www.phparch.com

July 2016
VOLUME 15 - Issue 7

php[architect]

harnessing magic

ALSO INSIDE

Community Corner:
Nacho Cheese

Leveling Up:
Simple is Better

Security Corner:
After the Breach

Education Station:
Running Mailing Lists with MailChimp and PHP

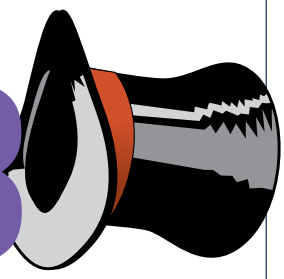
finally{}:
Ageism in the Development Community (from Both Sides)

php[architect]

JULY 2016
Volume 15 - Issue 7

CONTENTS

harnessing magic



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

Harnessing Magic

After working with computers for a while, you realize that programming is a peculiar field of endeavor. Coding is often portrayed as magical. Its practitioners are thought to possess some secret knowledge and by some complex series of incantations can make miracles happen.

Even if you know how computers and programming work, I'm sure there are aspects of understanding that elude each one of us. I know for some people it's regular expressions. Personally, I have a hard time wrapping my brain around the terms used in functional programming as well as the shift in thinking required from traditional object-oriented programming. And just how compilers really work to transform text to machine instructions seems like some deep magic to me.

But, as you study and practice, the real magic will happen. Glimmers of understanding and those "Aha!" moments are what drew many of us to tinker with computers in the first place. Those moments can happen all the time as you challenge yourself to learn something new and when you share your new found knowledge with friends.

> *Any sufficiently advanced technology is indistinguishable from magic.*
>
> *- Clarke's Third Law*

In this issue, we've collected features that will help you demystify some arcane topics, maybe even one that you've been avoiding learning until now. In *Regex is Your Friend*, Liam Wiltshire will show you when to use a regular expression and when not to use one. He'll explain how to use conditional matches and how to improve regex performance. David Corona will explore *Removing the Magic with Functional PHP*. Instead of relying on conventions and depending on other classes to work their magic, you'll see how to use functional programming to write clearer, easier to understand code. In his third feature on security, Edward Barnard gives advice on *Implementing Cryptography*. Cryptography certainly seems like some magical math stuff that helps keep our data secure, but doing it correctly can be really tricky. He'll share the pitfalls to avoid. In *Reference Counting : The PDO Case Study*, Gabriel Zerbib studies the dark art of PHP references and how they are juggled by the Zend Engine. A good understanding of how they work will help you understand how memory is managed when your code executes.

Our columns this month are also full of magic. In *Education Station*, Matthew Setter looks at automating email in *Running Mailing Lists with MailChimp and PHP*. He'll show you how to use a straightforward Mailchimp client to perform tasks outside of the service's UI. David Stockton will share his tips for writing programs that are easier to understand and debug in *Leveling Up: Simple is Better*. In *Community Corner*, Cal Evans explains what *Nacho Cheese* has to do with your contributions to open-source projects. This month's *Security Corner* has advice from Chris Cornutt on what to do *After the Breach*. He looks at the practices and tasks that will help you deal with being hacked. *finally{}*, Eli White discusses *Ageism in the Development Community (from Both Sides)* and has advice for you if you are just starting your career or are a seasoned greybeard.

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/July2016_code

Implementing Cryptography

Edward Barnard

Cryptography is extremely difficult to get right. It's a way to make news headlines when you get it wrong. Unfortunately, some of the online examples for PHP are obsolete or flawed. Here we give you a concrete place to begin: Learn what you need to know about randomness, and learn to correctly encrypt and decrypt a string. Further your learning with the Additional Reading list at the end of this article.

Use the Encryption Library

In theory, you should never have to "roll your own" encryption code. Experts will point you to the PHP encryption libraries. The library should "just do it," freeing you from needing to know anything about encryption.

In many cases, it's just not that simple:

- Third-party integrations may force you to "roll your own" encryption to meet third-party requirements.
- Your production environment may not support the newer libraries, or have access to certain PHP extensions.
- Your legacy code base may contain "roll your own" encryption code.
- With web services, both server and client may need access to the same library, forcing you to "roll your own" code using that library.

We'll be using OpenSSL[1] for this article. OpenSSL is commonly available to both server and mobile app developers.

You are likely thinking that if we are using OpenSSL, we are not "rolling our own" cryptography. On the contrary, that means we *are*. The problem is that OpenSSL only has the basic low-level tools: encryption, decryption, HMAC. It's awfully easy to make the wrong choices when using OpenSSL.

Even if you can use end-to-end cryptographic libraries, you still need to understand the basics. Because it's so easy to get it wrong, it's good to know that you *are* on the right track. PHP 7, incidentally, provides some much-needed cryptographic tools.

This article is about some basics. We'll cover randomness, encrypting a string, and decrypting a string.

Ashley Madison

Ashley Madison[2] made headlines by getting it wrong. They thought they got it right, but forgot about their earliest users. Once again, it's awfully easy to get it wrong!

Strong password hashing means you have protected against known attack techniques. The general idea is that it takes an attacker too much real time to crack each password to make such attacks practical.

Hashing generally refers to a one-way mapping of an input value to a hash value. You store the salted and hashed password. You can't ever derive the real plain-text password from that hash. Instead, when a user presents their password, you hash *it* and see if it matches the stored (and salted) hash. You have no way of knowing if the presented password is close or not; it matches or it doesn't.

The rest of this article is about symmetric encryption. We'll be using the same secret key to encrypt and decrypt our message. It's important for you to understand password hashing, but that topic is outside the scope of this article[3].

