events

May

Lone Star PHP
May 7–9, Dallas, TX
http://lonestarphp.com

SymfonyLive Paris 2016
May 7–8, Cologne, Germany
http://paris2016.live.symfony.com

Joomladagen 2016
Mays 15–17, Zeist, Netherlands
http://joomladagen.nl

SymfonyLive Cologne 2016
May 27–29, Cologne, Germany
http://cologne2016.live.symfony.com

Upcoming Events

May

New Orleans DrupalCon
May 9–13, New Orleans, LA
https://events.drupal.org/neworleans2016

Meet Magento—Netherlands
May 12–13, Utrecht, Netherlands
https://www.meet-magento.nl

phpDay 2016
May 12–14, Verona, Italy
http://2016.phpday.it

PHPKonf
May 21–22, Istanbul, Turkey
http://phpkonf.org

WordCamp Minneapolis 2016
May 20–22, Minneapolis, MN
http://2016.minneapolis.wordcamp.org

php[tek]
May 23–27, St. Louis, MO
http://tek.phparch.com

CakeFest 2016
May 26-29, Amsterdam, Netherlands
http://cakefest.org

International PHP Conference 2016
May 29–June 2, Berlin, Germany
https://phpconference.com

PHPSerbia Conference 2016
May 28–29, Belgrade, Serbia
http://conf2016.phpsrbija.rs

June

PHP South Coast 2016
June 10–11, Portshmouth, UK
https://cfp.phpsouthcoast.co.uk

Dutch PHP Conference 2016
June 23–25, Amsterdam, The Netherlands
http://www.phpconference.nl

July

php[cruise]
July 17–24, Baltimore, MD (leaving from)
https://cruise.phparch.com

LaraCon US
July 27–29, Louisville, KY
http://laracon.us

August

Northeast PHP
August 4–5, Charlottetown, Prince Edward Island, Canada
http://2016.northeastphp.org

October

Bulgaria PHP 2016
October 7–9, Sofia, Bulgaria
http://bgphp.org

LoopConf
October 5–7, Ft. Lauderdale, FL
https://loopconf.com

Brno PHP Conference 2016
October 15, Brno, Czech Republic
https://www.brnophp.cz/conference-2016

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

Things We Sponsor

Developers Hangout
Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

NEPHP
NorthEast PHP & UX Conference. August 4–5, 2016
2016.northeastphp.org

NomadPHP
Start a habit of Continuous Learning. Check out the talks lined up for this month.
nomadphp.com

DC PHP
The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

FredWebTech
The Frederick Web Technology Group
meetup.com/FredWebTech

Keeping Credentials Safe

Chris Cornutt

One of the foundations of secure systems are the pieces of information they use to authenticate the user or client on the other end of the line. Along with the huge amount of authentication systems out there comes a wide range of potential credential types. These are the "keys to the kingdom" of your application, and your users trust you to protect them with the highest level of security and prevent them from falling into the wrong hands.

Examples of these kinds of credentials range from simple to much more complex, including:

- username and password combinations
- two-factor authentication device identifiers
- federated identity connection details
- tokens (like OAuth)

While some of these are more common than others, there's a huge concern around storing them safely.

Goals

One of the main goals behind securing the credentials stored in your system it to protect them from the prying eyes of the outside world. You want them to be secure, not only in how they're stored but also in a method that protects them were they to be leaked. Let's take a look at these two main goals a bit more in depth.

First off, let's tackle the "at-rest" problem. Let me start off by saying that if you're storing any kind of credential information in plain text in your system, you need to act immediately to fix this. While it's tempting to store something like a device identifier as plain text for ease of use, were this piece of information to be leaked somehow (for example, through a SQL injection attack) the attacker could use that to spoof a device to your system.

Another hallmark of bad credential storage is hardcoding the unprotected values. As developers, it's way too easy to just think that "what's in the code stays in the code" with PHP. After all, it's processed on the server and only the resulting output is ever sent to the user, right? Unfortunately, all it takes is one simple local file include vulnerability and they could harvest the entire contents of your application's code…complete with that plain text hardcoded credential. You can see how this could be a bad thing!

