www.phparch.com

June 2016
VOLUME 15 - Issue 6

php[architect]

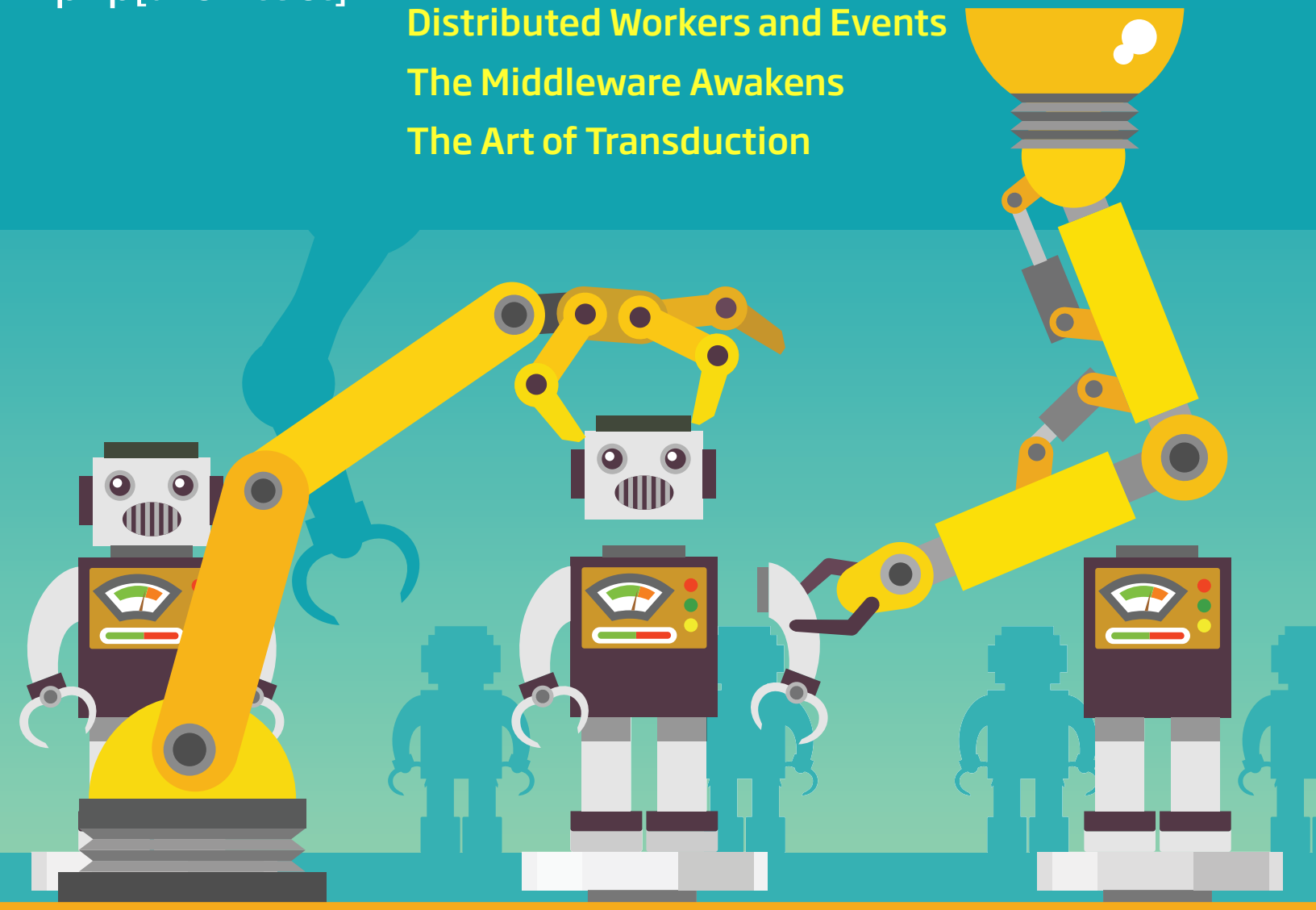
decoupled by design

Security Architecture: Securing your Web Services, Part Two

Distributed Workers and Events

The Middleware Awakens

The Art of Transduction



ALSO INSIDE

Professional Development for Professional Developers

MySQL's JSON Data Type

Community Corner:
PHPeople

Education Station:
Easy Audio and Video Manipulation with FFmpeg

finally{}:
Ups and Downs of an Entrepreneur

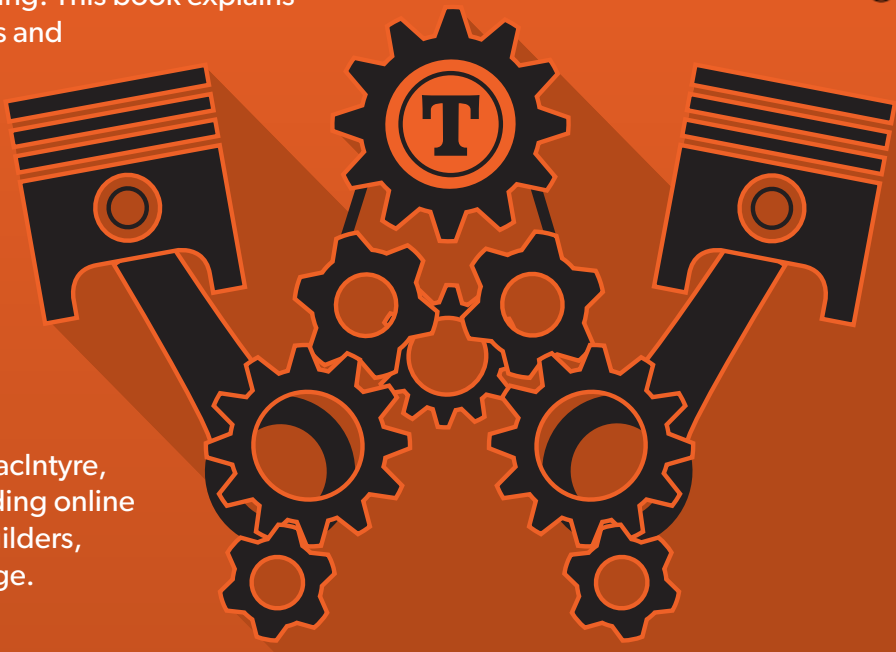
Building Exceptional Sites with WordPress & Thesis

by Peter MacIntyre

Need to build customized, secure, search-engine-friendly sites with advanced features quickly and easily? Learn how with this guide to WordPress and the Thesis theme.

WordPress is more than a blogging platform—it powers one-fifth of the Web. Since its release, enthusiastic contributors have pushed the envelope in using WordPress as a platform to build social networks, e-commerce storefronts, and more. However, the ecosystem of plugins and themes is vast and can be overwhelming to sift through, especially if you're new to it.

Building Exceptional Sites with WordPress & Thesis is a practical guide to using WordPress with the Thesis theme to provide an flexible and customizable foundation for building a wide variety of websites without requiring much—or any—programming. This book explains how to customize pages using Thesis and provides a survey of must-have plugins to help your website track and understand your website traffic analytics, harden your site's security and defenses against hackers, improve Search Engine Optimization (SEO), stay in communication with your users via email, handle e-commerce, offer tiered access to your site, and more.



Written by PHP professional Peter MacIntyre, this book distills his experience building online solutions for other WordPress site builders, developers, and designers to leverage.

Purchase Book

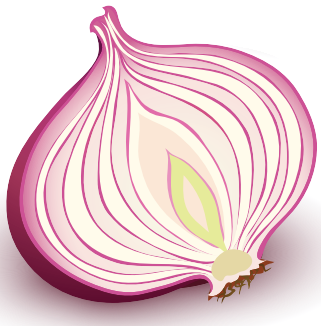
http://phpa.me/wpthesis-book

php[architect]

JUNE 2016
Volume 15 - Issue 6

CONTENTS

decoupled by design



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

Decoupled by Design

The fourth php[tek] that we've organized concluded yesterday. Now, I find myself in the St. Louis terminal waiting for my flight home. I also find myself reflecting on the keynotes. One common thread that weaves through Jeremy Mikola's, Samantha Quiñones', and Cal Evans' presentations is the value and importance of the people that make up the greater PHP community by what they contribute to build it. These contributions range from contributing to core, to maintaining libraries and frameworks, to running a user group or conference.

This takes significant effort and investment by the people who give back. We have to be careful that they do not burn out. In addition to gratitude, as Cal wrote in last month's issue, we should also be looking to invigorate the PHP community. New faces and perspectives are always welcome. I urge you to be on the lookout for ways to bring new people to it. Start by getting your co-workers to your next UG meetup. If there's a conference nearby, convince your boss to send your team.

This month we look at tools to help you decouple your code and applications. Christopher Pitt will help you use *Distributed Workers and Events* to make your PHP code asynchronous. He looks at various options for making code run outside of a single thread. If you're looking to implement PSR-7 in your applications request & response cycle, in *The Middleware Awakens* Ian Littman looks at a common implementation that allows for middleware layers. Middleware allows you to compose a chain of independent components to handle authentication, IP logging, and more as you build your app's response. Since many modern day applications are returning JSON data to one or more clients, Dave Stokes looks at how you can store, index, and retrieve JSON natively with MySQL 5.7 in *MySQL's JSON Data Type*. He'll show you how to get started and the tricks and limitations you'll need to know. If you're decoupled application is talking to or providing one or more APIs, don't miss *Security Architecture: Securing your Web Services, Part Two* by Edward Barnard. In this part, he has advice for an effective web services security approach.

Also in this issue, Steve Grunwell writes about *Professional Development for Professional Developers*. His solid advice will help you stay up to date with new technologies without burning out. In *Education Station*, Matthew Setter takes a look at *Easy Audio and Video Manipulation with FFmpeg*. If you need to manipulate and automate working with audio and video files, don't miss it. David Stockton teaches you how to uses Transducers in *The Art of Transduction* in *Leveling Up*. If you're looking for alternative to `foreach` for processing arrays, you'll find them here. In *Community Corner*, Cal Evans asked Amanda Folson to write about why she likes the PHPeople in the PHP community. She shares what draws her to it. Last, Eli White shares *Ups and Downs of an Entrepreneur* that he's experienced and why its worthwhile in *finally*.

Errata *In last month's article on OAuth 2.0 by Ben Ramsey, we used an endpoint that is only available to older applications. If you registered a new application, the endpoint for retrieving a user's feed is different. Ben has written about how to correct it on his blog: http://bram.se/phparch-oauth2.*

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!

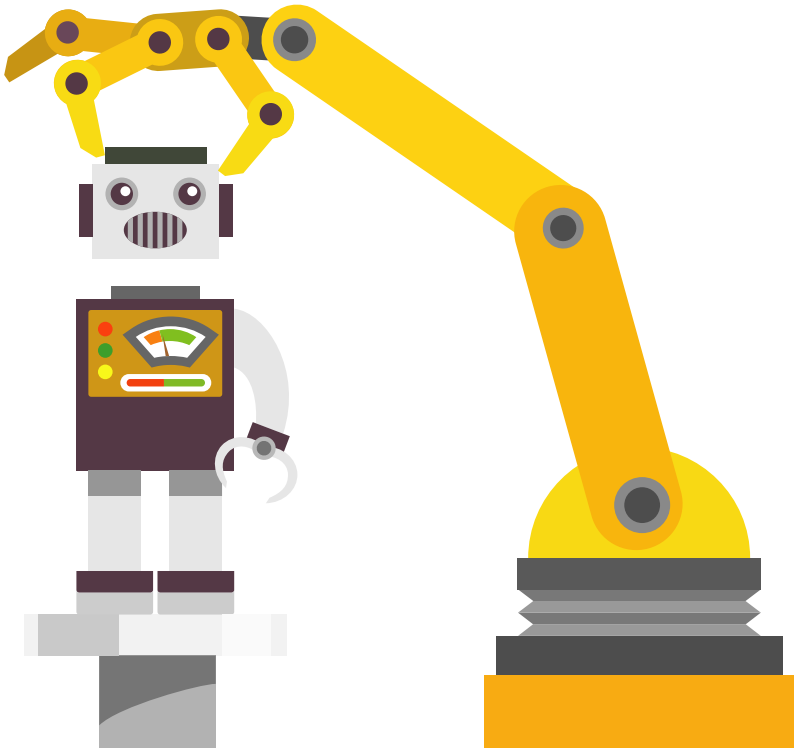
Stay in Touch

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

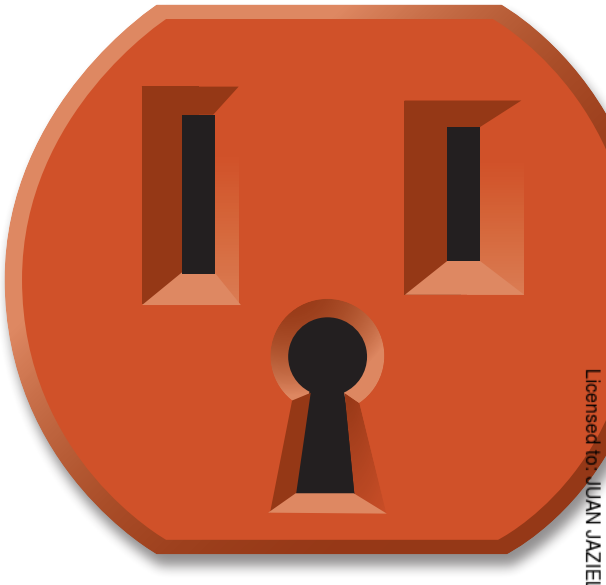
- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this issue's code package:

http://phpa.me/June2016_code



Security Architecture: Securing your Web Services, Part Two



Edward Barnard

Any web service security architecture should be a combination of standard practice and applying the lessons learned from your attackers. You don't look for perfection. Instead focus on "raising the bar" high enough that there's too much effort needed for the possible gain. Identify your weakest links. Provide the flexibility for further hardening of your web services in the future should attacks show that this is needed. We'll examine the reasons for each decision.

Web Service Security

Part One contained this advice 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.*

Part One, *Learn from the Enemy,*[1] took that first sentence to heart. We don't have the details yet, but we understand that web service development requires us to come from a different perspective.

It's time to "level up." We now focus on that second sentence, *You will learn to defeat the enemy.*

You've heard about *Authentication* and *Authorization* before. I get that! This time, though, we'll see why Authentication and Authorization do **not** work when it comes to web services.

Each of these bits of "learning to defeat the enemy" are to explain what does not work, why it does not work, and what we need to know to do something about it.

Authentication

Authentication[2] is the act of identifying yourself and proving that you are who or what you claim to be.

With your web services, it's natural to assume that if your URL was reached by a correctly formatted request, it came from your app. The basic problem is that there is really no absolute way to prove that it *did* come from your app, and there is therefore no real way to know that it did *not* come from your app. How can this be?

The basic problem is that your app is released into the wild. Anyone can install the app. It could be installed on a jail-broken iPhone, for example. Anything the app can do, an attacker can fake or mimic. Your attacker can record any sequence of requests and responses and play them back later (a replay attack).

From the server's perspective, all you know is that someone reached your server with a correctly formatted request. You can't, for example, restrict the traffic to a "safe" internal network. If your app is out in the wild, you have no choice but to allow traffic to come in from the wild. You have absolutely no way to verify independently that the client is who it claims to be. Authentication, in the larger sense, is simply not possible.

Authentication is certainly *theoretically* possible. For example, to access company email from home, you might have installed a security certificate on your home computer. That certificate came from your employer. Therefore, when you make the connection and present that employer-provided certificate, your employer knows that you are you.

This is the *Mutual Authentication* protocol[3]. The client (your email program from home) presents a certificate to the company server, and the company server presents a certificate to your client. Each side has an independent means of verifying the integrity and authenticity of the other side's certificate.

Another possibility is that you reach your company's server via VPN or an SSH tunnel. You have the login credentials and therefore you are who you say you are.

To be sure, certificates can be stolen and passwords lost. That's outside this discussion.

How does this work with an app downloaded from the App Store? It's appropriate to hand out a security certificate to company employees. There's a reasonable level of trust involved.

On the other hand, it is *not* reasonable to hand "the keys to the kingdom" to every unknown person who downloads and installs your app. In other words, *Mutual Authentication* is not available to you.