Working with Encryption

If you're considering doing encryption, you're probably either doing data storage and retrieval, or you're transmitting information from one place to another. If you're encrypting data for later retrieval, you have control of both encryption and decryption. That's relatively easy.

What's more difficult is when you have two different developers, organizations, or platforms at the two ends of the transmission. For example, your web services API is passing encrypted data between server and client.

The problem is this: You can't know if you got it right until you decrypt the string. If you're able to decrypt the string, great. If not, you have no way to know what went wrong. Did you call the decryption function correctly? Do you have the right secret key? Did your encrypted string get truncated or mangled? Was the encryption wrong to begin with? You don't even know where to *start* looking.

I once had a frustrating problem developing some web services. The server-side encryption was responding correctly from my development environment, but rejecting all client requests in production. Needless to say, neither I nor the client-side developer were happy.

In this case, it came down to PHP's `mbstring` (multi-byte string) library/extension. I was using the library to chop apart raw binary secret keys. My development environment had an up-to-date library because we'd installed it as a dependency for PHPUnit (my unit-testing tool). I don't run PHPUnit in production, and it

1 *OpenSSL: https://www.openssl.org*

2 *PC World: Ashley Madison coding blunder made over 11 million passwords easy to crack: http://phpa.me/am-11M-hacked*

3 *PHP's online manual is a good place to start if you need to handle passwords: http://php.net/password*

therefore never occurred to me to install an obscure PHPUnit dependency.

To make things worse, I did have an old version of `mbstring` installed on the production servers. It simply didn't work for slinging around raw 256-bit encryption keys.

All of my server-side unit tests ran perfectly, because they all ran in the (correct) development environment. All of my server-side production tests also ran perfectly. The server was doing both encryption and decryption. It was doing it wrong, but it was doing it wrong the same way in both directions.

Can you imagine our frustrated client-side developer trying to explain that something is wrong, when all server-side tests pass *and* all app-side tests pass?

How did we find the problem?

I dumped out intermediate results for every step of the encryption process. Since it's raw data, I dumped each step out as printable hex digits and base64. Our app developer created unit tests which confirmed that each step matched my generated values. They did.

This was no surprise. We already knew we were getting the right encryption in my development environment. I moved the same dumps to production. Suddenly I was seeing strings of zeroes where they shouldn't be! That's where my PHP code was using PHP's multi-byte functions. I found the library discrepancy and re-dumped those intermediate steps. All app tests now ran green (passing) against our production servers. Victory!

My point in this story is showing you that working with encryption is tough by design. If something goes wrong with the cryptography, *no* information gets leaked as to what might have gone wrong. We don't want to guide an attacker trying to break our security. Unfortunately, the situation is equally opaque to you as the developer trying to get things working. You can see how managing dependencies correctly is critical here.

Randomness

You need randomness because you need to keep secrets. If a secret is easy to guess, it's not much of a secret.

The measure of randomness is called *entropy*. Entropy is the degree of uncertainty about something. Years ago I would ask my daughter to pick a number between 1 and 10. She'd always pick 7, because it was her lucky number. There is not much uncertainty (entropy) contained in her "random" choice.

Let's take another example. Sixteen million Model T Fords were sold. That's a 24-bit number (2**24 is 16,777,216). The first Model Ts had a few color choices. That gives us 2-3 bits of entropy within that 24-bit number. Henry Ford later stated, "Any customer can have a car painted any color that he wants so long as it is black." That drops the uncertainty (entropy) down to zero.

If we pick a number between 1 and 10, but we tend to pick even numbers, the numbers are not as random as intended. If the *distribution* of random numbers isn't even, your numbers have less entropy, since some numbers will have a higher frequency. In English-language text, for example, the letter "a" comes more frequently than the letter "z." Combinations such as "th" are far more likely to appear than combinations such as "tq." *Cryptography Engineering* (chapter 21.2) reports that English-language text only contains 1.5–2 bits of entropy per letter.

FIGURE 1

Figure 1 is a photo showing one of the sixteen million Model Ts manufactured. However, given that the people in the photo are my grandparents, the choice of photo is a lot less random than you'd think.

Using Randomness

Let's consider a real-world example. Let's do AES encryption with 256-bit keys in CBC mode. That's a real-world choice.

We need a secret key for the encryption. It needs to be 256 bits. We'll use the password `"123456"` as our secret key. No, wait, to be more secure we'll use `"12345678"`. Run the text string `"12345678"` through SHA-256 and you have a 256-bit secret key:

```
$secretKey = hash('sha256', '12345678', true);
```

AES-256 will work just fine with your 256-bit derived `$secretKey`. Would anyone be so stupid as to use `"12345678"` as their secret encryption password? All experienced attackers know the answer is "Yes."

When encryption requires something that is "random" or "unguessable" and is *X* number of bits long, that means the encryption requires *that many bits of entropy*. Given that our secret pass phrase is on the top ten list of known passwords, we have maybe 3 bits of entropy, not the required 256 bits of entropy. Would an attacker check for something that stupid? Yes, they would!

Most PHP-based random number sources fail this test. The functions `uniqid()`, `rand()`, `mt_rand()` all have issues with predictability. Even `openssl_random_pseudo_bytes()` gets it wrong[4].

On Linux systems, the correct (best available for our purposes) source of randomness is the `/dev/urandom` pseudo-device. I'm not qualified to tell you the correct source for non-Linux systems, so I won't. PHP's `mcrypt` extension includes the `mcrypt_create_iv()` function, which can draw from `/dev/urandom`. Note that this is `urandom` with a **u** and not `/dev/random` without the **u**. That's a critical difference; don't get it wrong.