> *Just after I submitted this article, detectify[1] reported finding Slack chat tokens in GiuHub "public repositories, support tickets, and public gists." Because developers included the tokens directly in code, and then committed to a public repository, any malicious user who found the tokens could use them to access internal conversations.*

So, we need to be sure that we're protecting this information as it sits in either your data source or code correctly. Keep in mind that what I'm talking about here is security-sensitive information, not necessarily all user information. While you can protect all of that other information in a similar way, it's not the topic of this article. I'm just focusing on key and credential management.

To simplify things, I'm going to use one of the most common credential examples—the username-and-password combo—to illustrate my points. First up: hashing.

Hashing

While *technically* hashing is a form of cryptography, it's not normally considered on the same level when people say they're "encrypting the data" they're using. Generally with cryptography, your goal is to be able to reverse, or decrypt, some data. With hashing, the intent is one-way only. Hashing essentially takes the string

1 *Detectify: Slack bot token leakage:* http://phpa.me/detectify-slack-tokens

value you've provided and uses a consistent algorithm to produce a string of characters that represent the original. In most of the widely used hashing types, this resulting string is made up of numbers and letters and is the same no matter how many times you hash the same value.

I mentioned that the goal of hashing is one-way. Hashing a value allows you to protect it from prying eyes at rest as it's not in a format that's readily readable. I will give one word of caution, however—despite the methods and intent of them being one-way, they can be reversed given enough time and computing power. This is why it's important to choose a hashing method that's going to make life difficult for any would-be attacker should they gain access to your data store's contents and the hashes themselves.

One of the most robust and popular hashing methods that's been made available to PHP recently is bcrypt. This hashing method is interesting in how it generates the resulting string of letters/numbers representing the value. Hashing methods like MD5 and SHA1, which are considered weaker, are fast, but that's because they only run their algorithm once and spit back a result. Bcrypt is different, though. With bcrypt, there's a "cost" factor that tells the hashing method how many times to run through the algorithm before returning a result.

> *One major note here: DO NOT use MD5 or SHA1 one for anything anymore. There are two reasons here: one is that the hashing methods for these two have made it easy to pre-compute the plain text-to-hash matches (commonly called "rainbow tables") and second, they're so fast to compute that the average md5 hash can be broken in less than a second and only a minute or two more for sha1.*

While you can use PHP's `crypt()` functionality to manually bcrypt strings, there's a much easier way: the password hash handling functions[2]. A while back, the PHP core developers decided that PHP needed a drop-dead simple way to safely protect passwords, as they're one of the most common pieces of credential information in PHP applications (really, in most web applications). A password-handling API was proposed as an RFC, accepted into the core, and added as of PHP 5.5.0. The functionality makes creating and verifying the hash easy. Here's an example of both:

```
<?php
// To create the hash
$password = $_POST['password'];
$hash = password_hash($password, PASSWORD_DEFAULT);

// To verify the hash
if (password_verify($_POST['password'], $fromDbHash)) {
   echo 'Valid password!';
}
```

In the first example in the code above, it uses the `password_hash` function to generate the resulting hash (simple, right?). The `PASSWORD_DEFAULT` constant is required and, for the time being, points to the bcrypt hashing method. In the future, if bcrypt was ever broken, this default could be updated to something else. In the second part of the example, the code is verifying the password the user submitted against the hashed value pulled from the data source (`$fromDbHash`). The `password_verify` function always returns a boolean for easy evaluation.

If you review the documentation for the password hashing functionality, you'll notice two options you can set: the cost and a custom salt. If you're not familiar with the term, a "salt" is just a unique value appended to the original value to help add entropy (randomness) to the resulting hash. With bcrypt, this value is a part of the resulting hash itself as it's not considered "secret." It is *highly* recommended not to provide this salt yourself, however, and to let the password hashing function do it for you, since it's built to generate a secure salt automatically for you. The other option is the "cost." I've mentioned this already as the number of times the bcrypt hashing is performed before returning a value. A word of warning here: while a higher cost sounds like a good idea, keep in mind that along with a higher cost comes more processing time. It might be longer than you want your users to wait.