1 *May 2016 php[architect]:* https://www.phparch.com/magazine/2016-2/may/

2 *Authentication:* https://en.wikipedia.org/wiki/Authentication

3 *Mutual Authentication protocol:* https://en.wikipedia.org/wiki/Mutual_authentication

Authorization

The authorization[4] process asks *Shall I allow you to use this web service?* For example, if you're not logged in, you should not be able to see account information. You should not be allowed to change the account's password unless you prove you know the account's current password. And so on.

Web services are stateless. That means that granting permission is based on the *current* web service. We likely see the following sequence:

1. The app (on behalf of the user) authenticates by providing the valid user name and password.
2. The server responds to the app with a token.
3. The app includes that token with all future web service requests. The token provides some level of *authorization* by virtue of prior *authentication*.

Information Leak

Suppose that you, as the attacker, observe these different web service error messages:

- That user name does not exist in our system.
- The password you provided does not match that user name.
- Your account is no longer active. Please visit our FAQ page for more information.
- We did not recognize your session token.
- An active session token is required.

Can you see that, with these helpful responses, you are teaching your attacker how to hack you? It's like the guessing game "hot and cold." If your guesses are getting closer to the correct answer, you are getting "hotter," and if you're on the wrong track you're getting "colder." Don't guide the attacker in solving your security system!

4 Authorization: https://en.wikipedia.org/wiki/Authorization

It's possible to respond with zero information while distinguishing the response in your server logs. For example, when you detect a brute-force login attempt with your web services, you could respond with HTTP status 418 (I'm a Teapot) and an empty (zero bytes) response body.

Your attack response needs to be automatic. Otherwise it could be "game over" before you realize battle was joined. In this case you can trigger alerts based on the 418 status code appearing in your server log. The 418 code means your deflector shields[5] are up.

You might not choose to have your server become a teapot. The point is that your normal firewall protections may not work with web services. That's because your app looks like a bot, and you *must* allow your legitimate app traffic to come in. You therefore need some way of notifying *your* humans that you're under attack.

Reuse and Replay

If the server provides some sort of login or authorization token, that token can be hijacked. You trust your server, sure, but you *know* that you cannot trust the client side of your traffic. Your traffic can be decrypted and observed with Fiddler or similar tools. Your attacker can carefully craft web service requests for the purpose of probing weaknesses in your API. Chances are that all it takes is a hijacked token.

Guessable Secrets

A hijacked token only gets you so far.

- If the token expires, then it's only good for that duration. Your attacker will need to "harvest" a fresh token from time to time.
- The token is only good for that one user login. Chances are that anyone can sign up for a new account. An attacker can sign up, use Fiddler to observe a valid login token, and use it. But at that point your attacker is only attacking his or her own account.

On the other hand, what if the token is guessable? If your attacker can generate valid tokens at will, you have a problem. The next article in this series, *Implementing Cryptography*, will have examples.

OAuth 2.0

I have heard the advice, "Implement OAuth 2. It takes care of the security."

The OAuth 2.0 Authorization Framework[6] is a good thing. The problem is that it does not solve the problem that we need to solve! The standard explains itself:

> *The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service…*

So far, so good. That does sound like what we need. Bear in mind what we've already learned, though. Do you see the catch? OAuth 2.0 rightly identifies itself as an *Authorization* framework. It does not solve our *Authentication* problem.

What problem are we trying to solve? You can't distinguish an attacker from your own app. You know this because your attacker has successfully pretended to *be* your app.

5 deflector shields: http://phpa.me/deflector-shields

6 The OAuth 2.0 Authorization Framework: https://tools.ietf.org/html/rfc6749

The OAuth 2.0 standard confirms you have a problem. Its Section 2.3, Client Authentication, notes that:

> *"the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client."*

In other words, if your app is out there in the wild, you absolutely cannot trust that you are talking to that app rather than an impostor.

How about logging in with a user name and password? Section 2.3.1, Client Password, notes the following:

> *Since this client authentication method involves a password, the authorization server MUST protect any endpoint utilizing it against brute force attacks.*

What can we conclude from all this? That you're on your own. OAuth 2.0 does have a place in the ecosystem, but we've made no forward progress. OAuth 2.0 specifically mandates that we solve these problems that we've not yet solved.

Your Security Architecture

Here is an effective web services security approach:

- "Raise the bar" enough that it's simply not worth it. You probably don't need the security level of a bank or government classified operation (unless you are one). "Good enough" is indeed good enough but do recognize that your enemy may have more to teach you.
- Focus your hardening efforts on the "front door." Make it really tough to authenticate via your web services.
- Use authentication tokens similar to the OAuth 2.0 standard. Make sure they expire quickly, and make them tough to renew. You might implement your own OAuth 2.0 server but carefully consider whether it in fact solves problems that you need to solve.
- Your app can almost certainly be reverse engineered. Make it as difficult as you can to do so. This matters because you likely store secret keys in the app.
- Change your app secret keys with every app release.
- Use certificate pinning[7]. This does not keep an attacker from attacking you, but it does keep an attacker from observing the web services via Fiddler. That strongly increases the degree of difficulty in attacking you.

Authentication and Authorization

Handle the login request and response with an encrypted payload. That goes against "standard practice" in that you should not need to do this. This is one of the places where *I* learned from the enemy. When your login credentials are in plain text (even across HTTPS), your enemy will find a way to read them.

If you're doing this sort of encryption, you need to embed a secret encryption key in your app. That's a problem. *Cryptography Engineering* by Ferguson, Schneier, and Kohno state their most important lesson as follows:

> *A security system is only as strong as its weakest link.*

In fact they believe this principle is so important that they insist every reader of their book place a note card near their workspace with this reminder. My note card is in view as I type this. What is your weakest link? Rest assured your enemy will teach you. Plan for it.

Meanwhile, you have an encryption key embedded in every copy of your app in the App Store. Someone could extract that key from the app image. However, that requires a far higher level of expertise than running a plain-text password list against your login web service. That "raises the bar" enough that your attacker may simply stay away.

Can you do anything more to improve your situation? Yes you can.

You may have an Android version of your app. It's available from Google Play. You might also have an iOS version of your app available through iTunes. Both apps need embedded secret keys for encryption. Therefore ensure that both apps have *different* secret keys. Someone able to harvest the Android version's secret key won't automatically have the iOS version's secret key and vice versa. That's not really a significant obstacle. Someone capable of harvesting the one is probably also capable of harvesting the other.

As you do app development, you'll be releasing updated versions of each app to its respective App Store. Ensure that each version gets a new secret key for encryption. If your attacker can harvest the key from one app version, he or she can harvest it from the next version as well. So this would seem to be pointless. On the contrary, you have a new tool!

You now have the ability to expire or revoke compromised secret keys. If you decide that any secret key has been compromised, mark it as invalid on the server side. Reject any web services attempting to encrypt with the revoked secret key. Would this protect you from an attacker who totally owns your app the moment it's released? No. But it gives you protection from pretty much everything short of that. There is a continuous give and take of attack and defense.

When you release a new version of your app through the App Store, the majority of your active members likely adopt the new version within a few weeks. That means that you can "retire" old encryption keys relatively quickly. Your attacker does not *get* years of free use of that secret encryption key! Again, this does not solve everything, but it does raise the bar. It encourages your attacker to just go away.

Encrypted Login

You have likely been advised that it's not best practice to encrypt web service requests and responses yourself. "Use HTTPS. That provides transport security. That's what it's for." In fact, we gave that advice in part 1.

Your enemy will happily teach you otherwise. As an additional measure, your login web service request should consist of an encrypted string. The string, encrypted with the app's secret key, includes:

- User name
- Password
- Possibly other site-specific information not relevant to our discussion

7 *SSL Pinning for Increased App Security:* http://phpa.me/ssl-pinning

Your login web service response also consists of an encrypted string. The server encrypts the response with that app's secret key. The response includes the following:

- A token, which we call the session token. It identifies users and the fact that they are logged in to your site. This token has a short expiration time. This token is similar to the OAuth 2.0 authorization token.
- The renewal token, which is an encrypted string. The app is *not* able to decrypt this string; only the server has the decryption key for this string. This token is similar to the OAuth 2.0 renewal token.
- The user's internal ID or whatever uniquely identifies the user outside the login credentials. Your app may use this in various ways but what's relevant is that it becomes part of the token-renewal request.

Any web service, then, that requires a logged-in user includes the session token with the login request. Keep a saved copy of that same token in your database on the server side. If you successfully find the presented token, you know who the user is and deem the web service request authenticated.

Provide the app with different error responses:

- If the session token is missing, respond "Missing Required Parameters."
- If the session token is correct but expired, respond "Expired Session."
- If the session token is not correct, respond "Invalid Session."

True, you do leak some information to a potential attacker. You need to tell the app what action to take to get past the error:

- The app should never see "Missing Required Parameters." If the for-real app sees this, you likely have a bug to fix.
- For "Expired Session," the app presents the renewal token (described next) and, upon getting a new session token, retries the request.
- For "Invalid Session," the app presents the login screen, requiring the user to manually type in his or her password. The app *never* stores the user's password.

Renewal Token

Login/authentication sessions must expire. If someone is logged in for months at a time, it's easy to hijack the login token. In other words, it would be easy for someone's account to get hacked. On the other hand, you can't expect active users to type in their password every few minutes. They won't remain users for long!

You need the renewal token to last a long time. Suppose your user mostly uses your site from the app on her iPad but peeks in using her smartphone once a month or so. Her phone needs to be able to store that renewal token for a month and have the token still be usable.

Fortunately OAuth 2.0 already tells us how to do this (Section 1.5). Except that it doesn't. OAuth 2.0 gives us the process flow but leaves out the details.

Here is the process:

1. When logging in, the server responds with both the login (session) token and the renewal token.
2. When the login token expires, send the renewal token back to the server.
3. Assuming the renewal token is valid, the server responds with a new login token *and* a new renewal token.

Here is how to do it:

1. The login response contains both the session (login) token and a renewal token. They are contained within an encrypted string. That means that neither token is visible to anyone able to sniff the web service traffic. An attacker (or the app, for that matter) must decrypt the string using the app's secret key to obtain the session token and renewal token.
2. The session token is plain for all to see in all future web requests. It's plain to see, that is, if you're able to decrypt the HTTPS traffic, which is more difficult with certificate pinning. But if you can get to it, it's there to see in plain text.
3. The session token is therefore a random value with 128 bits of entropy. *Cryptography Engineering* gives 128 bits as the acceptable standard but also notes that it is *very* common to miscalculate how much entropy you really have.
4. The renewal token is returned to the server inside an encrypted string. The app encrypts the string using its secret key. This means that the renewal token is never ever seen "in the clear" while being transmitted in either direction. The app stores the renewal token as securely as it can manage. Since the renewal token is *only* transmitted inside an encrypted string using the app's secret key, you've made it as difficult as you can for an attacker to harvest.

What does the renewal token contain?

- The internal user ID for this user.
- The session table row ID for this user or whatever ensures internal consistency.
- The timestamp showing when this renewal token was generated and encrypted.

When we decrypt and validate the renewal token (on the server side with the secret key known only to the server), the user ID and table row ID need to be internally consistent for us to know this is a valid token. Of course the successful decryption has us starting with a high level of confidence.

When the app packages up and encrypts the renewal token for the token-renewal web service, the app includes the internal user ID that it saved (along with the renewal token) from that successful login "way back when." The app is unable to inspect the renewal token, and thus only the server can compare all the pieces to ensure everything matches as it should. We've now built in the ability to invalidate (revoke) any or all renewal tokens. Every renewal token, inside its encrypted string, has the timestamp showing when that token was generated.

Meanwhile, in the database, your token table has a "renewal invalid before" column with a timestamp. Suppose, for example, that your user logged in a month ago. The app obtained a login token (long since expired) and a renewal token (good for several months). The app should be able to get a fresh login token by correctly formatting and encrypting the renewal token.

Now suppose you determine that this user account has been compromised. You can set a timestamp of "now" in your database. This timestamp means "Do not accept any renewal token that was generated before this timestamp." The renewal token was generated a month ago and is therefore no longer a valid renewal token. Why not? Because you just rescinded it in your database.

The renewal token is an encrypted string that can only be decrypted by the server. It has integrity checking and the app can't mess with it. That encrypted string has a plain old timestamp inside, and it's that timestamp that lets you revoke any or all tokens.

Remember, your app is out there in the wild. The server can't reach out to every app on every device anywhere on the planet and blow away its renewal token. Instead you simply revoke the token on the server side.

Certificate Pinning

This is another measure you can take since you have control of your own servers, of course, and their security certificates. You hopefully also have control of your own app. This means that you can preload your security certificate (or public key) in your app. Then, when the app connects to your server, it can verify that the server's certificate is in fact the expected certificate (or public key). If it isn't, the app refuses further communication.

For a normal user, this means the app user can be confident that the app is in fact talking to the real server. See the OWASP article on Certificate and Public Key Pinning[8] for details.

Suppose you have an attacker monitoring the HTTPS traffic via Fiddler. Fiddler acts as an HTTPS proxy. The app talks to Fiddler, and Fiddler talks to your server. Any client (including your app) has the option of validating or verifying the server certificate or public key upon first connection. Your app can determine that it's not talking directly to your server and refuse further communication.

This article is about protecting the server-side web services, but certificate pinning is about protecting the app. How does this help us on the server side?

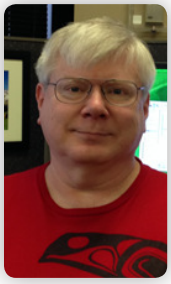
The way to successfully use your web services is to mimic your app. Private RESTful web services generally don't publish instructions on how to use those web services. An attacker learns to use your web services by observing how the app uses your web services. With certificate pinning, you make it far more difficult for an attacker to watch what's happening under the covers.

Security Implementation

You have now created your web services security architecture based primarily on hardening your login process and certificate pinning. You haven't seen any code. The companion article *Security Implementation* shows how to generate your session tokens and how to encrypt and decrypt your login data.

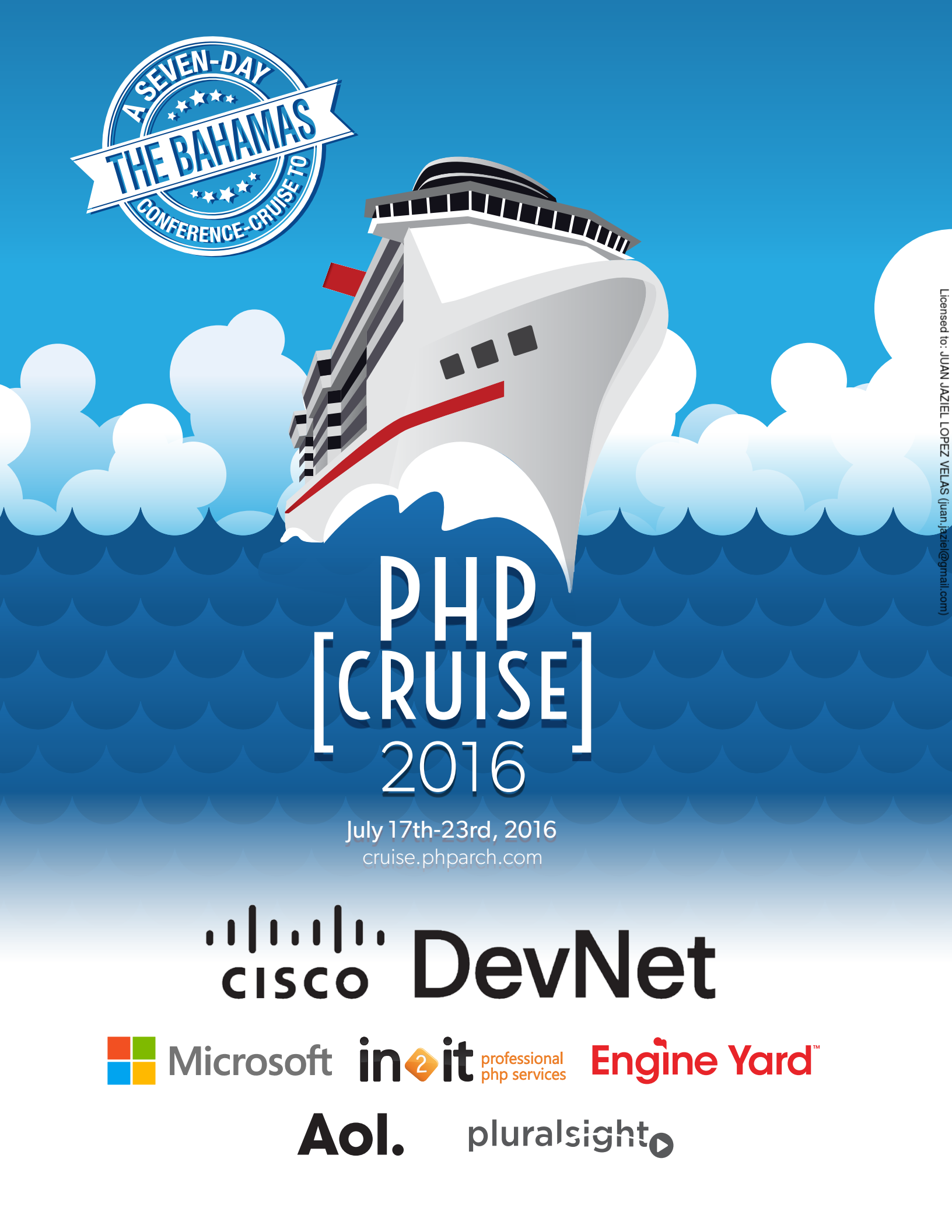
Every set of web services comes with potential attackers. See the reading list in *Learn from the Enemy: Securing Your Web Services, Part One*. Understand your likely threats. Understand your attackers' likely motivations.