Do not use the `mcrypt` extension for cryptography. To the best of my knowledge, it has no active developer support. It's effectively been abandoned, even though it remains available for PHP 4, 5, and 7. However, it does contain that portal to `/dev/urandom` that you need. The OpenSSL extension does *not* provide direct access to `/dev/urandom`.

4 *Bug #70014:* `openssl_random_pseudo_bytes()` *is not cryptographically secure:* https://bugs.php.net/bug.php?id=70014

For example:

```
$secretKey = mcrypt_create_iv(32, MCRYPT_DEV_URANDOM);
```

Again, be sure you specify `MCRYPT_DEV_URANDOM` with a **U** and not `MCRYPT_DEV_RANDOM` without the **U**. The first parameter is the number of random bytes you want. We need 256 bits, which is 32 8-bit bytes.

Now we have a 256-bit secret key to use for AES encryption that really does have 256 bits of entropy. You need to figure out a way to save that key so that it can be used by both the sender and receiver, without any possibility of an attacker obtaining the secret key. That issue is outside the scope of this article.

> *Note that PHP 7 includes a reliable built-in source of randomness. Once you make it to PHP 7, you'll find the situation has improved.*

The Session Token

I find a number of places where I don't directly use cryptography, but I do need to generate a token that can't be guessed or mimicked. For example, with your web services, a site member (via your mobile app) logs in to your server. The server generates a session token to use for future web service requests.

We want to ensure no attacker can guess or generate that token. *Cryptographic Engineering* sets the standard as 128 bits of entropy. 128 bits is 16 bytes. Here is our solution:

```
$random = mcrypt_create_iv(16, MCRYPT_DEV_URANDOM));
$token = substr(base64_encode($random), 0, 22);
$token = str_replace(['/', '+'], ['-', '_'], $token);
```

First we generate 16 bytes (128 bits) of random data. Since we need to transmit the token via HTTP/HTTPS, we use `base64_encode()` to transform the value to printable characters. The base64-encoding produces a 24-character string, including two padding characters at the end. We don't need those padding characters, so `substr()` cuts it down to 22 usable characters.

We don't plan to ever `base64_decode()` the token. It's simply a random value to be passed back and forth, so it doesn't hurt anything to chop off the padding at the end. We don't lose entropy in doing so.

Finally, HTTP/HTTPS treat `/` and `+` as special characters. The `/` can break up a URL and `+` can be transformed to a space. To prevent any potential transmission problems, we convert to each to `-` and `_`. Again, because we will never try to decode the token, it does not matter if we swap characters. What's important is that we have a token which retains 128 bits of entropy.

Encrypting and Decrypting a String

Before you start making cryptographic decisions, find out whether you have libraries available to make the correct decisions for you. Let's hope you do! Meanwhile, though, let's look at a concrete example.

Suppose that with your web services, you have decided to use AES encryption in CBC mode with a 256-bit secret key. Your mobile app uses this method in talking to your server via your web services. This means that both the server and the app must know the same shared secret key.

Encryption is pointless unless you can guarantee the integrity of the transmission. If an attacker modifies the encrypted message, you need to detect that fact. Use HMAC (Hash-based Message Authentication Code). HMAC requires a separate 256-bit secret key.

Our mode of encryption requires a random "starting point." This is so that, if the same text is encrypted twice with the same secret key, the encrypted string will be different. This starting point is called the *Initialization Vector* or *IV*.

Key Creation

We need two secret keys (one for encryption, one for HMAC authentication), and each of the keys needs 256 bits of entropy. We pull 64 bytes (512 bits) from the random number generator and base64-encode the result for easier storage.

```
$largeKey = base64_encode(
   mcrypt_create_iv(64, MCRYPT_DEV_URANDOM)
);
```

Both sender and receiver need to securely retain copies of the above `$largeKey`.

Key Derivation

We have a large key stored as a base64-encoded entity:

1. Decode the entity into raw data.
2. Take the left half, the first 32 bytes (256 bits).
3. Take the right half, the second 32 bytes.
4. Create the encryption password as SHA-256 of the left half.
5. Create the authentication (HMAC) password as SHA-256 of the right half.

```
$raw = base64_decode($largeKey, true);
$left = mb_substr($raw, 0, 32, '8bit');
$right = mb_substr($raw, 32, 32, '8bit');
$encryptionKey = hash('sha256', $left, true);
$authenticationKey = hash('sha256', $right, true);
```

The SHA operations do not add any security to the encryption. We're taking a 256-bit value and transforming it into a different 256-bit value. One is no more or less guessable than the other.

You're doing this in the context of hardening your web services. You likely need to be storing that secret key in your app. The app can be downloaded and installed by anyone, including all potential attackers. The app could, in theory, be decompiled or reverse-engineered. The SHA operations mean that you don't directly use the secret key as compiled into the app.

This is one more level of protecting information from potential attackers. Given the low cost, I prefer to transform the secret keys with SHA before use.

Authentication

Cryptography Engineering chapter 6.7 "Using a MAC" (Message Authentication Code) begins by warning:

> *Using a MAC properly is much more complicated than it might initially seem. We'll discuss the major problems here.*

Therefore, take note that *my* example code may be correct for *my* use case, but that does not ensure that *you* have it right for *your* use case. *Cryptography Engineering* is six chapters in before it even *begins* to discuss the complications of message authentication!

For example, do you authenticate (generate the HMAC checksum)

first and then encrypt the whole result, or do you encrypt first and generate the authentication code from the encrypted string? Our example does the latter; however, *Cryptography Engineering* concludes chapter 7.2:

> *You can argue for hours which order of operations is better. All orders can result in good systems, all can result in bad systems. Each has its own advantages and disadvantages. We choose to authenticate first for the rest of this chapter. We like the simplicity of authenticate-first, and its security under our practical paranoia model.*

The encrypted string consists of three parts:

- The 16-byte Initialization Vector
- The 32-byte HMAC (authentication code)
- The cipher text

To decrypt the string, we need both the secret key *and* the Initialization Vector. Without the Initialization Vector, we can't decrypt. It's standard practice to send the Initialization Vector as plain text (not encrypted) along with the cipher text.