Encryption

The next step up in credential protection is encrypting the original value using a predetermined algorithm and a matching key value. Encryption is a really deep topic and you could spend a really long time trying to determine the best algorithm and mode you need to use for your data. I'm going to give an example here but this is by no means to be considered a hard and fast "this is super secure!" kind of example. This is just one of the many options available to PHP users right now. I'm going to make use of the `mcrypt` extension that's installed by default on most systems these days. If not, there's usually a package you can drop in that will add it in relatively easily.

When I was talking about hashing before, I pointed out that its intent is one-way only. Encryption instead takes a two-way stance, making it possible to reverse the value that's stored back to its original plain-text form. Encryption should only be used when this kind of process is needed. If you only need to take a value and compare the results (like the password example above), hashing is a much better way to go.

Listing 1 is an example of what encrypting a simple string looks like.

If you're not familiar with some of the terminology involved in encryption, let me give you a brief primer:

- The "algorithm" (in this case Rijndael 256) is a mathematical formula that is used to translate the value provided into cipher text.
- The "mode" (in the example it's "CBC") is an indicator of how the formula was applied in the transformation of the original value.

LISTING 1

```
01. <?php
02. $source = 'sup3r-53cr3t-p4ssw0rd';
03.
04. $key = file_get_contents('/path/to/keyfile');
05. $ivSize = mcrypt_get_iv_size(MCRYPT_RIJNDAEL_256, MCRYPT_MODE_CBC);
06. $iv = mcrypt_create_iv($ivSize, MCRYPT_DEV_URANDOM);
07.
08. // Encrypt the string
09. $result = mcrypt_encrypt(MCRYPT_RIJNDAEL_256, $key, $source, MCRYPT_MODE_CBC);
10.
11. // Decrypt the string
12. $iv = substr($result, 0, $ivSize);
13. $original = mcrypt_decrypt(MCRYPT_RIJNDAEL_256, $key, $result, MCRYPT_MODE_CBC, $iv);
```

2 *php.net: Password Hashing:* http://php.net/password

- An "initialization vector" (the "IV") is a randomized string provided to the algorithm as a requirement for its transformation of the data.

In the example above, we combine these three things with a "key" value tailored to the algorithm we've chosen to generate an encrypted version of the original. This key is just a plain text string that meets requirements for the algorithm, like length and complexity, for it to function at its best. If a key doesn't meet the requirements for the chosen method, it will usually be padded out until it does.

Also, let me include a reminder here: *this "key" value is one of the most valuable pieces of the whole encryption process.* Without it an attacker has to work *very* hard to reverse the encrypted version of the original. You have to keep this value safe! Never, ever hardcode it anywhere in your code, and keep the file containing its value outside of the document root with permissions so that only the web server user can read it.

Third-Party Services/Software

Finally, I do want to quickly mention another option that's been growing in popularity in recent years: passing off the credential management and storage to tools or services that are specifically designed for this task. There are two main kinds of integrations that can happen here, and if you choose to go this route you'll need to determine which is a better fit. You can either use some kind of third-party software (like Vault) to store secrets in or you can completely offload the credential management to other services. This offloading usually comes with connections to social media (how many times have you seen a "Log me in with Twitter" button?) where the user's details are actually stored.

The main advantage to these options is that you don't have to worry about storing the credentials yourself, reducing your overall risk level. Unfortunately, you're also adding a different kind of risk around the use of these tools and services, since you're trusting the service to keep your credentials safe. As mentioned, you just have to sit down and see what level of risk you're willing to accept and if using these methods makes sense.

Credential storage can be one of the most challenging tasks in the overall scheme of an authentication system. Hopefully I've given you a good overview of some of the methods that are currently out there and something you can start applying to your systems today. If you haven't already done so, I'd highly suggest an audit of your current credential storage methods just to see where you stand. Then, if you need to make improvements, you have some good suggestions for where to start.

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce.