Build your own web services security architecture based on your own threat analysis. You now have specific tools for hardening it.



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

8 Certificate & Public Key Pinning: http://phpa.me/cert-pinning



Distributed Workers and Events

Christopher Pitt

You can't throw a stick into a bush these days without it hitting someone who wants to talk to you about asynchronous PHP code. Heck, I'm one of those people. But what we don't tell you is that it's probably not right for you until you learn how to work around PHP's synchronous core.

We can show you examples of non-blocking network I/O and event loops gone wild. But what happens when you need to write a file, or update the database, or any of the other things you do every day?

I recently worked on a library to calculate the difference between two images. The idea was to capture a baseline image of an interface and then compare it to subsequent captures during automated testing.

I discovered that identifying multiple, unconnected regions of change can be computationally intensive on any but the most simple of images. If I were trying to do this in response to an API request, it would block the API from serving any more than a handful of requests at a time.

So how could I distribute this workload to other processes, or even other network nodes? What are some of the challenges I would face? Come along as I take a journey toward truly distributed processing…

Unblocking the Server

Popular asynchronous libraries create easy abstractions for handling network operations, but they're still a little disorganized when it comes to other aspects of general programming.

To create servers that are completely non-blocking, we need to be able to read and write files, communicate with databases, and even crunch a lot of data—all without blocking the same server process.

When one request or response blocks the server process, all other requests and responses are blocked at the same time. An asynchronous server can theoretically handle far more concurrent traffic than a synchronous one, but if one of the requests blocks it for a time, the gain in concurrency is lost.

There are a few ways that we can push blocking operations out of our main process (which handles HTTP requests and responses). Without first-class support for common asynchronous operations in the PHP core, this is the best way to keep blocking operations away from the HTTP server.

Exec

The simplest and most widely-supported method is to start new parallel processes, with the `exec` function[1]:

exec/background.php LISTING 1

```
01. <?php
02. $images = array_slice($argv, 1);
03.
04. foreach ($images as $image) {
05.     $differences = Image::fromPath($image)->getDifferences();
06.
07.     Console::printLine($image . ": " . $differences);
08. }
```

Exec allows parallel processing of operations, although they are limited to the same server.

exec/foreground.php LISTING 2

```
09. <?php
10. $images = join(" ", [
11.     "image1.png", "image2.png", "image3.png",
12. ]);
13.
14. exec("php background.php {$images}", $output);
```

The main process runs `foreground.php` (Listing 1), which starts `background.php` (Listing 2) in the background. There could be a noticeable delay between starting the background script and receiving the output. To avoid that, we can redirect the output:

```
exec(
    "php background.php {$images} > /dev/null &", $output
);
```

> **Be careful when using `exec`!** *Validate and filter any user-defined data, using methods like* `escapeshellarg` *and* `escapeshellcmd`.

Redirecting output removes the blocking problem, but it also introduces a complication we'll come back to in a bit.

If you're able to install extensions, there are a few alternatives to `exec` that are more robust.

1 *PHP exec:* http://php.net/function.exec

Process Control

For process forking, you could try the Process Control extension[2] (commonly referred to as PCNTL). Listing 3 shows an example using PCNTL.

pcntl/foreground.php **LISTING 3**

```
01. <?php
02. $pid = pcntl_fork();
03.
04. if (!$pid) {
05.     $images = array_slice($argv, 1);
06.
07.     foreach ($images as $image) {
08.         $differences = Image::fromPath($image)->getDifferences();
09.
10.         Console::printLine($image . ": " . $differences);
11.     }
12. } else {
13.     pcntl_waitpid($pid, $status);
14. }
```

This extension is also limited to parallel processing on the same server.

The `pcntl_fork` function returns different things for different processes. When it's called, it forks the main process into multiple processes. It returns an integer value to the main process (which is the process ID of the child process fork), and it returns nothing to the child process.

There's a third return value, `-1`, when it can't fork the process. That's rare, but good to know about!

We can wait for the child process fork to complete by calling `pcntl_wait` with the child process ID. That blocks the main process until the child process fork has completed.

PThreads

Alternatively, we can use the PThreads extension[3] to move blocking operations to new threads (see Listing 4).

PThreads provides an assortment of multi-threading classes to extend. In this case, we extend the Thread class, which is where we perform the blocking operations. It's important to know that you can spawn multiple threads with PThreads in much the same way you can fork multiple child processes with Process Control.

Gearman

Gearman[4] is the first option that easily allows work to be done on a network of servers, though it requires slightly more boilerplate to do so. It operates through a daemon—a message queue of sorts—that delegates tasks to running worker scripts. Take a look at Listing 5 to see an example of adding tasks.

pthreads/foreground.php **LISTING 4**

```
01. <?php
02. class ImageDifferencesThread extends Thread
03. {
04.     private $images;
05.
06.     public function __construct(array $images) {
07.         $this->images = $images;
08.     }
09.
10.     public function run() {
11.         foreach ($this->images as $image) {
12.             $differences = Image::fromPath($image)->getDifferences();
13.
14.             Console::printLine($image . ": " . $differences);
15.         }
16.     }
17. }
18.
19. class PingThread extends Thread
20. {
21.     public function run() {
22.         for ($i = 0; $i < 5; $i++) {
23.             sleep(1);
24.             Console::printLine("ping");
25.         }
26.     }
27. }
28.
29. $thread1 = new ImageDifferencesThread([
30.     "image1.png",
31.     "image2.png",
32.     "image3.png",
33. ]);
34.
35. $thread2 = new PingThread();
36.
37. if (!$thread1->start() || !$thread2->start()) {
38.     Console::printLine("Could not spawn thread(s)...");
39. }
```

gearman/foreground.php **LISTING 5**

```
01. <?php
02. $client = new GearmanClient();
03. $client->addServer("127.0.0.1", 4730);
04.
05. $client->setCompleteCallback(function(GearmanTask $task) {
06.     Console::printLine("Task is complete");
07. });
08.
09. $task = $client->addTask("find-differences", serialize([
10.     "image1.png",
11.     "image2.png",
12.     "image3.png",
13. ]));
14.
15. $client->runTasks();
```

2 *PCNTL extension: http://php.net/book.pcntl*

3 *PThread extension: http://php.net/book.pthreads*

4 *Gearman: http://gearman.org*

This connects to the running daemon, and adds a task to it. The task will be delegated to the worker scripts handling the "find-differences" task. Those worker scripts look like Listing 6.

gearman/background.php **LISTING 6**

```
<?php
$worker = new GearmanWorker();
$worker->addServer("127.0.0.1", 4730);

$worker->addFunction("find-differences",
   function(GearmanJob $job) {
      $images = unserialize($job->workload());

      foreach ($images as $image) {
         $differences = Image::fromPath($image)->getDifferences();

         Console::printLine($image . ": " . $differences);
      }
   }
);

while ($worker->work()) {
   if ($worker->returnCode() !== GEARMAN_SUCCESS) {
      Console::printLine("Could not work");
   }
}
```

In both cases, the client and the worker connect to the same Gearman daemon, running in parallel to the worker script. To get this to work, you actually need the following things running at the same time:

- tab 1 → `gearmand`
- tab 2 → `php gearman/background.php`
- tab 3 → `php gearman/foreground.php`

In addition, we've used an event callback. There are a few others we can also hook into:

- `$client->setCreatedCallback()`
- `$client->setDataCallback()`
- `$client->setWarningCallback()`
- `$client->setFailCallback()`
- `$client->setExceptionCallback()`
- `$client->setStatusCallback()`
- `$client->setWorkloadCallback()`

Some of these are triggered automatically, but they can also be triggered from inside the worker:

- `$job->sendData()`
- `$job->sendWarning()`
- `$job->sendFail()`
- `$job->sendException()`
- `$job->sendStatus()`
- `$job->sendComplete()`

But here's where we run into a problem with Gearman. The callbacks are a great way to get updates on how background tasks are progressing, but they also block the main process until all the worker tasks are complete. Once we call `$client->runTasks()`, all processing in the main thread stops. To unblock the main thread, we need to run the tasks in the background, using a special set of Gearman methods:

```
$task = $client->addTaskBackground("find-differences", ...);
```

Once we do that, we no longer have access to the callback functions. They never fire!

The Communication Channels We Have

This is where things become a bit of a hack, in the sense that none of these options really offer a clean solution to this problem. We're able to create background tasks, essentially sending information in a single direction.

But we want to be able to send information in both directions, throughout the life of a blocking operation, and without blocking the main process.

There are a number of things we could try. My favorite among them are message queues. Message queues are essentially distributed arrays, in much the same way as Redis or Memcache are distributed objects.

Message queues maintain named lists

zeromq/push.php **LISTING 7**

```
<?php
$context = new ZMQContext();

$socket = new ZMQSocket($context, ZMQ::SOCKET_PUSH);
$socket->connect("tcp://127.0.0.1:5555");

$socket->send(serialize([
    "image1.png",
    "image2.png",
    "image3.png",
]));
```

of strings, and allow clients to push and pop messages off of them. One of my favorite message queues is ZeroMQ[5] (see Listing 7).

One of the reasons I like ZeroMQ so much is its built-in asynchronous interface. You can use it alongside any of the popular asynchronous libraries, and not have to write much special code for it to be truly asynchronous.

ZeroMQ acts as a kind of shared daemon—push and pull sockets connect to the same "place"—and as long as either side of the connection is open (usually the script pulling messages off the queue), any number of sockets can send push or pull messages to the same place.

zeromq/pull.php **LISTING 8**

```
<?php
$context = new ZMQContext();

$socket = new ZMQSocket($context, ZMQ::SOCKET_PULL);
$socket->bind("tcp://127.0.0.1:5555");

Console::printLine($socket->recv());
```

Pulling messages off the queue looks much the same (see Listing 8).

If we use something like ZeroMQ (or any other message queue, object store, etc., for that matter), we can transfer state between different processes. We can even transfer state to different network nodes.

5 *ZeroMQ:* http://zeromq.org

The Communication Channels We Want

Now that we've seen the asynchronous functions and extensions we can use, and how we could potentially communicate well between them, let's think about what our ideal communication layer could look like…

Have you heard of *The League of Extraordinary Packages*[6]? It's a group of PHP developers who share the burden of maintaining PHP libraries that are held to a high standard of quality, documentation, and testing.

One of their libraries is the Event library[7]. Normal use looks something like this:

```
$emitter = new League\Event\Emitter;

$emitter->addListener("onComplete", function($event, $status) {
    Console::printLine("The task is complete: {$status}");
});

$emitter->emit("onComplete", ["ok"]);
```

Wouldn't it be great if we could work this behavior into something like Gearman, so that we could use the same kinds of events, but also an asynchronous main process? Perhaps something like Listing 9.

ideal/foreground.php **LISTING 9**

```
<?php
$manager = new Acme\Emitter();
$manager->connect("127.0.0.1", 5555);

$manager->addListener("onProgress", function($event) {
    Console::printLine("The task is kinda busy");
});

$manager->addListener("onComplete", function($event) {
    Console::printLine("The task is complete");
});

$manager->addTask($task);
$worker = new Acme\Worker();
$worker->connect("127.0.0.1", 5555);

$task = $worker->getTask();

if ($task->isComplete()) {
    $worker->emit("onComplete");
}
```

To do something like this, we'd probably need to combine the Event library with ZeroMQ (or another tool of comparable functionality). Let's begin by abstracting events into something we can serialize easily, as in Listing 10.

I've chosen the name of `MemoryEvent` for this first implementation (see Listing 11), because the event data is just stored in memory. Since `Event` extends `Serializable`, I've also implemented the serialization methods, so these objects will be easier to send over the wire.

6 *The League Of Extraordinary Packages:* https://thephpleague.com

7 *League:* http://event.thephpleague.com/2.0/

`Serializable` *is built into the PHP core. You can find out more about it at* http://php.net/function.serialize.

When you're dealing with serialization and transmission, it's a good idea to sign and validate your data. I've skipped that here, in the interest of time.

acme/Event.php **LISTING 10**

```
<?php
namespace Acme;

use Serializable;

interface Event extends Serializable
{
    public function getName();
    public function getParameters();
}
```

acme/MemoryEvent.php **LISTING 11**

```
<?php
namespace Acme;

use InvalidArgumentException;

final class MemoryEvent implements Event
{
    private $name;
    private $parameters;

    public function __construct($name, array $parameters = []) {
        $this->name = $name;
        $this->parameters = $parameters;
    }

    public function serialize() {
        return serialize([
            "name" => $this->name,
            "parameters" => $this->parameters,
        ]);
    }

    public function unserialize($serialized) {
        $data = unserialize($serialized);

        if (!isset($data["name"])) {
            throw new InvalidArgumentException("malformed event");
        }

        if (!isset($data["parameters"])) {
            throw new InvalidArgumentException("malformed event");
        }

        $this->name = $data["name"];
        $this->parameters = $data["parameters"];
    }

    public function getName() {
        return $this->name;
    }

    public function getParameters() {
        return $this->parameters;
    }
}
```

Next, we need to consider what the manager (or server) interface will look like, as shown in Listing 12.

acme/Server.php **LISTING 12**

```
01. <?php
02. namespace Acme;
03.
04. use Closure;
05.
06. interface Server
07. {
08.     public function addListener($name, Closure $closure);
09.     public function removeListener($name, Closure $closure);123
10.     public function emit($name, array $parameters = []);
11.     public function tick();
12. }
```

Each `Server` implementation should provide methods to add, remove, and emit events (just like the Event library). In addition, we'll need to make things non-blocking, which basically means polling ZeroMQ for new events without blocking other processes. That's the reason for the `tick` method. The implementation is a bit of work, as you can see in Listing 13.

Much of this new class just deals with adding, removing, and emitting to event listeners. Each listener is stored first according to the name of the event and then by the hash of the closure provided.

The tick method tries to pull a message off the queue, and if it finds one, it tries to unserialize it so that it can be dispatched to the event listeners.