However, you need to ensure an attacker did *not* manipulate either the Initialization Vector or the cipher text. Therefore, the HMAC must cover *both* the Initialization Vector and the cipher text.

The Initialization Vector, the HMAC, and the cipher text are raw binary data. We therefore base64-encode each of the three items separately for transmission. Base64-encoding does not use the : character, and so we can use two : characters as separators between the three entities.

Encrypt an Array

Let's get to it. Assume we have a PHP array `$data` to encrypt and transmit. We have derived the keys as shown previously.

LISTING 1

```
<?php
$largeKey = base64_encode(
    mcrypt_create_iv(64, MCRYPT_DEV_URANDOM)
);

$raw = base64_decode($largeKey, true);
$left = mb_substr($raw, 0, 32, '8bit');
$right = mb_substr($raw, 32, 32, '8bit');
$encryptionKey = hash('sha256', $left, true);
$authenticationKey = hash('sha256', $right, true);

$now = new DateTime();
$data = ['foo' => 'bar',
         'now' => $now->format(DATE_ISO8601)];
$message = json_encode($data);
$initVector = mcrypt_create_iv(16, MCRYPT_DEV_URANDOM);
$cipherText = openssl_encrypt(
    $message, 'aes-256-cbc', $encryptionKey,
    true, $initVector
);
$toCover = $initVector . $cipherText;
$hmac = hash_hmac(
    'sha256', $toCover, $authenticationKey, true
);
$result = base64_encode($hmac) . ':'
        . base64_encode($initVector) . ':'
        . base64_encode($cipherText);

var_dump($result);
```

1. Convert our PHP array to JSON format, which is a plain text string, for encryption.
2. Generate 128 bits of random data as the Initialization Vector.
3. Encrypt the JSON string. The third parameter 1 is the constant `OPENSSL_RAW_DATA` beginning with PHP 5.4.
4. Concatenate all material which must be checksummed, that is, the Initialization Vector and the cipher text.
5. Generate the HMAC checksum.
6. Form the result into a single string for transmission.

Decrypt a String

Listing 2 is a PHP function or method that throws exceptions to return plain-English-text error messages. Assume we have derived the encryption and authentication keys as in Listing 1. Note that we use `hash_equals()`, which is not available until PHP 5.4. The function documentation http://php.net/function.hash-equals provides a plain PHP function to use for earlier PHP versions.

LISTING 2

```
<?php

function doDecryptResult($result, $authenticationKey, $encryptionKey) {
    $result = (string)$result;
    $results = explode(':', $result);
    if (3 !== count($results)) {
        return 'Invalid input string';
    }
    $hmac = base64_decode($results[0]);
    $initializationVector = base64_decode($results[1]);
    $cipherText = base64_decode($results[2]);
    $toCover = $initializationVector . $cipherText;
    $calculated = hash_hmac(
        'sha256', $toCover, $authenticationKey, true
    );
    if (!hash_equals($hmac, $calculated)) {
        throw new \Exception('Encrypted string not valid');
    }

    $message = openssl_decrypt(
        $cipherText, 'aes-256-cbc', $encryptionKey,
        true, $initializationVector
    );
    $unpacked = json_decode($message);
    if (null === $unpacked) {
        throw new \Exception('Decrypted string not JSON');
    }

    return (array)$unpacked;
}

// we can test our function with this:
require "listing1.php";
$x = doDecryptResult(
    $result, $authenticationKey, $encryptionKey
);
var_dump($x);
```

1. Ensure we are working with a string data type.
2. Split the string into its three parts.
3. If we did not get exactly three parts, return an error.
4. Base64-decode the three parts to obtain HMAC, Initialization Vector, and cipher text.
5. The HMAC calculation covers the Initialization Vector and cipher text.

6. If our calculated value does not match the transmitted value, return an error. We do not decrypt until we know the string has not been tampered with.
7. Decrypt. The third parameter 1 is the constant `OPENSSL_RAW_DATA` with PHP 5.4 onwards.
8. JSON-decode the result. If JSON-decoding failed, return an error.
9. Return the result after casting it to an array. `json_decode` has an option for this; also be aware that encode/decode behavior changes between different PHP versions.

Involving Experts

Cryptographic Engineering laments:

> *Among cryptographers, Bruce's first book, Applied Cryptography, is both famous and notorious. It is famous for bringing cryptography to the attention of tens of thousands of people. It is notorious for the systems that these people then designed and implemented on their own.*

The final chapter of that same book begins:

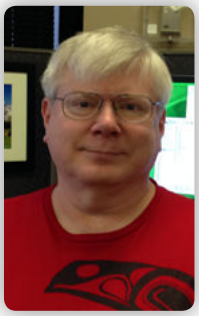
> *There is something strange about cryptography: everybody thinks they know enough about it to design and build their own system. We never ask a second-year physics student to design a nuclear power plant… Yet people who have read a book or two think they can design their own cryptographic system.*

This article treads the same shaky ground. I've given you a concrete example without the full explanation of context. I've read a book or two but that does not make me a cryptographer.

Involve experts where you can: obtain detailed advice, ask questions, and get answers. Continue your own education with the Additional Reading.