April Happenings

PHP Releases

- PHP 7.0.6: http://php.net/archive/2016.php#id2016-04-29-1
- PHP 5.6.21: http://php.net/archive/2016.php#id2016-04-28-2
- PHP 5.5.35: http://php.net/archive/2016.php#id2016-04-28-1

News

Jordi Boggiano: Composer Goes Gold

Jordi Boggiano has posted some excellent news for all of the Composer users out there—the widely popular dependency management tool has officially "gone gold" and has tagged the stable v1.0.0 version of the tool. Jordi talks about one big change that happened recently around the "self-update" feature of the tool.
http://phpdeveloper.org/news/23851

Zeev Suraski: PHP 7 is Gaining Momentum

In his new post to his site Zeev Suraski talks about the momentum growing behind PHP 7 and some of the recent articles about companies making the move and the overall impression of the new version. He gets into a bit more detail about the numbers that Badoo published and gives a quick "thank you" to Dmitry Stogov for helping to spearhead the effort to get PHP 7 out the door from Zend.
http://phpdeveloper.org/news/23842

Matthew Weier O'Phinney: On Deprecating ServiceLocatorAware

In this post to his site Matthew Weier O'Phinney talks about a change in the Zend Framework `ZendMVC` package to deprecate the `ServiceLocatorAware` interface and some of the unexpected backlash of it. He shares some of the "constructive" feedback they received when they made the change, most importantly asking for a justification for the change.
http://phpdeveloper.org/news/23934

Phillip Shipley: Docker Makes Upgrading to PHP7 Easy

In this post to his site Phillip Shipley talks about Docker and how using it for your PHP deployments can make it much easier to upgrade to PHP 7. He gets into a bit of detail about how upgrade process and some of the smaller issue he faced along the way. He also includes the update to his Dockerfile he made to change to PHP 7 (only a few characters) to rebuild with PHP 7.0.4.
http://phpdeveloper.org/news/23922

Leonid Mamchenkov: Adventure in Composer Private Repositories

In this new post to his site Leonid Mamchenkov talks about some of his "adventure with Composer private repositories" in some of his deployment work with CakePHP 3 applications. Unfortunately he was getting a `RuntimeException` when he was trying to pull in a plugin through the same private repository workflow. As it turns out, it was the `composer.json` of the main application repository that was the problem. He includes the fix he made to the configuration on a sample CakePHP 3 project, showing how to switch it to a "vcs" type for more correct handling.
http://phpdeveloper.org/news/23916

Alex Bilbie: OAuth 2.0 Device Flow Grant

In a new post to his site Alex Bilbie looks at a good approach to simplifying the OAuth 2 authorization flow for a device and some of the simple PHP that can power it. He talks more about this better experience involving a simple code presented to the user, a special URL to link the device and the typical OAuth-ish authorization page to link the request to your account. He then explains how it would work with a PHP backend.
http://phpdeveloper.org/news/23908

Adam Culp: Using an SSH Tunnel to Step Debug Through a Firewall

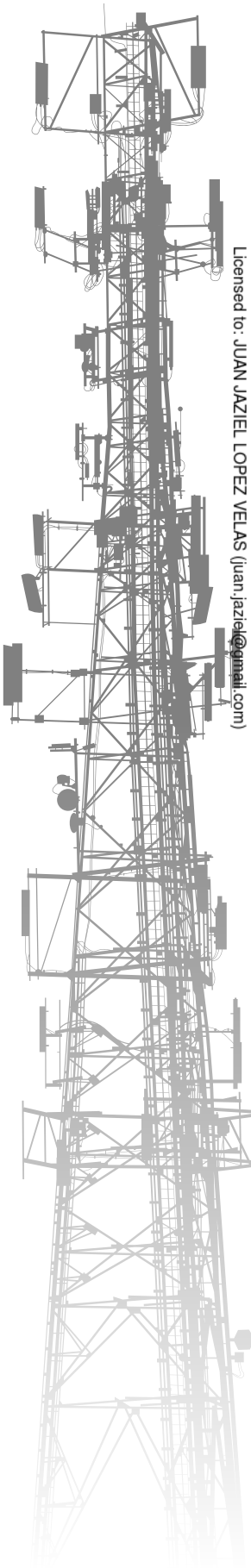
Adam Culp has followed up some of his previous posts about the setup and configuration of remote debugging in PHP applications (more specifically in Zend Studio) with this new post. In it he shows how to use a SSH tunnel to allow debugging to happen through a firewall for those cases when direct access isn't possible.
http://phpdeveloper.org/news/23900