> *Note the use of* `ZMQ::MODE_DONTWAIT`. *Without it, the tick method would block until a message was received.*

On its own, the server is no more useful than just using the Event library on its own. To really make it shine, we need to define a client interface (Listing 14).

acme/Client.php **LISTING 14**

```
01. <?php
02. namespace Acme;
03.
04. use Serializable;
05.
06. interface Client
07. {
08.     public function emit($name, array $parameters = []);
09. }
```

acme/ZeroMqServer.php **LISTING 13**

```
01. <?php
02. namespace Acme;
03.
04. use Closure;
05. use ZMQ;
06. use ZMQContext;
07. use ZMQSocket;
08.
09. final class ZeroMqServer implements Server
10. {
11.     private $socket;
12.     private $listeners = [];
13.
14.     public function __construct($host, $port) {
15.         $context = new ZMQContext();
16.         $this->socket = new ZMQSocket($context, ZMQ::SOCKET_PULL,
17.                                       spl_object_hash($this));
18.         $this->socket->bind("tcp://{$host}:{$port}");
19.     }
20.
21.     public function addListener($name, Closure $closure) {
22.         if (empty($this->listeners[$name])) {
23.             $hash = spl_object_hash($closure);
24.             $this->listeners[$name][$hash] = $closure;
25.         }
26.
27.         return $this;
28.     }
29.
30.     public function removeListener($name, Closure $closure) {
31.         $hash = spl_object_hash($closure);
32.
33.         if (isset($this->listeners[$name])
34.                 && isset($this->listeners[$name][$hash])) {
35.             unset($this->listeners[$name][$hash]);
36.         }
37.
38.         return $this;
39.     }
40.
41.     public function tick() {
42.         if (!($event = $this->socket->recv(ZMQ::MODE_DONTWAIT))) {
43.             return;
44.         }
45.
46.         $event = unserialize($event);
47.
48.         if (is_object($event) && $event instanceof Event) {
49.             $this->dispatchEvent($event);
50.         }
51.     }
52.
53.
54.     private function dispatchEvent(Event $event) {
55.         $name = $event->getName();
56.
57.         if (isset($this->listeners[$name])) {
58.             foreach ($this->listeners[$name] as $closure) {
59.                 call_user_func_array($closure, $event->getParameters());
60.             }
61.         }
62.
63.         return $this;
64.     }
65.
66.     public function emit($name, array $parameters = []) {
67.       return $this->dispatchEvent(new MemoryEvent($name, $parameters));
68.     }
69. }
```

The client has one job—to emit events through ZeroMQ back to the listening server. The implementation is a bit more code (see Listing 15).

acme/ZeroMqClient.php **LISTING 15**

```
01. <?php
02. namespace Acme;
03.
04. use Exception;
05. use ZMQ;
06. use ZMQContext;
07. use ZMQSocket;
08.
09. final class ZeroMqClient implements Client
10. {
11.     private $socket;
12.
13.     public function __construct($host, $port) {
14.         $context = new ZMQContext();
15.
16.         $this->socket = new ZMQSocket($context,
17.                             ZMQ::SOCKET_PUSH, spl_object_hash($this));
18.
19.         $this->socket->connect("tcp://{$host}:{$port}");
20.     }
21.
22.     public function emit($name, array $parameters = []) {
23.         $event = new MemoryEvent($name, $parameters);
24.
25.         $this->socket->send(serialize($event), ZMQ::MODE_DONTWAIT);
26.     }
27. }
```

The client is much simpler than the server, although that's to be expected since it only needs to emit events. Emitting events from the client has only the appearance of emitting to an actual event emitter.

In fact, the events are serialized and stored in ZeroMQ until the corresponding server pulls them off the queue and handles them.

We can use these in the following ways:

```
$server = new Acme\ZeroMqServer("127.0.0.1", 5555);

$server->addListener("onComplete", function() {
    Console::printLine("The task has completed");
});

while (true) {
    $server->tick();
    sleep(1);
}
```

We can call the `tick` method inside an infinite loop, and when new serialized events are picked up, the appropriate events will be emitted to waiting listeners.

```
$client = new Acme\ZeroMqClient("127.0.0.1", 5555);

$client->emit("onComplete");
```

Running these in two separate tabs, we can effectively send and receive messages between processes. The infinite loop is a bit of a placeholder for something better—an event loop.

Event Loops

Popular asynchronous libraries like React[8], Icicle[9], and AMPHP[10] implement event loops. These are similar to infinite loops in that they are designed to carry on until they are shut down. They also allow polling of various resources, sockets, and so on.

It's easy to use our server with an event loop:

```
Icicle\Loop\periodic(1.0, function() use ($server) {
    // this will happen every second
    $server->tick();
});
```

If we were to use the infinite loop approach, our main process would block while waiting for new event messages from the queue. If we start to use an event loop, we can still get new events from the queue, but we can also do non-blocking operations like receiving and responding to HTTP requests.

Conclusion

> *I've implemented this distributed event emitter for you to use, over at Github[11].*

There really are a lot of ways to execute code in parallel. These methods are useful for isolating blocking operations away from the highly concurrent parts of our applications.

But only when we fully understand the ups and downs of each, and ways to get the communication we really desire between them, can we start to design great asynchronous systems.

With this post, I hope I've given you a taste for API design that transcends traditional, synchronous PHP architecture. Perhaps it's time for you to take that really inefficient part of your application and transform it into a marvel of high performance.

Christopher Pitt is a developer and writer, working at SilverStripe. He usually works on application architecture, though sometimes builds compilers or robots. @assertchris

8 React: http://reactphp.org

9 Icicle: https://icicle.io

10 AMPHP: http://amphp.org

11 asyncphp/remit: https://github.com/asyncphp/remit

The Middleware Awakens

Ian Littman

Back in May 2015, PSR-7 went gold, providing a framework-agnostic HTTP request and response interface that is usable in both client-side and server-side contexts. A year later, we've got a double handful of frameworks implementing the PSR, and myriad middleware layers that are usable with any of them. In this post, I'll give a history lesson on how we ended up here, then show how to piece together middlewares and frameworks to quickly augment a web app's functionality.

This is not the PSR you're looking for

This article was written prior to the PHP FIG's HTTP Middleware PSR discussion, and reflects the proposed middleware interface used by Slim 3, Zend Expressive, and a few others. An alternate interface being discussed looks more like what StackPHP and Laravel use; see Anthony Ferrara's blog post[1] on the merits of this format. The intent of this article is not to crown the former approach as the One True Middleware Standard, though the examples do use that format.

HTTP and the $_SERVER Menace

Let's pretend for a moment that PHP web frameworks don't exist, and rewind a few years (if you'd rather not, skip this section). Suppose you have a language built for the web interacting with your web server, either via a CGI variant or as a module built into the server. Your script starts when a web request comes in, looks at a few components of the request (query strings, form-encoded POST parameters and cookies are easiest), decides what to generate as a response, and maybe remembers to set a header or two before the response is sent out.

Easy enough for slapping together some dynamic functionality on a web site, where end users may not utilize the more nuanced aspects of the HTTP protocol. Not so easy when you're trying to use HTTP messages as first class citizens—in an API, for example. Heck, even working with direct file uploads or JSON request payloads is a slight stretch. Testing? Better buffer your output and wait to send your headers, lest you let the genie out of the bottle prematurely.

So you build an abstraction layer between your code and the raw output-buffering tools PHP gives. You figure out how to map `$_SERVER['HTTP_*']` (for versions of PHP from recent history) to most of your headers, a few other scattered server vars to others, `php://input` to your HTTP POST body, and so on. If you're unlucky, the framework you find yourself building has to run on multiple browsers, complicating matters further; Apache and NGINX handle requests a bit differently.

Now, this situation isn't entirely the language's fault: its interaction with the server twiddling HTTP bits directly has historically been over CGI or something close to it. The CGI protocol effectively sets a load of request-local environment variables (`$_SERVER`), then pipes the body of the HTTP request to a script, then gets headers and a response body as output of the script. PHP's abstractions on top of HTTP reflect that protocol.

That said, other languages that didn't grow up hand-in-hand with CGI don't have that historical baggage. As a result, they (Ruby Rack, Python WSGI) picked abstractions the HTTP message that are, while not perfect, cleaner. PHP was still the easiest way to churn out templated HTML, but for folks wanting to build apps that more cleanly abstracted HTTP interactions, frameworks needed to build that layer in userland.

And build they did, with each framework piecing together a slightly different interface to make proper HTTP interaction a bit more palatable. The result: frameworks large and small now have some sort of HTTP abstraction, with a number using Symfony's `HttpFoundation` objects directly, or implementing something similar. These request and response objects snake their way through the execution lifetime of a PHP call, revised along the way via getters and setters, and in some cases constructor-injected into an MVC's controller. Still coupled to a single-request-per-script-invocation model, but cleaner than what was there before.

Stacks, Immutability, and Attack of the (Clone)s

Enter Node.js and Express, an incredibly popular web micro-framework inspired by Ruby's Sinatra. To avoid a mess of unmaintainable JavaScript spaghetti, the framework encourages developers to do one thing at a time to the HTTP request and response as they pass through a stack of *middlewares*. A middleware is implemented as a callable that takes the then-current version of the request, the then-current version of the response, and a callback for the next layer in the stack. It might do something as simple as removing a header or two to ensure that vulnerability scanners have to look harder to figure out which version of Express you're running.

This paradigm of operation inverts the HTTP request's role, as compared to PHP. Instead of building an entire execution environment around a single request and then running that environment, requests flow through a fixed set of interactions like marbles through a maze. The plural component in this analogy is important, since in the case of Express the server outlives the request by a large margin, and multiple requests may be traversing the maze

1 *Anthony Ferrara: All About Middleware:* http://phpa.me/ircmaxell-middleware

simultaneously, provided that all but one of them are waiting on some sort of non-blocking I/O at any point in time.

A few projects in PHP started mirroring this middleware-centric implementation, although they continued to be served up one request at a time. Notable examples include the StackPHP project[2] (based on Symfony's HttpKernel) and Slim Framework 2[3]. As full-stack framework silos were broken into their constituent components, composing middlewares into a coherent application started looking like a tenable option for building reusable HTTP interaction stacks… as long as the middleware interfaces matched. And some of the more interesting middleware applications could be used with an HTTP client (like Guzzle) or an HTTP server (or a framework pretending to be one).

Enter what was to become PSR-7[4], a spec for HTTP message interfaces, championed by both client-side (e.g., Guzzle) and server-side (e.g., Zend, Slim) parties. While the spec didn't specify middleware signatures per se, it did nail down interfaces that those middlewares could use, without ever having to worry about munging superglobals in the process. As a result, the leap to a common middleware spec was a short one: Slim Framework 3 implemented PSR-7 before it was accepted[5], tracking the spec's changes on an untagged 3.x branch until it stabilized. Zend's Matthew Weier O'Phinney built `phly/http`, now Zend Diactoros[6], modeled after Node's HTTP server using the PSR-7 interface, then built on that foundation with phly/conduit, now Zend Stratigility[7], which shares Slim's middleware spec.

Why immutability?

There was debate back and forth on the topic leading up to acceptance of the PSR. The final decision boiled down to the fact that HTTP messages are effectively value objects; for example, changing a request from what it was on the wire creates a new, different, HTTP message that should be treated as such. Enforcing immutability means that a message's flow through an application is explicit rather than implicit, allowing for cleaner debugging and the opportunity for performance enhancements that couldn't happen if a request or response object's state could change out from under something operating on it.

As an example, an HTTP client can frame up the common parts of a request first, then send off variants of that request (e.g., with a revised path) without worrying that they're changing the state of one request by sending another. Another example: middlewares operating on immutable objects can behave like pure functions, with the testability, memoization, and other functional-programming benefits that arise from that construct. Put another way, immutability makes it a bit easier to prove your app does what you think it's doing.

The one catch to the immutability bit is that message bodies are exposed as streams, which can't necessarily be locked down, although the spec recommends that HTTP client response and server request streams be opened as read-only to avoid potential issues with this caveat.

2 *StackPHP: http://stackphp.com*

3 *Slim Framework 2: http://docs.slimframework.com*

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

5 *Slim 3 Middleware Docs: http://phpa.me/slim-middleware*

6 *Zend Diactoros PSR-7 Implementation: http://phpa.me/zend-diactoros*

7 *Zend Stratigility Middleware Pipe: http://phpa.me/zend-stratigility*

The Slim-pire Calls Back

An extended version of this example is online at http://ian.im/s3p7gist1.

With that bit of history out of the way, let's look at how to use this type of middleware with one of the frameworks built around it: Slim 3. Its request and response classes implement (and slightly extend) PSR-7's `ServerRequestInterface` and `ResponseInterface`, respectively, and its callbacks take the format shown in Listing 1.

LISTING 1

```
01. <?php
02. // using PHP 7 group use declarations and return typehints
03. use Psr\Http\Message\{RequestInterface, ResponseInterface};
04.
05. $middleware = function(RequestInterface $req,
06.                        ResponseInterface $res,
07.                        $next) : ResponseInterface {
08.     // do stuff with request and response before passing on
09.     $res = $next($req, $res); // pass on to next in the stack
10.     // do stuff with response after passing on
11.     return $res;
12. };
```

The above middleware will be the currently running function twice during the life cycle of a web request: once before calling `$next`, and once after. `$next` calls either the next middleware later if there is one, or the actual route callable if that's all that's left. Conversely, when the middleware returns, it cedes control to the next layer outward, or to Slim's response output procedure if the current middleware layer is the outermost.

At any point, a middleware can decide to return a response right away rather than passing responsibility further down the stack. This allows for quick handling of events like authentication failures or rate limit events; just return early when the API client has called one time too many.

Middlewares can be (and usually are, for third-party cases) class instances, provided they implement `__invoke()` with the above signature. They can take advantage of Slim's dependency injection capabilities to, for example, interact with a constructor-injected authentication layer that knows nothing about the intricacies of HTTP.

In Slim's case, middlewares are added to a route, route group, or the entire app by chaining `add()` calls to the appropriate place. These calls can either use a closure directly, reference a class name, or reference a service that has been registered with the Slim container.

Putting all of the above together, let's set up a quick auth middleware that, based on the Authorization header, either short-circuits the request or passes it through:

… and now let's use it in a quick app, along with a few other middlewares, neither of which are tied to Slim's implementation of PSR-7.

Each of these layers isn't much to look at, but taken together they result in an app that has authorization that can take an IP whitelist into account[8], with user attributes following the request associated with them rather than polluting global state… and we didn't even

8 *RKA IP Address Middleware: http://phpa.me/akrabat-ip-middleware*

Authentication middleware **LISTING 2**

```
<?php
class MyAuthMiddleware
{
    protected $auth;

    public function __construct($authService) {
        $this->auth = $authService;
    }

    public function __invoke($req, $res, $next) {
        if (!$req->hasHeader('Authorization')) {
            return $res->withStatus(401);
        }

        $auth = $req->getHeader('Authorization');
        $user = $this->getUserFromHeader($auth)[0];
        if (!$user) {
            return $res->withStatus(401);
        }

        if ($user->isBanned()) {
            return $res->withStatus(403);
        }

        return $next($req->withAttribute('user', $user), $res);
    }

    // getUserFromHeader() implemented here
}
```

A Slim 3 app with various middlewares **LISTING 3**

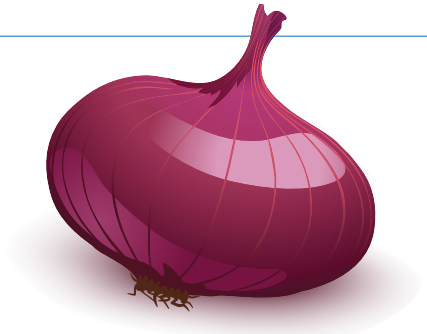
```
<?php
$app = new Slim\App();
// define an auth dependency at index "auth" here
$app->getContainer()['authMiddleware'] = function ($c) {
    return new MyAuthMiddleware($c['auth']);
};

$app->get('/hello', function($req, $res, $args) {
    return $res; // do all the work in a middleware instead
})->add(
    function($req, $res, $next) { // inside auth middleware
        $res->getBody()->write("Hello ");
        $res = $next($req, $res);
        $res->getBody()->write(
            $req->getAttribute('user')->getName() . "\n"
        );
        return $res;
    }
);

$app->get('/bye', function($req, $res, $args) {
    return $res->withStatus(204);
}); // no Hello here!

// global middlewares
$app->add('authMiddleware'); // has access to IP set from below
$app->add(new RKA\Middleware\IpAddress); // outermost middleware

$app->run();
```



have to write the finicky IP detection code!

Want more middleware inspiration? Check out `oscarotero/psr7-middlewares` on GitHub and Packagist for a collection of examples in a single repo[9]. Rather layer your middlewares on something other than Slim? Zend Expressive is built on top of Stratigility, which is built on top of Diactoros, which implements PSR-7 and expects the same Request-Response-next middleware signature.

A Note on Lumen

One thing PSR-7 did *not* do was dictate that all frameworks conform to its demands. Instead, the spec encourages the use of adapters where ripping out a perfectly good HTTP message format would be too difficult. One such case is Symfony, including downstream projects such as Laravel and Lumen[10]. For Lumen in particular, pulling in a couple of libraries (Symfony's PSR-7 bridge[11] and Zend Diactoros) is enough to allow route and controller callables to take PSR-7 requests in and spit PSR-7 responses out.

Middlewares, on the other hand, may not be set up for this. Lumen's middleware signature uses two parameters rather than three, with separate methods for running a middleware before or after the core app route logic[12]. Also, in my tests any modifications to the HTTP request were thrown away once one of the above middlewares handed the request back to Lumen's stack. That's not to say that Lumen's middleware system (or Silex's, for that matter) is bad. It just won't work with the examples provided here.