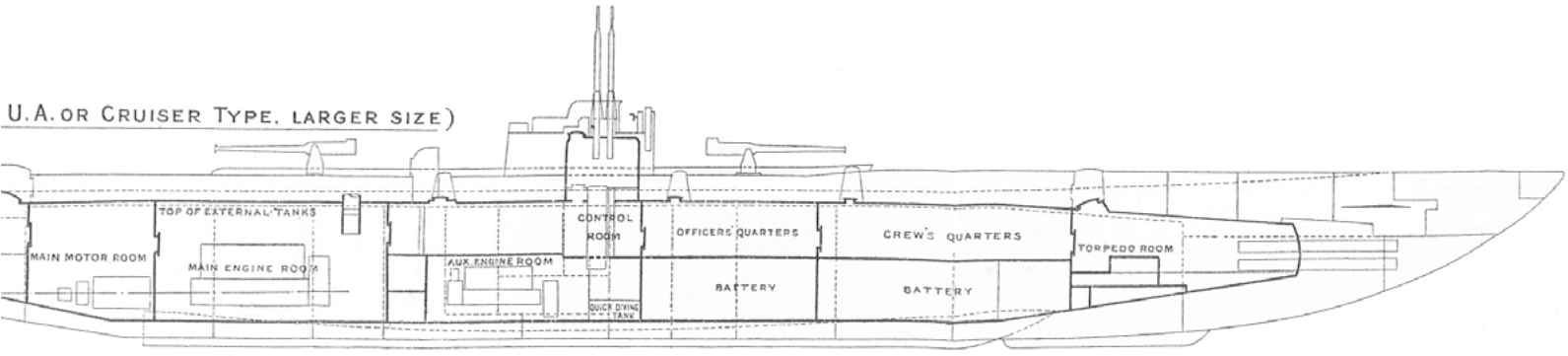
Additional Reading

1. *Cryptography Engineering: Design Principles and Practical Applications* by Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. Reading a book or two won't make you a cryptographer. But read the book or two anyway, starting with this one. http://www.amazon.com/gp/product/0470474246
2. *Information Security* at Stack Exchange, http://security.stackexchange.com. I find the *Information Security* folks to be friendly, helpful, authoritative, and thorough. Learn to ask questions correctly and you'll be delighted with the responses. Don't be shy, but show that you've thought things through before typing out the question. Related are *What to do when you can't protect mobile app secret keys?*[5] and *How to encrypt in PHP, properly?*[6].
3. *Myths about /dev/urandom* by Thomas Hühn, http://www.2uo.de/myths-about-urandom/. Excellent article about randomness and random number generators.
4. *Insufficient Entropy For Random Values* by Padraic Brady. A good, thorough, enlightening discussion. Click the top left corner of the page to continue with the entire online book, *Survive The Deep End: PHP Security*. http://phpa.me/phpsec-insufficient-entropy
5. *How To Safely Generate A Random Number*. This article explains one of the ways that OpenSSL gets it wrong, and why you want to be using /dev/urandom. http://phpa.me/safely-rng
6. *Block cipher mode of operation*. Also, *Precisely how does CBC mode use the initialization vector?*[7]. These explanations may help you understand how to use AES encryption correctly. http://phpa.me/block-cipher-op
7. *Using Encryption and Authentication Correctly (for PHP developers)* by Paragon Initiative staff. Their web site has a number of useful articles, including *The State of Cryptography in PHP*[8]. http://phpa.me/paragonie-correctly
8. *The Cryptographic Doom Principle* by Moxie Marlinspike. It's a fun read on a serious topic, and why my examples are authenticate-then-decrypt rather than the other way around. http://phpa.me/crypto-doom-principle

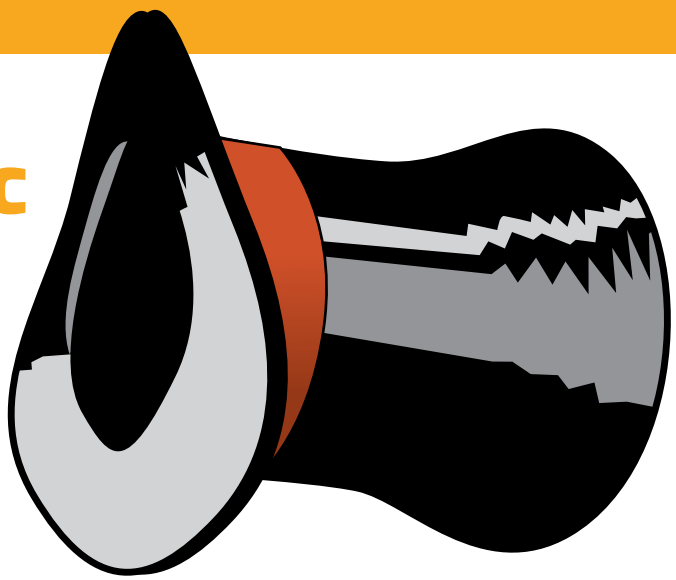


Ed Barnard has been programming computers since keypunches were in common use. He's been interested in codes and secret writing, not to mention having built a binary adder, since grade school. These days he does PHP and MySQL for InboxDollars.com. He believes software craftsmanship is as much about sharing 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

5 http://security.stackexchange.com/q/100129
6 http://security.stackexchange.com/q/80888
7 http://crypto.stackexchange.com/q/29134
8 http://phpa.me/state-php-crypto



Removing the Magic with Functional PHP



David Corona

Everyone likes magic. I like magic; you like magic. Then why would we want to get rid of it? Where's the fun in that? Magic is fun,and not when it comes to writing software. In this article, I am going to be talking about magic in programming. I'll discuss why it's bad and how we can reduce the amount we put in our code.

All Our Applications are Full of Magic

As programmers today, we have access to more tools than ever. Although that's a great thing, we need to be careful about which tools we choose to use in our applications. Magic is always there, wanting to obfuscate our code with all kinds of tricks such as reflection and magic methods. We have to constantly be on our toes. Maybe that is an obvious concept, but it is one that cannot be stressed enough.