Jeff Geerling: Streaming PHP—Disabling Output Buffering in PHP, Apache, Nginx, and Varnish

In a recent post to his site Jeff Geerling shows you how to disable the output buffering that PHP includes and create "streaming PHP" code similar to Drupal's recently introduced BigPipe handling. He decided to try out different configurations to see if he could reproduce the same thing outside of Drupal and - good news, everyone - he found a reliable way.
http://phpdeveloper.org/news/23856

Jesse Schutt: Simplifying Conditional Expressions

Jesse Schutt has posted a set of helpful hints around simplifying conditional expressions in your PHP code. This can not only make them more readable but also easier to maintain in the future. He touches n a few different types of conditional refactoring and provides examples for each.
http://phpdeveloper.org/news/23845



Hindsight & Planning

Eli White

Hindsight is 20/20, or so they say, but can't we do better at planning where we are supposed to be in the future? It's a complicated topic that I can't possibly manage to do justice to in just a page here, but let me touch on the subject, at least.

Making Plans

The famous Chinese philosopher and general Sun Tzu once said: "No plan survives first contact with the enemy." This is an extremely accurate statement, whether in software, the entrepreneurial world, or in actual combat. (Spoken by someone who has experienced all three.)

So why do we make plans if we know they will always fall apart? What's the point if plans are doomed to fail? The fact is that you cannot go through life with *no* plan at all. Answering "Where do you want your company to be in five years?" with "I don't know, ask me in five years" leaves you without goals, without direction, and without guidance as how to move forward.

In the end, having the plan isn't actually what's important—it's in the process of creating a plan where the value is born. The discussions that you will have, whether as a family, as a company, or as an employee, about the future will require you to explore many alternatives and discuss the pros and cons of each one.

It's these discussions that are of value when the #$@% hits the fan, and your plan completely falls apart. You (and more importantly, your whole team) can fall back on the knowledge and insights gained during the planning process, and quickly adapt to the right step to take to carry on.

Learning from Your Hindsight

Because of all this, it means that quite often you will look back at something you did and go, "Why did we ever think that was a good idea?" Typically, though, if you can re-evaluate your thought process at the time, you will find that it was made for a good reason, because at the time, it was the best option you had.

That's really all you can ask for—to make the best decision that you can at a given moment. But you need to always be looking back and evaluating how things have gone. You learn from your past choices that have been shown to now be mistakes in the present, in order to make better plans for the future.

You can't predict everything that might happen. Life will throw you, your company, and your project a curveball from time to time. The important thing is to learn from what happened, find an option to move forward, and carry on.

Applying this to the PHP Project

While this article wasn't meant to actually be about PHP specifically, you can easily take a look at the PHP Project as exactly a case of this. There are lots of things in the PHP language that, in hindsight, could have been designed better in the beginning. But they were the best options that the creators of the language had to work with at the time.

Those old decisions, those warts of the language, are part of what PHP is now. The fact that PHP has persevered over the years, and has continued to grow and become the dominant language for producing web applications at the moment, is all due to those early decisions.

So as we move forward as a language and as a community, let's not complain about the decisions that were made in the past. Let's embrace them, learn from them, and let's continue to make future plans to grow the language, knowing that there will be another curveball waiting for us right around the bend.

Eli White is the Managing Editor & Conference Chair for php[architect] and a Founding Partner of musketeers.me, LLC (php[architect]'s parent company). He constantly plans, but never quite finds himself where he thought he was going. @EliW

PHP SWAG

PHP Drinkware

PHPye Shirts

Laravel and PHPWomen Plush ElePHPants

php[architect]

Visit our Swag Store where you can buy your own plush friend or other PHP branded gear for yourself.

As always, we offer free shipping to anyone in the USA, and the cheapest shipping costs possible to the rest of the world.

Get yours today!

www.phparch.com/swag

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