Return of the Async?

One of the potential benefits of PSR-7's immutability is that immutability makes interleaved flows through an application safer, to the point that long-lived web servers with concurrent, asynchronous requests would find PSR-7 to be a natural fit. Middlewares would sit along the HTTP server's execution path, revising requests and responses as they reached that stage in the stack.

The catch here is that the PSR-7 stream interface, based strictly on the spec rather than the underlying implementation, doesn't have non-blocking write support. This throws a wrench into single-threaded event loops like React and Icicle, which can't (yet) farm out writes to the HTTP streams to other processes so their main process doesn't hang.

That said, bridges between the two projects and PSR-7 do exist, including one by Matthew Weier O'Phinney between React and Zend Diactoros[13]. Also, both React and Icicle's HTTP message classes are either similar to PSR-7's interfaces or are headed in that direction, making integrations a bit easier. As these frameworks integrate multi-threaded or multi-process concurrency, PSR-7 just might offer a new hope for those contexts.

When Ian isn't building, maintaining, optimizing or complaining about APIs of the companies he works with, or playing video games poorly, he's enjoying the Austin, TX countryside and reverse-heckling the likes of Keith Casey in his hometown. @iansltx

9 Oscar Otero Middleware Collection: http://phpa.me/otero-psr7

10 PSR-7 In Lumen: http://phpa.me/lumen-psr7

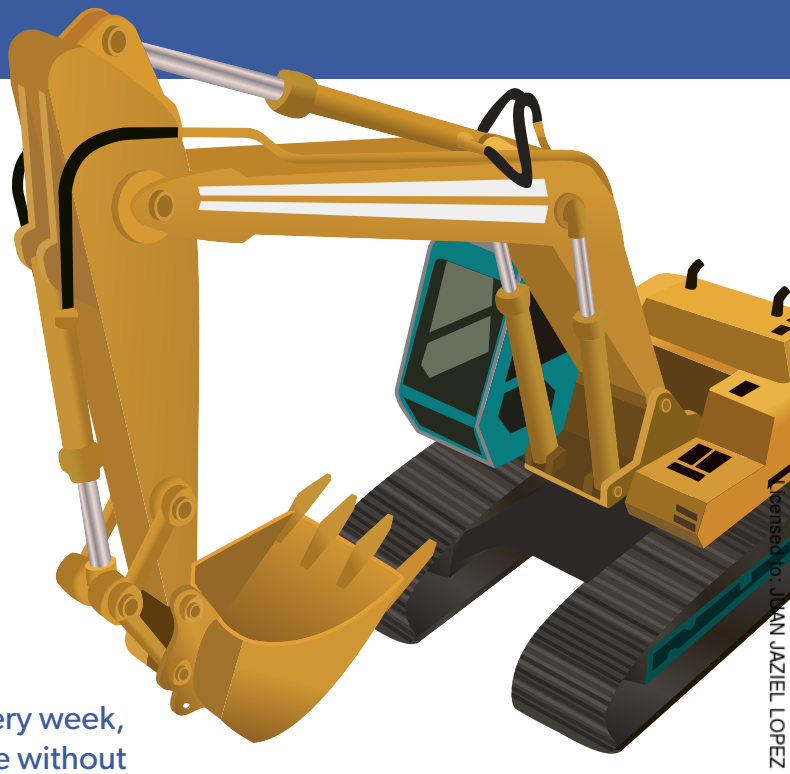
11 Symfony PSR-7 Bridge: http://phpa.me/symfony-psr7-bridge

12 Lumen Middleware Docs: https://lumen.laravel.com/docs/5.2/middleware

13 PSR-7 React Bridge: http://phpa.me/react-psr7

Professional Development for Professional Developers

Steve Grunwell



Programming is often a full-time job, but so is being a part of the programming community. With new technologies, tools, and buzzwords popping up every week, how can a professional stay on top of his or her game without going completely insane?

As developers, we have one of the best jobs in the world: we get to solve problems every day, reducing "real" business problems into puzzles made of data, logic, and code. Rather than performing hard, back-breaking labor, we have the luxury of being presented with a situation and tasked with instructing a machine how best to handle it.

We developers are artisans: Our studio, an office or coffee shop; our tool, a keyboard; our canvas: a text editor. As such, we mustn't rest on our laurels but strive to be better, constantly growing and improving. With so much to see and do, how can a programmer—especially one with a full-time career—continue to hone his or her craft?

The following guidelines come from nearly a decade of software development experience—a self-study driven by passion, curiosity, and taking tremendous pride in a job well done. Whether you're a computer science student, a self-taught developer just getting started, or a seasoned professional, these nuggets of wisdom can help you be the developer you've always wanted to be.

Programming Should be Your Passion

Whether a fresh-faced junior or a battle-tested senior, the best trait you can possess as a developer is to be passionate about what you do. You don't have to dream in code (though it probably doesn't hurt), but 4 out of 5 dentists agree that someone passionate about their work has infinitely more potential than someone who is talented but lacks ambition.

What does it mean to be passionate? It means finding yourself thinking about the best way to tackle a problem, even when you're off the clock. Being so excited to try out a new framework or technique that you look up at the clock and realize it's past midnight. Passion is that euphoric release you get when you've finally shipped your code and you feel as though **there's no problem you can't solve**.

I'll caution you, however, to not confuse a passion for an obsession. Being a great developer does not mean chaining your life to the keyboard, churning out code until your eyes bleed. The best developers know the value of engaging in activities away from the computer (you know, the "real" world), and the best employers will recognize this, too.

I don't play the lottery, but when the jackpots get high people love to talk about what they'd do were they to strike it rich. When people ask me what *I* would do, I always give the same answer: "If money were no object and I didn't have to work, I'd still be writing software, **but it would be the software I want to write.**"

Sure, I'd probably have a beach house somewhere, and I might be too busy traveling the world to put in a regular 40-hour work week, but software will always be a part of my life *because* it's something that I'm passionate about.

Keep Up with the Industry...

…but not *everything* going on. The software industry is constantly changing, and it can be a struggle to keep up with news about the technologies you're *already* using, never mind the new frameworks, libraries, and workflows that are popping up every week.

Relax, no (reasonable) person is expecting you to be the expert on everything. Once you accept this fact, the fear and anxiety around keeping up with *all the things!* should fade away.

Find the ways to get news that fit your lifestyle the best, be it Twitter, RSS feeds, magazines, or user groups. Look up profiles of other professionals you admire, and see who *they* are following. Seek out Slack channels relevant to your interests, and subscribe to newsletters that compile the news of the day or the week for you.

Attend a conference (or several). Sit in on a session about a new technology you've been curious about but haven't had time to research; even half an hour of hearing firsthand experiences about a tool can be far more beneficial than hours of grokking documentation and "Hello World" apps.

If you don't have the time or budget for a conference (or you don't like to travel), seek out remote learning opportunities like NomadPHP[1]. Most larger cities have locally organized monthly meetups, too.

The world of software development moves pretty fast; if you don't stop and look around once in a while, you could miss it.

1 NomadPHP: https://nomadphp.com

Don't Worry About What Other People are Doing

It's not a new phenomenon, but our hyperconnected world only amplifies the "Fear of Missing Out" (FOMO). Everyone's talking about #phpcruise on Twitter, but you have a scheduling conflict. Your coworker is flying across the world to give a keynote at a conference, and you're just binge-watching shows on Netflix.

It's easy to look at the fun that other people are having and feel as though you're missing out, but what we often fail to realize is that we're comparing the entirety of our lives to someone else's highlight reel. Are you really "slacking off" in the community, or are you choosing to spend your evenings with your family instead? Don't let a single interest, no matter how passionate you are, consume you.

One thing that unifies every person on Earth, rich or poor, is that we only have so many hours in a day; balance them among your passions and interests, let others do the same, and share your experiences together. You're not missing out, you're simply spending your time differently.

Keep an Eye on Your Stress Levels

Writing software isn't always a walk in the park, and it's very easy to fall prey to taking on too much work in too little time. Perhaps it was a bad estimate and now you're in over your head, or maybe your sales team committed to an unrealistic timeline, but we've all faced times where we felt under the gun to deliver.

Relax. Take a deep breath. Very few of us are writing code that would cause irreparable harm if it didn't work perfectly (this advice does not apply, however, to those programming weapons targeting systems, automated transportation, and/or medical equipment), so don't treat every deadline like a life-or-death situation. If it seems like you might not be able to fully deliver on time, work with your team and the project stakeholders to adjust expectations; you may be surprised how understanding people can be.

If your employer offers you paid vacation time, take advantage of it (and if they don't, find an employer who does); burnout is a **very** real thing, and if you're not careful you may find yourself loathing this career which, as we've already established, you should be passionate about. Even the best of us know that sometimes it's necessary to unplug for a while and recharge our mental batteries. Take a trip, pick up a new hobby, or enjoy the timeless tradition of the "staycation" and veg out on the couch.

If the burnout doesn't recede, find someone you trust to talk to about how you're feeling. There's a frightening number of developers living with mental health issues, including depression and anxiety. Fortunately, there are also organizations dedicated to providing support for those in the technology industry suffering from these illnesses. Learn about mental health and its treatments, find out what mental health benefits your employer offers, and know that **there's no shame in seeking help**. Some community resources to check out include:

- Prompt: Mental Health in the Technology Industry—http://mhprompt.org
- Open Sourcing Mental Illness—http://funkatron.com/osmi
- Mental Health First Aid—http://www.mentalhealthfirstaid.org

Keep a Record of Your Ideas

When your mind is geared to solve problems, you can never be sure when inspiration might strike. Find a way to write down these ideas to revisit later; I've found that Evernote[2] can be great for this, as my ideas are synchronized between my computer and phone. I also keep pens and pads of paper around the house and in my car, just in case something hits me when I'm away from a screen.

Make a point of periodically going back through these ideas. Some will be laughable or completely off-the-wall, but you might surprise yourself with what you dream up when you're not actively *trying* to invent something.

Having a place to keep track of ideas for later can also be a great way to stay focused on the task at hand and not get distracted; sometimes a client call or a watercooler chat with a coworker is enough to spark an idea, but you if you can't afford to spend the time fully exploring it (or don't want to risk doing that on company time), jot down as much as you can and keep the note handy throughout the day. As it stews on your mental back burner, add details to the note as they form, so you don't miss anything when you revisit the idea at a later date.

2 *Evernote, free: https://evernote.com*

Open-Source FTW

The barrier to entry for developers has never been lower, and that's due largely in part to open-source software. These tools, open to examine and often free to use, make the cost of getting started with software development almost nil (you still need a computer and an Internet connection, after all).

Need an editor? Atom is free and gaining lots of attention. An operating system? There are dozens of flavors of Linux, free for you to install and modify to your heart's content. An astonishing amount of the Web runs entirely on open-source software, and many of its authors welcome outside contributions.

Do you wish that program could do X? Awesome, reach out to the authors and suggest it. If you're able, write the feature yourself and open a pull request; your dream feature could become a permanent part of the application for everyone to use, and before long the original author may be asking for *your* advice on how the project should move forward!

Open-source software can also be a tremendous educational tool; instead of treating our software as mysterious black boxes, free and open-source software (FOSS) pops the hood and lets you see how the internals work together. When you're learning a new language or framework, there's nothing better than being able to poke around the codebase of a real, in-the-wild application.

Employers love open-source, too: there's no better way to prove "I know how the internals of this software work" than being listed as a project contributor. If you're in the market for a new job—especially if you're just breaking into programming—making open-source contributions is one of the single best things you can do to bolster your résumé, attract recruiters, and break into the software community.

Code Speaks Louder than Words

You can read every blog, complete every tutorial, and spend hours a day working on Code Katas, but the ultimate test of your knowledge is shipping working production code. At some point, every developer must finally commit (no pun intended) to pushing code out to the world.

Yeah, it's scary. Maybe you made some mistakes. What if they don't like it? What if they told you you're no good? Well buck up, George McFly, because making mistakes

and getting feedback is how we better ourselves.

Put that pet project up on GitHub and tweet out a link. Ask your colleagues, mentors, and other trusted professionals to take it for a spin. With any luck, someone will find a bug or some weird edge case you hadn't considered. The next time you look at this problem, you'll remember that and approach it differently.

Likewise, if you're using an open-source tool and find a bug, don't be afraid to let the author know. These works are often labors of love, and if you're having a problem with it there's a good chance you're not the only one. Offer to help the author fix the issue, even if just by way of a good bug report. It's easy to be an armchair programmer, proclaiming "that software sucks"; put your money where your mouth is and write something awesome.

Never Stop Learning

The day a programmer decides that he or she have learned everything they need to know is the day that coder's professional development comes to a screeching halt. There are *always* new things to learn, new ideas to be explored, and old assumptions to be questioned.

Make sure you're regularly revisiting old projects, even if you're not responsible for maintaining them. Is that code something you'd ship today? If the answer is "no" (or "not without some refactoring") then congratulations, you're a better developer than you were when you wrote that code months ago!

> *If you don't find at least one thing that makes you cringe in code you wrote six months ago, you might be doing it wrong.*

The new things you choose to study don't even have to be directly related to your area of expertise. Moving at your own pace, learn Ruby on Rails and see how you can apply its principles to your next PHP app. Write something in Python just to try it out. Dig into the internals of a library you use all the time to figure out how it works, what makes it tick. These are all activities that will broaden your horizons to what other languages, frameworks, and communities are doing. Then, take that knowledge back to your own work and repeat the cycle.

Find (and be) a Mentor

Mentorship is one of the single best ways for someone to get stronger in an area of study. Whether a formal, scheduled mentorship or an ad hoc relationship with someone you can bounce ideas off of, a mentor helps you grow, challenges your assumptions, and shares the benefit of his or her experiences.

My first mentor, Matt, was a web designer at work. At the time, I was doing mostly video production until my boss asked if I could try to make heads or tails out of some PHP a former employee had written. With only some BASIC programming under my belt, I started deconstructing the code, figuring out how it all worked. Whenever I'd get stumped, I'd pop into Matt's office and ask him if he could explain it to me.

Before long, I got the hang of writing PHP applications. Matt and I would toss ideas around during lunch, and I'd try to build them in the evening. Naturally, there were a lot of projects that never got off the ground and the code was primitive, but these exercises gave me the confidence that so many junior developers sorely lack.

I've had many more mentors throughout my career, but it's only recently that I've embraced the idea of acting as a formal mentor for other developers. If you have experiences to share, I'd encourage you not only to learn from mentors but to pay that favor forward by joining an organization like PHP Mentoring[3].

Learn About Humility

Programmers tend to be logical, analytical people; we don't run from a problem, but instead try to reason out the best way to approach it. Even if our grades didn't necessarily reflect it, I'd be willing to bet a fair number of us would say that school came "pretty easily" to us, forcing us to satisfy that urge for mental stimulation through advanced coursework, extracurriculars, or tinkering (whether the faculty approved or not).

Being one of the smartest people in the room throughout school can make for a rude awakening when you start your career. Even if you're an A-player within your company, you're bound to cross paths with someone in the community who knows more than you do.

When this happens, there are a few ways to handle it: you could feel threatened, become confrontational, get exiled from the community, and live with a chip on your shoulder forever, **or** you could admit that this person may know more about this one specific topic, learn from him or her, and then share what you've learned with the larger community. The choice is yours, but only one lets you…

Be a Positive Force in Your Community

Believe it or not, computers don't care what race, gender, sexual orientation, religion, or age their programmers are. There's nothing that makes a 27-year-old white male a better programmer than a 55-year-old Latina woman, yet some bad apples in the development community seem to think it's okay to proclaim otherwise. These people are known as "trolls," and **we do not feed the trolls!**

If you don't fit into the stereotypical model of "a programmer," **good!** The technology industry isn't meant to be some exclusive club where you need to know the secret handshake to be accepted. Rather than dividing ourselves, we should celebrate the different viewpoints that a rich, diverse community can offer. You can do your part by being open to new ideas, accepting of all well-intentioned community members, and not being afraid to call someone out when they're making other people feel threatened or unwelcome.

We're All in This Together

No two developers are exactly alike, nor will their journeys be exactly the same. Take comfort in knowing that you're a part of the larger software community, a fraternity of creative thinkers, problem-solvers, tinkerers, and puzzlers. Our industry is one where a little curiosity can open entirely new worlds, so ask questions, try new things, and never stop moving forward.