How do we fight magic? We do so by leveling up our magic resistance! We learn how to write our software in a way that does not rely on magical tools. Luckily, we have two weapons at our disposal to help us.

Simplicity (+10 MR)

Our first weapon of choice? Simplicity. This should be our ultimate goal when writing software. Simplicity will not only help us understand our code better but also will help the next person as well. Trust me: these people will thank you. Let's start by looking at a few things that are magical and how we might simplify them.

Object-Relational Mappers (ORMs)

These are fairly common today. Nearly every large framework has some type of ORM built in to automagically read and write to a database. They sure do make your life easier… until they don't. Here is the thing about ORMs: they are made to work for all applications and use cases. They have no way to optimize for the queries your application needs to make. You may have found yourself writing some SQL queries manually because your ORM was being dumb and slow. Doing so was likely much easier and quicker, too. Here is a query you might write.

```
$results = $accounts
   ->where("first_name = 'Bob'")
   ->andWhere("last_name = 'Sacameno'")
   ->andWhere('deleted IS NULL')
   ->andWhere('ref_id = 2')
   ->not()->andWhere('other_id = 3')
   ->orderByDesc('created')
   ->limit(10)
   ->getResults();
```

"Give me all accounts in order whose name is Bob Sacameno, is not deleted, and has other matching IDs." Nothing too complicated. Who knew there were so many Bob Sacamenos. Here is the same query if you would have written it manually. You should be using something like PDO, but you get the idea.

```
$results = mysql_query("SELECT * FROM accounts
    WHERE first_name = 'bob' AND last_name = 'Sacameno'
    AND deleted IS NULL AND ref_id = 2 AND other_id <> 3
    ORDER BY created DESC LIMIT 10");
```

This one was much easier to write. I didn't need to download a large library and read through a bunch of docs. Now what if you wrote this query, but it's not working quite right. In other words, it looks pretty straightforward. *"It should just work"* right? Which one would you rather debug? Have you ever seen the queries that ORMs can spit out? Yikes!

Dependency Injection

Any application of a certain size is probably using some type of dependency injection (DI) library. You may be using it for testing to inject ermocks and stubs. It sure makes managing your dependencies easier… until it doesn't. You might be declaring all of your dependencies up front in a config file or class. Something like this:

```
$container->bind('iFooContext', '\Foo\Context');
$container->bind('iBarController',
   '\Controllers\BarController');
```

This is actually quite nice because we are explicitly defining what our dependencies are and how to resolve them. When I use a DI library, I like to stick to these explicit definitions. You may also be using constructor injection, which does some auto-injection of dependencies based on what arguments your constructor declares. This is the magical version that uses reflection and other magical incantations to determine how to resolve your dependencies.

Now, again, what happens when a dependency is not getting injected properly? How do you figure out where the magic stopped working? Maybe you push some code around and hope for the best, or you ask Google and cross your fingers. What if I told you that you could inject all your dependencies yourself and that you don't need to use a library? That would be pretty sweet.

Automapping

Here is another good one. There is a popular library in .Net that does auto mapping. I've used it several times until I realized I didn't need it. It takes a data model from your database and maps it onto your business logic model. The idea is sound. You don't want to write your business logic directly against database schema, because a database schema tends to change. Therefore, you need a way to map back and forth between the two. Let's say you have an account model.

```
class Account
{
   public $id,
   public $name,
   public $email;
}
```

Then you can map it like this.

```
$account = Mapper::map('Account', $databaseAccount);
```

Seems simple enough. I only needed to write one line of code. These libraries will use all kinds of reflection to determine how to map your classes, but they can't figure out everything. What if we need to define some custom mappings? We have to start writing code like this:

```
Mapper::createMap('Account')->for('name', function($a) {
   return $a['first_name'] . ' ' . $a['last_name'];
});
Mapper::createMap('Account')->for('email', function($a) {
   return strtolower($a['email']);
});
```

Yikes! The simplicity starts to break down pretty quickly. If we were doing it manually, we might add a function like this to our account model class:

```
public function map($account) {
   $this->id = $account->id;
   $this->name = $account->first_name . ' '
      . $account->last_name;
   $this->email = strtolower($account->email);
}
```

Then call it like this:

```
$account->map($databaseAccount);
```

Ah, yes. Much better. We wrote a function on our model that can map data, and we called it. It doesn't get much simpler than that. We didn't need to read docs to figure out how to map our data model. Now, *again*, what happens when your data model isn't getting mapped properly? Which one would you rather debug?

These tools all have something in common. They help us write less code, but they do not reduce our codebase. We are not reducing 20 lines of code down to 2 lines by using that mapping library. We are increasing that 20 lines of code to 1,000+ lines. As soon as you bring that library into your project, it becomes your code. *You* are responsible for it. Your manager or customer is not going to care that there is a bug in your ORM.

Then how can we write our programs in another way in which we won't need to rely on so many of these tools? To answer that question, we need to look at our second weapon in the fight against magic.

Functional Programming (+40 MR)

Entire books have been written on functional programming. You could say it is a fairly large topic. For this article, however, I will only covering a few basic concepts, just enough to make the last section make sense. If you're already a pro, feel free to skip this section.

To keep things simple, functional programming focuses on functions. Everything is a function that returns a value. In functional code, a function's output ideally is dependent only on its inputs, without side effects. What does that mean? Let's see.