Steve Grunwell is a Senior Web Engineer at 10up, living and working in Columbus, OH. When he's not writing software, he can be found speaking at conferences, blogging about software development, or continuing his search for the perfect cup of coffee. @stevegrunwell

3 *PHP Mentoring:* https://phpmentoring.org

MySQL's JSON Data Type

Dave Stokes

You can now store JSON documents in a column of a MySQL database. MySQL 5.7 was released in October of 2015 with many new features, but the most popular with developers has been the native JSON data type. So like an integer or a text field, you can now shove an entire valid JSON document into a single column of your database.

In the pre-5.7 days, many developers packed text fields or BLOBs with JSON data. But it was hard to search, manipulate, or use that data. Often, nasty REGEX expressions would be needed for even the most basic lookups. You could store the key/value data pairs, but you probably would not want to.

What Does a JSON Column Look Like?

The JSON data type is just like any other data type. You can easily create tables and shove in your JSON data. Behold:

```
CREATE DATABASE phpa;
USE phpa;
CREATE TABLE phparch (stuff JSON);
INSERT INTO phparch (stuff)
    VALUES ('{"a" : 1, "b" : 2, "c" : 3}');
```

A `SELECT * FROM phparch;` will return all the data from the `phparch` table.

```
+--------------------------+
| stuff                    |
+--------------------------+
| {"a": 1, "b": 2, "c": 3} |
+--------------------------+
```

I highly recommend that you risk severe finger trauma by typing the above code into an instance of MySQL 5.7 to follow along. It is faster to go down a guided path than to stumble by yourself in the dark.

Now it has to be a valid JSON document, so no Garbage-In-Garbage-Out. The MySQL server checks for validity, places the data in a binary optimized format for quick searches, and you can use regular, old-fashioned Structured Query Language to get to your data. Also, if you repeat keys you will find that only the first key will stick and the duplicates will vanish. So `A:1, B:2, A:3` will end up as `A:1, B:2`.

So far so good? JSON is just like all the other native data types. Only more so!

JSON Functions

So now that you have all that lovely JSON data, you probably want to use it. To support the new JSON data type, there are new functions for creating, searching, modifying, and getting metadata information on it.

The JSON_EXTRACT function returns your old binary friends for true/yes or false/no. So a search for names from a names column in a table named info would look like:

```
SELECT JSON_EXTRACT(stuff, "$.c") FROM phparch;
```

Note the `$.c'` where the `$` refers to the current row of data and `c` is the name of key. And the key `"c"` gives us the value of *3*.

```
+----------------------------+
| JSON_EXTRACT(stuff, "$.c") |
+----------------------------+
| 3                          |
+----------------------------+
```

There is even a shortcut, `column->path`, for `JSON_EXTRACT`, so that you could use the following in place of the above query:

```
SELECT stuff->"$.c" FROM phparch;
```

Returns the following:

```
+--------------+
| stuff->"$.c" |
+--------------+
| 3            |
+--------------+
```

The statement below:

```
SELECT JSON_KEYS(stuff) FROM phparch;
```

Will return the keys `a`, `b`, and `c`.

```
+------------------+
| JSON_KEYS(stuff) |
+------------------+
| ["a", "b", "c"]  |
+------------------+
```

There are other functions for appending data to the JSON column, inserting new data, removing old data, and for getting metadata on the JSON data. All the functions' details are in the MySQL Manual[1].

So What are the Drawbacks?

For decades, database folks have been telling you to normalize your data—cut it up into similar groups like street address, credit card number, postal code, etcetera. But now, with the JSON data type, you can put an entire document in one column of a table. This breaks many database rules.

This means you have to do some extra work to get your JSON columns to work with non-JSON columns in your MySQL instances. The JSON column cannot be indexed directly, which means searching through all those documents in the table can be slow. Unless you know the trick!

1 MySQL JSON Functions: http://phpa.me/mysql57-json-functions

The Trick for Fast JSON Data Searches

Generated columns can be created from JSON data, and these columns *can* be indexed for fast SQL query searches. The data is extracted from JSON and materialized in a new column. Now you can compare non-JSON columns with the JSON data in the generated columns. Pretend that the `c` key/value pair has higher importance than other key/value pairs, and as such we will want to use Structured Query Language (SQL) to quickly find the records we desire.

```
ALTER TABLE phparch
   ADD Ctype char(2) AS (stuff->"$.c") STORED;
```

The above will create a new column and plug in the value from the `c` key in the `stuff` column. This means we can have a high-performing query like:

```
SELECT stuff FROM phparch WHERE Ctype = 3;
```

There are two types of generated columns. The first form is known as a *virtual generated column* and computes values of the column on demand when the column is read. A use case would be multiplying the price of an item by a sales tax rate. You know the price and you know the sales tax, so multiplying the two and storing the product could be seen as wasteful compared to just having it calculated only when needed. The second form is known as a *stored column*. In this case, the data is stored in its own column. Indexes on JSON data require the use of stored columns.

Other Things to Watch Out For

There are some limitations you should be aware of when working with JSON columns. First, JSON columns cannot have a default value.

How large a JSON document be stored? The `max_allowed_packet` size server setting regulates the size of your JSON column. The maximum as set by the MySQL protocol is 1GB. Yeah, you think you can live with that one-gig limit now, and years ago many swore you would never need more than 640K in a personal computer.

Plus, this is rather new to the relational database world. SQL Server and PostgreSQL have also added JSON data types and the standards are still in flux. The ability to cram unstructured JSON data into a column of structured data is a new thing and mixing implementations is not going to be pretty for a while. Other relational databases are sure to follow. If your application is meant to support different database backends, you may want to hold off on using JSON columns or be aware that you may need to tweak your SQL statements for each one.

Conclusion

Now you can have the best of both worlds—structured data and unstructured data all in the same database all at the same time. Not only can you augment your normalized data with additional, unstructured attributes, but you may not need a dedicated NoSQL server in your infrastructure.

Dave Stokes is a MySQL Community Manager for Oracle and previously was the Certification Manager at MySQL AB/Sun/Oracle. He started using Personal Home Page about the same time MySQL 3.23 was released and soon after gave up plans for global domination. @stoker

Easy Audio and Video Manipulation with FFmpeg

Matthew Setter

Wouldn't it be nice to be able to script the workflow, using something that takes the raw files and does it all for us—preferably in PHP? This month, I'm going to show you how to do just that, by using the php-ffmpeg library and the open-source package FFmpeg. We'll be able to script this functionality, so that we can do it repeatedly, whenever we need to, without relying on applications like iTunes, Windows Media Player, or VLC.

What would the Internet be without audio and video? Likely, it would be a medium that never took off or became an integral part of everyday life. And it's not just the Internet where audio and video are essential. What about the exploding world of podcasts? What about online tutorials and screencasts, spectacularly realized in the PHP world with Laracasts? What about creating compilation videos of the recent holiday, or storing all our music in MP3 format?

Without audio and video, the modern world would be a much sadder, quieter place than it is today. Sure, there is a range of websites and services that allow us to convert or transcode a video or audio file across a range of formats. We can download and install applications that can also extract slices of video or scale, rotate, and merge them together.

If you're anything like me, automating this process probably sounds enticing. To put it in context, I regularly create online training courses and produce two episodes a month of my podcast, http://freethegeek.fm. I've used a range of software and services, including Audacity[1], and auphonic[2], to perform all the post-processing I need.

What is FFmpeg?

But first, what is FFmpeg? Drawing directly from the project's website, FFmpeg[3] is:

> *A complete cross-platform solution for recording, converting, and streaming audio and video.*

Going further, FFmpeg can *decode*, *encode*, *transcode*, *mux*, *demux*, *stream*, *filter*, and *play* almost any video or audio format in existence. It's available on all major platforms, including *Mac OS X*, *Microsoft Windows*, *BSD*, and *Solaris*.

That's a lot to take in if you're new to working with audio and video formats other than listening to or watching them. So here's an explanation of some the terms mentioned above:

- **Transcode**: Change a file from one format to another, such as converting an MP3 to a WAV file.
- **Mux (multiplex)**: Combines different types of data in a single stream or file.
- **Demux (demultiplex)**: Split a video and audio into separate files.
- **Stream**: Listen to music or watch video in 'real time' on a different device.
- **Filter**: Apply some form of processing to the file. There are three types of filters: pre, post, and intra. Pre-filters run before encoding. Post filters run after encoding. Intra filters run during encoding.
- **Encode**: Digitize video or audio information into different video or audio standards from a source.

Installing FFmpeg

As the library we'll be using is, in effect, a wrapper over FFmpeg, we have to install the FFmpeg binaries first. Here's how to do that, depending on whether you're running *Mac OSX*, *Linux*, or *Windows*. If you're on Windows, then download the build for your version of Windows from Zeranoe FFmpeg[4]. They have options for 32- and 64-bit installations.

If you're on Linux, there are builds available for download, or you can use the package manager of your choice to install it. Look for a package named *ffmpeg* or similar.

If you're on Mac OSX, you can download and install a build for that as well, or use one of the package managers, such as MacPorts[5] and Homebrew[6], to do it for you.

Installing the Library

With the binaries installed, the next thing to do is to install php-ffmpeg. Like all modern PHP libraries, this can be done using Composer. To do so, from the root of your project directory, run the command:

```
composer require php-ffmpeg/php-ffmpeg .
```

This will add php-ffmpeg as a dependency to an existing project or create a `composer.json` file and add it as the first dependency, if this is a new project. Regardless, the library will be available shortly after that in the vendor directory.

1 *Audacity:* http://www.audacityteam.org
2 *auphonic:* https://auphonic.com
3 *FFmpeg:* https://ffmpeg.org
4 *Zeranoe FFmpeg:* https://ffmpeg.zeranoe.com/builds/
5 *MacPorts:* https://www.macports.org
6 *Homebrew:* http://brew.sh

Converting a Stereo Track to Mono

Now that php-ffmpeg is installed, let's see how to convert a stereo track to mono. If you're not familiar with the difference between stereo and mono, when you're listening to most podcasts you're listening to audio in mono. When you're listening to a recording of a symphony orchestra, you're listening in stereo.

The difference is that in mono, the audio plays the same thing at the same time across all speakers, whether there is one or several speakers. In stereo, different audio tracks can be heard at different times from different speakers. For example, you might hear a guitar solo on your left side, and the lead singer on your right.

Let's say I've accidentally recorded my podcast in stereo format, and I want to change it to mono. This can be important as there will be extra audio information in the stereo file, making it significantly larger than it needs to be. Let's see how to convert it, and reduce the file size.

```
<?php

require_once('vendor/autoload.php');

$ffmpeg = FFmpeg\FFmpeg::create();
```

First off, we need to include Composer's autoloader and create a new `FFmpeg` class instance. There's no requirement to differentiate between audio and video on instantiation—the `FFMpeg` class can handle both types of files.

The static method will scan your system paths looking for the required binaries. Your package manager may have put them in a non-standard location, so if they're not found, you can use one or both of the following configuration options to tell the library where to find them.

- **ffmpeg.binaries**: the path to FFmpeg
- **ffprobe.binaries**: the path to FFprobe

```
$audio = $ffmpeg->open('audio.mp3');
$format = new FFmpeg\Format\Audio\Wav();
$format->on('progress', function ($audio, $format, $percentage) {
    printf("%s%% transcoded\n", $percentage);
});

$format->setAudioChannels(1);

$audio->save($format, 'track.wav');
```

With an `FFmpeg` object instantiated, let's convert the file. In the code above, I first open a sample audio file I created, called `audio.mp3`. I then specify a format to convert the file to, in this case, WAV.

Out of the box, php-ffmpeg supports *AAC*, *FLAC*, *MP3*, *OGG-Vorbis*, and *WAV*. To see what formats your installation supports, from the command line, run:

```
ffmpeg -protocols
```

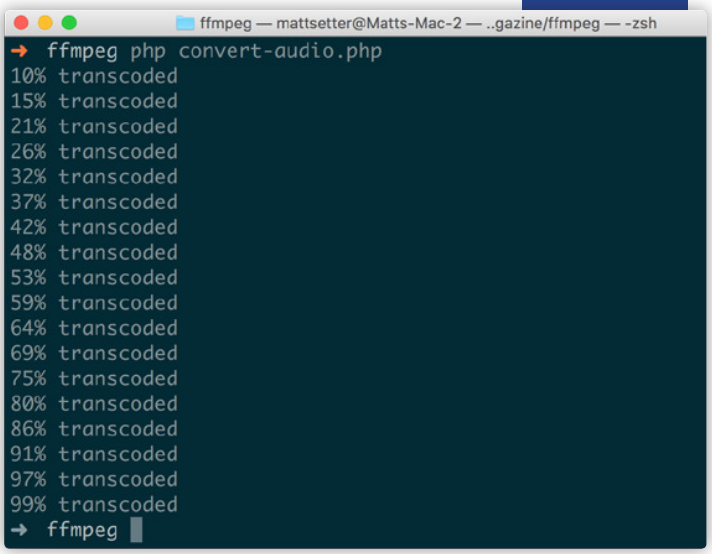
That way if an error occurs, you can more readily make sense of it and install any extensions or codecs, as needed.

With the format object instantiated, I then add a listener on the progress information provided by the format object. Specifically, I'm listening to the progress event, by specifying the name of the event (that is, progress) and a callback to handle the event.

I do this because I want to print out the progress information—it's handy to keep track of how much progress has been made, and gives an indication of how much more time is required. Much preferable to watching a black screen with no indication.

After that, I set the audio channel to 1, which in effect creates a mono audio track. Finally, I save the file, passing in the `$format` object and the file name to save the new file as `track.wav`. In Figure 1, you can see example output, showing the progress as transcoding progresses

FIGURE 1 php-ffmpeg transcoding output



Reduce the Bit Rate

Now that we can transcode a file to another format, what about changing the bit rate? According to the BBC, bit rate[7] is defined as:

> *… how many bits of data are processed every second. Bit rates are usually measured in kilobits per/second (kbps).*

The higher the bit rate, the greater the quality, and size, of the audio file. Conversely, the lower the bit rate, the lower the quality, and the smaller the accompanying file size. You can think of it a lot like working with images for the web. How much quality do you need? How high (or low) a level of quality is acceptable?

This is just as important for audio files, especially given the fact that not all countries have low-cost/high-data plans. And for some of those that do, the infrastructure may not always support streaming large files at the rates we desire.

So to ensure that we consider a broad spectrum of users, we're now going to reduce the bit rate of the audio file we used previously to 64Kbps, which is standard for podcasts. If you're interested in finding the right bit rate for your audio file, check out this post from Richard Farrar[8]. In general, lower bit rates are acceptable for conversations and discussions, while high bit rates will make music and performances sound richer.

To lower the bit rate, we only need to add one further call on the

7 *BBC: Bitrate definition:* http://phpa.me/bbc-bit-rate

8 *Richard Farrar:* http://www.richardfarrar.com/choosing-bit-rates-for-podcasts/

`$format` object, which you can see below.

```
$format->setAudioChannels(1)->setAudioKiloBitrate(64);
```

The rest of the code remains the same. Now this won't be a fair comparison of file sizes, as WAV files are a lot larger than equivalent MP3 files.

So to give you an apples-to-apples comparison, I converted an MP3 file with a bit rate of 192 kbps to 64 kbps: the file size dropped from 86mb to 29mb. Not bad, considering that the two files, to the listener, would likely sound almost identical.

Extracting an Image from a Video File

Now that we've played around with audio files, let's work with some video files. We'll ease into it by extracting an image from a video file at an arbitrary point in its timeline.

There are several reasons you may be interested in doing this, including having a thumbnail for the video that the user can see before they start playback. Let's say that's what we're doing. To extract an image we only need the code below.

```
$video = $ffmpeg->open('video.mp4');
$frame = $video->frame(
    FFmpeg\Coordinate\TimeCode::fromSeconds(2)
);
$frame->save('image.jpg');
```

Here, as before, we first open the file. We then call the `frame()` method, passing in a `TimeCode` object, which specifies the time at which to extract the exerpt. Finally, we call `$frame`'s save method, passing in the filename to save the image as `image.jpeg`. Try this with a video of your own, and pick an arbitrary point in the timeline where you want to extract a frame.

Resizing a Video File

Now that we've extracted a frame, let's do something slightly more complex, and scale a video down to a smaller resolution. Let's say that we have a video site, something like Laracasts[9], and we need to create smaller versions of our video for a preview page.

Our original video, the one that will be viewed by users when they're watching the screencast, has a resolution of 1504 x 846, but that the thumbnail video will have a resolution of 320 x 240. To do that, we can use the code sample below.

```
$ffmpeg = FFmpeg\FFmpeg::create();
$video = $ffmpeg->open('video.mp4');
$video
    ->filters()
    ->resize(new FFmpeg\Coordinate\Dimension(320, 240))
    ->synchronize();
$video->save(new FFmpeg\Format\Video\X264(),
        'video-320x240.mp4');
```

Here, we've opened the file, applied a resize filter to it, specifying the dimensions to resize the video to, and then saved it again, with a name containing the dimensions of the file.

Notice that we've also made a call to the `synchronize()` method. While this is not necessary in this example, it's worth mentioning, because it ensures that there's no lag between the video's video and audio content.

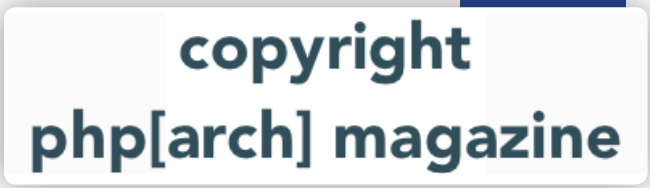
9 Laracasts: http://www.laracasts.com

Adding a Watermark to a Video File

Continuing with the screencast analogy, let's say that you want to add a watermark to your videos so that it's harder for people to rip off your work and pass it off as their own.

To do that, you could add a watermark like the one in Figure 2, likely at the bottom left or right of the video, showing your name or your company's name.

FIGURE 2 A video watermark

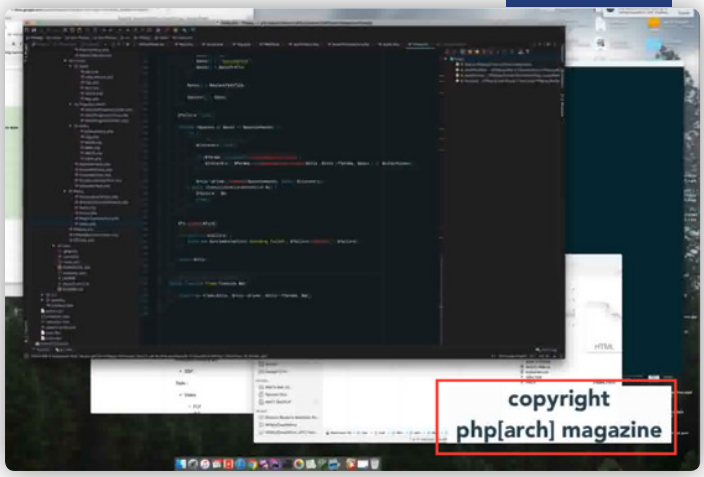


To do this, we can use the same code as before, replacing the call to `resize()` with a call to `watermark()`, as you can see in the code sample below.

```
$video->watermark('./watermark.png', [
    'position' => 'relative',
    'bottom' => 50,
    'right' => 50,
])
```

Here, we've specified the location of the watermark file, along with the details of its placement. When we run the script again, the video will now have a small watermark on it, which you can see in Figure 3.

FIGURE 3 A watermarked video file



Transcoding a Video File to Several Formats

Let's finish up by seeing how to transcode our video to several different file formats. To do that, we only need to call video's `save()` method for each format that we want to save the file to.

To each call to `save()`, we pass in a video format object and a filename. Currently, php-ffmpeg supports five formats: *OGG*, *WebM*, *WMV*, *WMV3*, and *X264*. Let's take a look at how to save the file to *OGG*, *WMV* and *WebM*.

```
$video
    ->save(new FFmpeg\Format\Video\OGG(), 'video.ogg')
    ->save(new FFmpeg\Format\Video\WMV(), 'video.wmv')
    ->save(new FFmpeg\Format\Video\WebM(), 'video.webm');;
```

We've already seen this in action, in the earlier example, where we resized the file. But it never hurts to be explicit.

Conclusion

That's your introduction to using the php-ffmpeg library and FFmpeg, for manipulating audio and video content. If you're a podcaster or screencaster like I am, then do yourself a favor and read up on both FFmpeg and php-ffmpeg to see how you can script up various parts of your workflow, saving time and effort and making your work much more consistent.

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.

PHPeople

Cal Evans

This month I am outsourcing the community column to a friend of mine, and a friend of the PHP community, Amanda Folson. Amanda disguises herself by day as a Developer Evangelist at @gitlab. When she's not helping spread the word, though, her secret identity is @AmbassadorAwsum on Twitter, where she drops wisdom bombs on a regular basis. Give her your attention, please. Have no fear, I'll be back next month with another old man rant from my lawn chair here in God's waiting room.

Community is my passion—so much so that I'm trying to make a career out of it. While I'm a member of and contributor to several communities, I've always had an affinity for the PHP community for a few reasons.

First, the resources provided by the community are fantastic. From documentation to code samples to libraries, you're sure to find plenty of resources to get you get started. If you're having trouble with a snippet, there are countless mediums that offer access to some brilliant people who are happy to give you a hand. If you're looking to solve a technological problem, chances are someone else has already solved it and put it on Packagist for you to use free of charge.

Second, the community is quite diverse. Day to day I interact with people from all over the world with varying backgrounds and experiences. To me, it's fascinating to hear their perspectives regarding technology and current events. And I've never once felt like our community's diversity was artificial.

Third, the events put on by the community are topnotch. As someone who travels to 30+ conferences per year across multiple communities, PHP events always stick out to me as being noteworthy. On a smaller scale, meet-up groups have also been exceptional when I've attended. It's always obvious to me that these events are labors of love put on by passionate groups.

All of the reasons I love the community come down to one thing: the PHPeople. I absolutely love the people who have chosen to take part in the community. I am someone who interacts with various communities on a daily basis via work and the PHP community has always stood out as having some of the most ardent people.

And that's why it kills me to see some of the things our members say to each other when things get heated in the community. Vitriol isn't exclusive to our community by any means, but I am someone who is invested in our success as a community and it stings a little even when I'm not the target. It's terrible to watch people tear each other apart over things that seem so minor sometimes.

A lot of us are guilty of it. I myself am regularly guilty of negative thoughts, words, and actions when I encounter something that I disagree with or dislike. Sometimes it's far too easy to focus on the bad instead of the good. Is it human nature? I don't know. But I *do* know that we can do better.

When emotions overrule logic and the primary communication medium is text via the Internet, it's easy to lose sight of the fact that there are real people on the other end of our hurtful communications. In moments like this, it's even easier to forget about all of the things that make this community such a wonderful thing to be a part of.

We've come so far in the last few years. We have PHP 7, we have a plethora of frameworks and libraries that exist to make our lives easier, and we're one of the top open-source projects (and arguably one of the most widely adopted). We owe it to ourselves and our fellow community members to do better.

Let's do better. Let's be better than we were yesterday. At the end of the day, there are so many fantastic projects to get involved in, so many wonderful user groups to help, and so many people who simply want to keep calm and ship code. Let's not allow a little conflict to derail us from Getting "Stuff" Done.

– Amanda Folson

Past 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

Upcoming Events

June

PHP South Coast 2016
June 10–11, Portsmouth, 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

PHPConf.Asia 2016
August 22–24, Singapore
http://2016.phpconf.asia

September

PNWPHP 2016
September 15–17, Seattle, WA
http://pnwphp.com/2016

DrupalCon Dublin
September 26–30, Dublin, Ireland
https://events.drupal.org/dublin2016

PHPCon Poland 2016
September 30–October 2
http://www.phpcon.pl

October

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

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

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

ZendCon 2016
October 18–21, Las Vegas, NV
http://www.zendcon.com

November

TrueNorthPHP
November 3–5, Toronto, Canada
http://truenorthphp.ca

php[world]
November 14–18, Washington D.C.
https://world.phparch.com

December

SymfonyCon Berlin 2016
December 1–3, Berlin, Germany
http://berlincon2016.symfony.com

ConFoo Vancouver 2016
December 5–7, Vancouver, Canada
https://confoo.ca/en/yvr2016

PHP Conference Brazil 2016
December 7–11, Osasco, Brazil
http://www.phpconference.com.br

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

The Art of Transduction

David Stockton

As software developers, we will run across new problems all the time, often with aspects similar to those of previous problems we've encountered. In these cases, we typically write similar solutions. Many times you've probably needed to process a bunch of data, whether an array, a stream, or a database query result set. Often, the go-to tool to solve these problems is the humble `foreach` loop. PHP provides other ways for certain cases that are faster than `foreach`, yet I'd guess they are used only a fraction of the time they could be.

Processing via Foreach

The common pattern I typically see is to get data from some source, usually as an array, loop over it via `foreach`, and either change it, output it, filter it, or perform some combination of these in order to massage the data and get it all fixed up how you want. It's relatively quick and often easy to understand, but there are better ways.

Suppose you have a CSV data file that's uploaded and you need to import the values to the database, but the included date field is not in the format your database expects. It's `foreach` to the rescue, right?

```
foreach ($rowData as $row) {
    $date = \DateTimeImmutable::createFromFormat(
        'm/d/Y', $row['created_date']
    );
    $row['created_date'] = $date->format('Y-m-d');

    // Write to database or whatever
}
```

More often than not, I see the above pattern, but the whole array is manipulated before anything is written to the database or to the output file. This leads to slowdowns due to processing the array multiple times, as well as memory usage that is higher than what is really needed.

PHP's Array Functions

Now, the `foreach` solution is fine—it works, and it can be relatively quick assuming you're not iterating over the whole set of data to fix it and then going back over it again to load it into the database, but the problem is that every part of it is "user land" code. PHP has some built-in functions that put the looping part in native (read: built in C) code, with the processing part provided by the programmer. That means the above bit becomes something like this:

```
array_walk($rowData, function($row, $index) {
    $date = \DateTimeImmutable::createFromFormat(
        'm/d/Y', $row['created_date']
    );
    $row['created_date'] = $date->format('Y-m-d');

    // Write to database or something
});
```

The `array_walk` function takes a *callable* and applies it to each member of an array, so it's great when you're able to manipulate and process in one shot.

There's a bunch of other similar built-in array functions for different situations, each of which will do their particular job faster than a `foreach` loop would.

There's `array_filter`, which is great for looping over an array and deciding if you want to keep a value in the array or discard it. It takes a callback that returns a `true` value to keep the row or `false` to discard. Suppose we have an array of objects that can indicate if they are active or not, and we want to filter down to just the "active" objects.

```
$objects = $this->getObjectsFromSomewhere();
$activeObjects = array_filter(
    $objects,
    function ($value) {
        return $value->isActive();
    }
);
```

At the end of the filter routine, the `$activeObjects` array will contain only the objects which returned a "truthy" value from the `isActive()` method.

Next up is `array_map`. It will change each row in the array according to the callback function. Here's an example of using the `array_map` to take an array of numbers and turn it into an array of the squares of those numbers.

```
$square = function ($value) {
    return $value * $value;
}

$squares = array_map($square, [1, 3, 5, 8, -2]);
```

The result of this function will be that the `$squares` array will contain `[1, 9, 25, 64, 4]`. There's one final PHP array function I want to talk about before jumping into the meat of the article.

The `array_reduce` function is used to move through an array and end up with a single value at the end. An example would be iterating over a shopping cart full of items to determine a total price, concatenating strings together, or any of a number of other common operations. Let's take a look at an example of totaling up shopping cart items:

```
$total = array_reduce(
    $shoppingCartItems,
    function($sum, $cartItem) {
        return $sum + $cartItem->getPrice();
    },
    0.00
);
```

Since this one looks a bit different, I'll take a minute to explain. As with all previous examples, the first argument is the array we're iterating over. The callback is a bit different, though. We have the first argument, `$sum`, which is a *carry over* value. Each time the callback is executed, the return value of it will become the `$sum` value for the next time it is executed. You can almost think of the body of the callback as being:

```
$sum += $cartItem->getPrice();
```

which is probably exactly what you'd write if you wanted to use a `foreach` loop to total up the values. The final argument, `0.00`, is used as the initial value. It will be passed in as the initial value of `$sum`. Once the function is done running, the `$total` value will contain the sum of all the items in the cart.

Dealing with all data as arrays in PHP is certainly possible, but it's often not the best way. Data in arrays means that everything needs to be kept in memory, and there's a limit to that. If you need to load large files of data, loading everything into memory first may take too long, depending on the file size, and your system may not be able to handle it due to memory limits. You may also have a need for a series of data which doesn't have an end—think of a cycle that goes through the days of the week.

If you can process the data a row at a time, it means that you'll be able to essentially process files as large as you want. Your code will have a known and predictable memory footprint even though the files or data you're working with may be extremely large.

Introducing Transducers

The name "Transducers" is a portmanteau of "transform" and "reducers." In short, they are functions that take in a reducing function and return a different reducing function. Transducers as a concept comes from Rich Hickey, the creator of the Clojure programming language.

In functional programming, which I'm not going into too deep here, there are three major functions or concepts: map, filter, and reduce. You can accomplish quite a lot with these. Let's take a look at how each works. Instead of working exclusively with arrays, though, like `array_map`, we want a map function that can work with any kind of iterable. This means not only arrays, but also generators and iterators.

Map is going to take a function defining the transformation we want and a data source.

Filter will take a function that gives back a true or false value, with `true` indicating the value should be retained and `false` meaning it should be discarded.

The reduce function will take a function defining how to take two values and end up with a single value, along with a data source and a potential initial value.

Transducers allow us to combine or compose these functions to make data processing simple and understandable.

Transducers.php

Shortly after transducers in Clojure were introduced, Michael Dowling (creator of Guzzle) started a PHP implementation of transducers[1]. You can build chains of functionality that work on iterators, generators, arrays, and Traversables. You can evaluate the results of these chains either eagerly or lazily. Eager evaluation means that all the calculations will be done before you start working with the results. Lazy evaluation allows you to stream data through the transducers and keep memory requirements down.

To follow along, you can install the package with the following Composer command:

```
composer require mtdowling/transducers
```

One of the ways I've used transducers is to transform an incoming file from a customer into a file that our current file loading system could handle. Typically, we import CSV (comma-separated values) files but they provided a TSV (tab-separated values) file of people data. Additionally, some of the values provided needed some work. The file had name fields all lowercased, the date field was in the wrong format, and for completely made-up-for-this-article reasons, we needed to know how many days until each person's birthday, or how long since their birthday if it had already passed.

Also, the file also had a header which we didn't need. Let's take a look at how we can build this out in Listing 1.

LISTING 1

```
01. <?php
02. use Transducers as t;
03.
04. /* SNIP Definition of the functions used below */
05.
06. $transformer =  t\comp(
07.     t\drop(1), // Get rid of the header
08.     t\map($convertToArray), // Turn TSV to Array
09.     $fixNames, // Capitalize names
10.     t\map($convertToDate), // Change to DateTimeImmutableObject
11.     t\map($addDaysFromBirthday), // Date math
12.     t\map($fixDateFormat) // Format DateTimeImmutable to Y-m-d string
13. );
```

We have a lot of things that are happening that we need to talk about. First of all, we're using `t\comp`, which is creating a function by composing other functions together. The functions will be executed in the order you specify. The importance of the order will become apparent soon, if it's not already.

The first function is one provided by the Transducers package. It will throw away the first row it encounters and let every following row through. This function will serve to drop the header row from the input.

Next, we need to understand that each incoming bit of data that will pass through is starting as a string with tab-separated values. We need to turn this into something we can more easily manage, like an array.

```
$convertToArray = function($tsvRow) {
    $arrayRow = explode("\t", $tsvRow);
    $columns = ['id', 'first', 'last', 'dob'];
    return array_combine($columns, $arrayRow);
}
```

1 *mtdowling/transducers.php: http://phpa.me/transducers-php*

This function explodes the row using tabs, and then combines it with another array representing the field names—what the actual columns represent. In other words, a row that comes in looking like this:

```
42   david   stockton   1/1/1999
```

will be transformed into an array like this:

```
[
   'id' => 42,
   'first' => 'david',
   'last' => 'stockton',
   'dob' => '1/1/1999'
]
```

This transformed array will be passed through the next part of our composed function, `t\map($fixNames)`. For `$fixNames`, I actually want to create a function that creates functions and use that twice to make a new composed function that will fix the capitalization on both the `first` and `last` field. As a side note, yes, I realize the futility of trying to properly fix name capitalization programmatically, but please bear with me for the example.