First Class Functions

Functions as data. This is probably the single most important concept of functional programming. Functions are treated the same as any other value is. You can pass them into and return them from other functions as if they were any other piece of data. PHP has anonymous functions for us to take advantage of. Here is a simple anonymous function used as a callback for some random asynchronous operation.

```
function getSomeAsyncData('foo', function($data) {
   // do something with async data here.
});
```

In PHP, you write an anonymous function much like any other function but without a name, hence the name *anonymous* function. You can see here that we have a function that is receiving another function as one of its arguments. This is also called a *Higher order function* because it receives a function as an argument. Higher-order functions are functions that can receive or return functions. This is also a fancy name you can use later to impress your friends.

```
function getAddFunction($a) {
   return function($b) use ($a) {
      return $a + $b;
   };
};

$add5 = getAddFunction(5);
```

In this example, we have a function called `getAddFunction`, and its job is to return a new function that takes a single number. We use this function to create an `add5` function that we can call like this:

```
$sum = $add5(10); // returns 15
```

This also shows off a concept called *partial application*, in which we partially apply the number 5 to an `add` function. So we create a new `add` function that needs one argument instead of two. We *partially applied* the number 5. If this example is confusing, take a moment to think through it a few times. If you have never seen anything like this before, it can be hard to wrap your mind around it initially. Consider what happens if we pass in different values for `$a` to `getAddFunction`.

Pure Functions

Pure functions are functions without side effects. A function receives input, runs a computation on that input, and returns a result. There is no concept of global state. When we call a function with a given value, we should *always* receive the same result. Every time. Period. Nothing else should happen.

```
function add($a, $b) {
   return $a + $b;
}
```

This function does one thing. It adds two numbers together and nothing else. It is considered a pure function. We can call this function 100 times with the numbers 1 and 2, and it will always return 3. As a comparison, this would be an *impure* version of the same function.

```
$amountToAdd = 5;

function add($a) {
   global $amountToAdd;
   return $a + $amountToAdd;
}

$sum = add(5); // returns 10?
```

You can see that this function adds `$amountToAdd` to the number. It uses a global variable inside of it. Therefore, if we called it 100 times, it may not always return 10. We cannot rely on the fact that `$amountToAdd` will always be 5. It might be changed at any moment by some other piece of code.

What things are side effects? Things such as mutating global state, file IO, printing messages, or sending html to a browser are some examples. Doing these things in a function creates what we call *impure* functions. They are dirty and tainted awful things. We want to limit the number of impure functions in our code. They should be isolated to the parts of the application that deal with IO and nowhere else.

Immutable Data

This says that once a variable has been set, it cannot be changed. That's it. You can't change it. Say we have an array of numbers. How would you normally add 5 to each number in the array?

```
$numbers = [1, 2, 3, 4, 5];

for ($x = 0; $x < count($numbers); $x++) {
   $numbers[$x] = $numbers[$x] + 5;
}
```

Sweet. This works great! Nothing can go wrong here, or can it? What if this array were passed in from another function that was relying on those original values that you just changed? That code is probably going to break because you just done changed them.

Unfortunately, PHP only has immutable data in the form of `const` or with some fancy `__get` and `__set` magic. Then how do we work around this limitation? Const all the things? Not quite. We need to think of our data as immutable and treat it that way. Of course we still need to be able to change our data because data that you can't change would render our application fairly useless. We don't need to actually change it: we can return new data. Let's change this to a map and return a new array instead.

```
$biggerNumbers = array_map(function($num) {
   return $num + 5;
}, $numbers);
```

Much better and it's easier to read IMO. We know what `array_map` does. It runs a function over an array and returns a new array from the results. Now if you recall, earlier we made a nice `$add5` function using the `getAddFunction(5)` function. Let's make this a little bit simpler by using it.

```
$biggerNumbers = array_map('add5', $numbers);
```

Oh, yeah. That's what I'm talking about. We don't even need to look at the `add5` function, because it says add5 right there!

That's it for now. There are a *lot* more functional concepts that you can explore on your own, which I hope you do. It just might change how you write code. Now let's use our new found knowledge to build a simple to-do application. Although this is a simple application, it provides several opportunities for us to use our new weapons.

To-do App

> *The source code for the sample application can be found at https://github.com/davesters/functional-todo-example/*

Close your eyes and imagine this (after you read this paragraph of course). What if we could build this application in a way that every class and every dependency was bootstrapped up front in one file right at the beginning of our application? If we wanted to know how a certain piece of code was being handled, or how a certain request traveled through our application, all we needed to do was to look at this one file. Doesn't that sounds incredible? Let's make it happen.

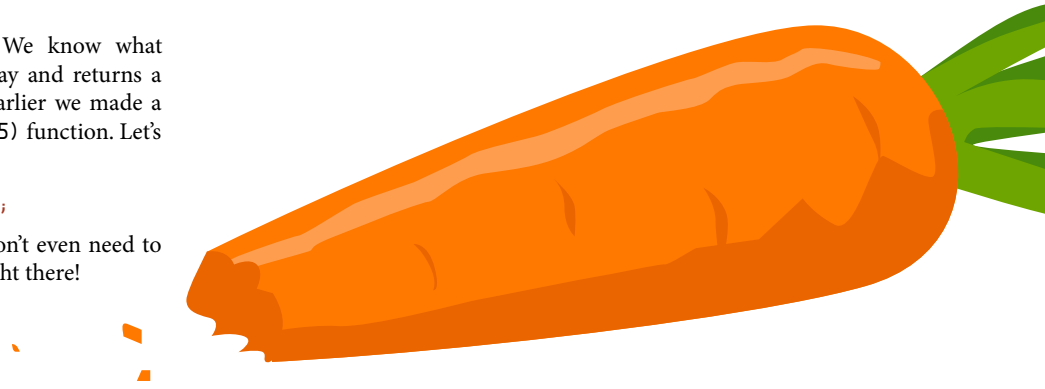
Index.php

First we will create an index file to house our bootstrap code. Take a look at Listing 1.