We could easily build a function that would capitalize a hardcoded field value:

```
function ($row) {
    $row['first'] = ucfirst($row['first']);
    return $row;
}
```

The function to fix last name would be nearly identical. We need to have a function that takes in a single value, `$row`, but where the name of the field can actually be provided as a variable. Since our map function needs a function that takes in a row, we need a different way to provide the name of the field we want to capitalize. This is where making a function that returns a function comes in.

```
// Function to return a function
$ucField = function($field) {
   return function ($row) use ($field) {
      $row[$field] = ucfirst($row[$field]);
      return $row;
   };
};
```

This code takes in a field name in `$field` and, through the use of a closure, closes around that field and returns a new function which will accept a row of data and will uppercase the first letter of whatever field we passed in. The `use` bit is where our function is able to essentially reach out into its parent's scope (the outside function) and use the value of `$field` that was passed in. It's a little bit like using PHP's global except way cooler and significantly less disgusting. Now we can make two new functions with this and compose them together:

```
// Map composition
$fixNames = t\comp(
   t\map($ucField('first')),
   t\map($ucField('last'))
);
```

We're using the `t\comp` again to create a new composed function that will work on both the first and last name fields. This is assigned to the variable `$fixNames`. We could have just added the two calls to `t\map` directly to our transform, but I wanted to demonstrate that you could compose these functions from other composed functions, making the final code more readable and understandable while leaving out a lot of clutter and repeated code. You could follow the same pattern with more complex operations as well.

Next up, we're running another map function in order to convert the invalid `month/day/year` provided format into a more standard `year-month-day` format. For right now, we're going to just replace the string date representation with a `\DateTimeImmutable` object since we'll need it in a moment. Here's the `$convertToDate` function:

```
$convertToDate = function ($row) {
   $date = DateTimeImmutable::createFromFormat(
      'm/d/Y',
      trim($row['dob'])
   );
   $row['dob'] = $date;
   return $row;
};
```

This one is also pretty straightforward. By using the DateTimeImmutable's `createFromFormat` function, we can convert it into an object that makes it easy to do date/time math. I threw in the `trim` call because in my sample data, when the date field (the last in the row) is converted to an array, the date field has a retained newline character. This was causing the `createFromFormat` function to fail parsing the date. We don't want that, and `trim` happily removes it. As before, we're simply placing our transformed value back into the row that was passed in, remembering to return it. If we don't return a value from a map function, the value sent to the next function will be `null`, which is probably not what we want.

At this point, we have our original data, but it's been transformed from TSV into an array, the first and last name fields have had their first letters' capitalized, and the `dob` field is converted to an actual `\DateTimeImmutable` object. All that's left is to determine how many days from or until the person's birthday and attach that data to the row and then turn the object back into a string.

Since we've kept the `dob` field as an object, we don't need to read or parse the value again—we can just use it. This means the `$addDaysFromBirthday` function looks like Listing 2.

There's a bit more going on here, but none of it is terribly complicated. The first line is creating a `\DateTimeImmutable` object representing right now. We use that object to extract the current year on the next line. Since we want the same value for all of our calculations, there's no point in creating and recreating the `$now` object within the map function, so I'm creating it outside the function and then using a closure to bring it into the function.

LISTING 2

```
14. <?php
15. $now = new DateTimeImmutable();
16. $thisYear = $now->format('Y');
17.
18. $addDaysFromBirthday = function($row) use ($now, $thisYear) {
19.    $dob = $row['dob'];
20.    $birthday = DateTimeImmutable::createFromFormat(
21.       'Y-m-d',
22.       $dob->format("$thisYear-m-d")
23.    );
24.
25.    $timeUntilBirthday = $now->diff($birthday);
26.
27.    $row['time_until_bday'] = $timeUntilBirthday->invert
28.       ? $timeUntilBirthday->format('%m months, %d days ago')
29.       : $timeUntilBirthday->format('%m months, %d days');
30.
31.    return $row;
32. };
```

The `$addDaysFromBirthday` function will have access to both the `$now` and the `$thisYear` values and they'll only need to be calculated once.

Next, we're extracting the `dob` field from the provided `$row` data. We determine the value of the person's birthday by replacing the year they were born with the current year and creating the `$birthday` object. Next, the `$timeUntilBirthday` object is created by determining the difference between now and the birthday. If the `invert` flag has been set, then we know that for this year, that person's birthday is in the past. Otherwise, it's still coming up. The format commands will create a nice description of how long until the person's birthday, or how long since they celebrated. This new data is placed into a new field called `time_until_bday` and the whole row is returned.

The data row now includes this string. Finally, we need to convert the `\DateTimeImmutable` object back into a date string that's in the format we need. The `$fixDateFormat` function is pretty straightforward then:

```
$fixDateFormat = function ($row) {
   $row['dob'] = $row['dob']->format('Y-m-d');
   return $row;
};
```

We simply replace the value in the `dob` field (the object) with a formatted string representation of that date. Then we return the row.

All these functions are composed together into one that will do all the work we need to convert from an incoming TSV file to an array, fixing name fields, properly formatting date fields, and adding new data about the number of months and days until or since a person's birthday. But so far, we've not actually processed any data.

The transducer signature we need to look at will take the composed function and a data source. Since I'd like to provide the data from the file via a generator to keep memory usage down, and process the outgoing file in the same way, I want the result to be an iterator. This means the transformation pipeline we created will be lazily evaluated—that is, each row from the file will be processed individually when we iterate over it.

First, we'll need a way to read the data and provide it via a generator:

```
$fh = fopen(__DIR__ . '/phpArchData.tsv', 'r');

$reader = function () use ($fh) {
   while ($row = fgets($fh)) {
      yield $row;
   }
};
```

This code opens the TSV file in read mode with `fopen`, and uses that file handle to create a generator that will yield a single line of that file each time it is iterated. Once the file runs out of data, the generator will stop.

All that is left is to iterator over our transforming composite function with the data and do something with it.

```
$format = "[%d] %s %s - %s (%s)\n";
foreach (t\to_iter($reader(), $transformer) as $data) {
   echo sprintf(
      $format,
      $data['id'],
      $data['first'],
      $data['last'],
      $data['dob'],
      $data['time_until_bday']
   );
}
```

As you can see, the `t\to_iter` function receives the generator as the data source, and the transforming function `$transformer`. In this example we are just echoing out the data. The output will look something like this:

```
[0] Aurelia Hegmann - 1976-08-03 (2 months, 18 days)
[1] Ena Metz - 1979-05-12 (0 months, 4 days ago)
[2] Johathan Terry - 1977-07-21 (2 months, 5 days)
[3] Roderick Hickle - 2008-06-05 (0 months, 20 days)
[4] Tyrel Auer - 2011-11-03 (5 months, 18 days)
```

This runs quickly and the memory usage is minimal. Depending on your needs, there are other options for combining your transformation function and your data. If you wanted the type of data to remain instead of being turned into an iterator like we did above, you could replace `t\to_iter` with `t\xform`. The eagerly evaluated functions include `t\transduce()`, `t\into()`, `t\to_array()`, `t\to_assoc()`, and `t\to_string()`.

Additionally, there's a `t\to_fn()` function that will give back a function that can be used in PHP's `array_reduce` function. The package also provides a number of built-in reducing functions that can be used, and it's possible to build your own transducers as well.

Included Transducer Functions

The Transducers.php package provides quite a few transducing functions. In the table on the following page I'll briefly touch on the other included functions.

Conclusion

Transducers and thinking about functional programming allow us to process streams of data in a way that's easier to understand than a large foreach. Composing map, reduce, and filter functions allows for powerful transformation and processing chains that are easy to understand, simple to test, and extremely useful. At this point I've used this package in a couple of different projects that are in production. It reminds me a bit of how middleware works but on a more generic, less-specific-to-web-requests level. I highly recommend taking a look at transducers and playing around with functional programming. For me, it was a lot of fun. 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.

Included Transducer Functions

function	description
`map(callable $f)`	Applies the map function `$f` to each value in a collection.
`filter(callable $predicate)`	Filters (removes) values from a collection that do not satisfy the predicate.
`remove(callable $predicate)`	Removes values that do satisfy the `$predicate` function.
`cat()`	Concatenates items from nested lists.
`mapcat(callable $map)`	Applies a map function to the collection and concatenates them to one less nesting level.
`flatten()`	Takes nested combination of sequential items and returns a single flattened sequence.
`partition($size)`	Takes the source and splits into arrays of the specified `$size`. If there are not enough to evenly split, the final array will have the remainder.
`partition_by(callable $predicate)`	Splits the inputs into arrays each time the callable `$predicate` changes to a different value.
`take($n)`	Takes `$n` values from the collection.
`take_while(callable $predicate)`	Takes from the collection while the `$predicate` function returns true.
`take_nth($n)`	Takes every nth value from a sequence of values.
`drop($n)`	Drops `$n` items from the start of a sequence.
`drop_while(callable $predicate)`	Drops values from the sequence as long as the `$predicate` function returns `true`.
`replace(array $map)`	Replaces values from the sequence according to the map.
`keep(callable $f)`	Keeps items where the `$f` function doesn't return `null`.
`keep_indexed(callable $f)`	Returns the non-null results of calling `$f($index, $value)`.
`dedupe()`	Given an ordered sequence, it will remove values that are the same as the previous value.
`interpose($separator)`	Adds the separator between each item in a sequence.
`tap(callable $interceptor)`	Tap will "tap into" the chain, to do something with the intermediate result. It doesn't change the sequence.
`compact()`	Trims out all "falsey" values from the sequence.
`words()`	Splits the input into words.
`lines()`	Splits the input by lines.

May Happenings

PHP Releases

- PHP 7.0.7: http://php.net/archive/2016.php#id2016-05-26-1
- PHP 5.6.22: http://php.net/archive/2016.php#id2016-05-26-3
- PHP 5.5.36: http://php.net/archive/2016.php#id2016-05-26-2

News

Marc Scholten: Accidental Complexity Caused By Service Containers In The PHP World

In this post to his site Marc Scholten talks about something that's become a side effect of using the inversion of control design pattern in PHP applications (specifically related to dependency injection): added accidental complexity. He illustrates with an example using the Symfony services container, a piece of the framework that allows the definition of dependency relationships via a YAML formatted file.
http://phpdeveloper.org/news/24029

SitePoint PHP Blog: Localizing Dates, Currency, and Numbers with Php-Intl

On the SitePoint PHP blog Younes Rafie has continued his series about the PHP "Intl" extension for use in internationalizing an application. in this second part of the series he moves away from just strings and looks at using it for currencies and numbers. The "Intl" extension makes these operations relatively simple with plenty of built-in objects and methods to help with the translations between the formats.
http://phpdeveloper.org/news/24026

Evert Pot: Why PHP-FIG Matters

There's been quite a bit of drama lately around the PHP-FIG (Framework Interoperability Group) organization in the past few weeks, mostly resulting from an inflammatory situation involving one of the member projects. There's been questions around about the PHP-FIG, its role in the community and how that might change in the future. In this post to his site Evert Pot shares some of his own thoughts about the group and why it still matters.
http://phpdeveloper.org/news/24012

QaFoo Blog: When to Abstract?

On the QaFoo blog they've posted an article that shares some of their thoughts on "when to abstract" in your code - essentially finding that point where abstracting out functionality makes sense. They start off by defining three different types of projects (internal, library and adaptable) and move into how this type changes when/how you abstract things in your code.
http://phpdeveloper.org/news/24010

Ibuildings Blog: Working with Entities in Drupal 7

On the Ibuildings site there's a tutorial posted talking about working with entities in Drupal 7 and how creating your own classes for them can make them easier to manage. First off, he starts with a refresher on what entities are and how they relate to the database schema. He points out the difficulties in using them and testing their types.
http://phpdeveloper.org/news/23968

Zend Framework Blog: Announcement: ZF Repository Renamed!

The Zend Framework blog has a post announcing the name change of the main Zend Framework repository on GitHub. The post also includes the instructions on how to update your current "remotes" in your git checkout (so you don't have to re-clone).
http://phpdeveloper.org/news/23965

Mark Ragazzo: Immutable Objects

In a post to his site Mark Ragazzo looks at immutable objects - what they are and how they can be used in a PHP application with some "final" functionality. He starts with a list of a few things to remember when implementing immutable objects (like using the "final" keyword) and problems that can come without them. He then gets into some examples, showing how to create immutable Address and Money objects and how to use them when you need to update/get values from the object.
http://phpdeveloper.org/news/23962

Mark Baker: In Search of an Anonymous Class Factory

In a new post to his site Mark Baker take a look at anonymous classes, a new feature in PHP 7, and a challenge he took on to figure out how to apply traits to them at runtime. His first idea was to build an anonymous class, extending the requested class that would come along with the traits/properties/functionality of the original class. He includes some of the code he tried to implement this solution and ultimately figured out that a factory would be a good approach to creating the structure.
http://phpdeveloper.org/news/23955

The Iconic Engineering: Decoupling—All the Best Bits of Your Favorite PHP Frameworks.

In this post on the Iconic Engineering, Aaron Weatherall looks at the trend toward offering more decoupled PHP components and libraries. In particular, he looks at how Zend Expressive allows you to chose the routing and templating libraries to use on installation. Last, he talks a bit about the benefits of decoupling for framework devs and everyone else.
http://phpa.me/decoupled-php-frameworks

How to Write a README That Rocks

Eric L. Barnes writes in the .dev blog on the importance of your project's README file and how to structure it to be useful to new developers evaluating your project. He starts by considering the questions that your README should answer which leads to a suggested outline of the sections to make sure to include.
http://phpa.me/dev-readme-rocks

Ups and Downs of an Entrepreneur

Eli White

I have been a part of drastically different companies over the years that span my career. I've worked for the government, for nonprofits, for academics, for enterprise-level companies, and for small startups. Each of them has been a very different experience. But being a part of running and owning our own business here has been a completely different one altogether.

Doing It All

When working for small startups, you definitely get the feeling that everyone has to do a little bit everything. Jobs need done; someone needs to do them. But this pales in comparison to the demands put upon the entrepreneur. Suddenly it's not just "the coder needs to do a bit of design." Suddenly you are needed to deal with lawyers, handle government regulations, figure out how to do accounting, and negotiate with vendors on the finer points of their services.

> *The thing most people don't pick up when they become an entrepreneur is that it never ends. It's 24/7.*
>
> *—Robert Kiyosaki*

The amount of additional tasks that you need to start doing is amazing (and daunting), even if you are on a team of five people versus starting a company by yourself. Until you've truly been in charge, you will never realize all the minor (yet extremely important) details that go into running a company.

Different Mentality

I once had a boss who, when giving me a corporate credit card so that I could more easily represent the company at conferences as an advocate, told me to "treat the company's money as if it were your own."

That's really good advice, but it turns out that until that money *is* truly your own money it never quite sinks in. In my opinion, I did a great job back at that company doing so. Making sure that I was only spending money on what I deemed to be important items for the company. Things with a good value of return, short or long term, that would help the company grow.

But in retrospect, it's still nothing like having that credit card in your hand and knowing exactly how much money the company has and how much revenue is coming in. Worrying about whether you will be able to make that next bill payment, worrying about whether you can make payroll, not only this month but for the next couple of months as well.

You now are intimately tied to the success of the company, the ebbs and flows of its cashflow. A bad month can leave you worried about your mortgage, while a good month makes you feel on top of the world. You will find yourself, when running a company, to be riding an emotional roller coaster, constantly living for those downhill wild rides and then dreading what comes around the next bend.

Why?

Why am I telling you all of this? For that matter, why would you want to even consider starting your own company given all of the woes listed above. Simply put, it's important for everyone to understand what they are getting into. All of us here at php[architect] are "accidental entrepreneurs." We didn't aim to start a media brand but kind of sideways fell into it along the way.

The thrills of truly being in charge, not just the manager, but truly being an owner of the company and making the decisions that mean life or death (for the company) is amazing. Choosing to do something or not do something, knowing the consequences and owning them. It's a freedom that you can't get in any other way. Yes, it comes with semi-constant concerns over your bank account and worry about one thing going wrong and shutting you down.

But it's absolutely worth it.

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. He seems to like being an entrepreneur so much, that he's now founded three different companies in his lifetime. @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