LISTING 1

```
01. <?php
02. // index.php
03.
04. require 'vendor/autoload.php';
05. require 'src/TodoApp.php';
06. require 'src/Mustache.php';
07.
08. $handlers = []; // [1]
09.
10. $mustache = Mustache::Render(new Mustache(), function($t) {
11.    return file_get_contents("/views/$t.mustache");
12. }); // [2]
13.
14. $klein = new \Klein\Klein(); // [3]
15.
16. $todoApp = new TodoApp($klein, $mustache, $handlers); // [4]
17. $todoApp->setRoutes();
18. $todoApp->start();
```

This is the starting point of our app. **[1]** We create an array that will house our route handler functions, which we will get to shortly. **[2]** Then we set up a Mustache template renderer. **[3]** We are using an open source micro framework called `Klein` that will handle the basic routing for our application. **[4]** Then instantiate the `TodoApp` class and pass in its dependencies.



TodoApp.php

Listing 2 shows the TodoApp class. It has a simple constructor. **[1]** The start function, which we called from `index.php`, which just calls the `dispatch()` method on the `Klien` app. **[2]** It will have a `setRoutes()` function and a **[3]** `handle()` function.

LISTING 2

```
01. <?php
02. // TodoApp.php
03.
04. class TodoApp
05. {
06.    private $app, $handlers, $mustache;
07.
08.    public function __construct($app, $mustache, $handlers) {
09.       $this->app = $app;
10.       $this->handlers = $handlers;
11.       $this->mustache = $mustache;
12.    }
13.
14.    public function start() {
15.       $this->app->dispatch(); // [1]
16.    }
17.
18.    public function setRoutes() { // [2]
19.       //...
20.    }
21.
22.    private function handle($view, $handlers) { // [3]
23.       //...
24.    }
25. }
```

Notice that we are receiving our Klien app instance from `index.php`. We instantiated it in our bootstrap code and passed it in. We will basically be doing this sort of injection everywhere. It will make things so much easier for maintenance and testing. We don't need a dependency injection framework if we can inject them ourselves!

Routes

Take a look at Listing 3. Here we have the code for the `setRoutes()` and `handle()` functions. The `setRoutes()` function is simple. **[1]** We set up our routes via `Klien`'s interface. This is where things get a little tricky if you are not used to working with anonymous functions. We call our `handle()` function the route handler that the `Klien` router will use. **[2]** The handle function takes a `$view` string and a `$handler`. We get the handler from our `$handlers` array, which we will set up in our bootstrap code shortly.

[3] Now the tricky part. Our `handle()` function actually returns a *NEW* function. The `Klien` router will call this new function when the route is hit. **[4]** The `$handlers` variable that we receive is just a simple array of functions. This array is run through an `array_reduce()`. The ultimate outcome of the reduce will be a `$model` we can use to render our view template. **[5]** We pass it to mustache and send the response to the browser.

These two functions can be a bit confusing at first, but they are extremely important, as they are the backbone of our entire application. All our code will be run through these two functions. Take a few moments to understand them if necessary. One terrific thing about how we are setting this application up is that these two files will contain the entirety of our framework and `Klien` interfacing code. The rest of our business logic code will be completely isolated and framework agnostic.

LISTING 3

```
01. // TodoApp.php
02. public function setRoutes() {
03.    $this->app->respond('/', //[1]
04.       $this->handle('index', $this->handlers['index']));
05. }
06.
07. private function handle($view, $handlers) { // [2]
08.    return function($request) use ($view, $handlers) { // [3]
09.       // get the request parameters or query string.
10.       $params = $request->params();
11.
12.       // reduce the handlers down to a single view model.
13.       $model = array_reduce($handlers,
14.          function($model, $handler) { // [4]
15.             return $handler($model, $params);
16.          }, []);
17.
18.       $this->app->response()
19.          ->body($this->mustache($view, $model)); // [5]
20.
21.       $this->app->response()->send();
22.    };
23. }
```

IndexHandler.php

Now that we have the complex part of the application out of the way, let's write the handler for our index route to get a list of to-do items. Look at Listing 4.

LISTING 4

```
01. <?php
02. // IndexHandler.php
03.
04. class IndexHandler
05. {
06.    public static function Handle($dataSource) { // [1]
07.       return function($params) use ($dataSource) { // [2]
08.          $todos = $dataSource('SELECT * FROM todos WHERE
09.          deleted IS NULL ORDER BY created'); // [3]
10.
11.          $count = array_reduce($todos,
12.             function($carry, $todo) { // [4]
13.               return $carry + (!$todo['completed'] ? 1 : 0);
14.             }, 0
15.          );
16.
17.          return [
18.             'title' => 'Todo App',
19.             'todos' => $todos,
20.             'total' => $count
21.          ]; // [5]
22.       };
23.    }
24. }
```

The `IndexHandler` class shows the common interface that all our business logic classes will follow. **[1]** It has one function called `Handle()`. This function takes a `$dataSource` function, which as you can tell by the name, is our source of data, or database access class. **[2]** We then immediately return a new function. This function will be called by the `array_reduce()` in the `TodoApp`'s `handle()` function. It receives a `$params` array, which will contain any relevant request information that our business logic would need, such as query string parameters, URL arguments, and so on.

[3] We call our `$dataSource` function with an SQL query, which returns a list of to-do items, [4] get a count of uncompleted to-dos in a `$count` variable, and [5] return an array. The next thing we need to do is set up this handler function in our bootstrap code. Remember that empty `$handlers` array we created in our `index.php` file?

```
// index.php

$handlers = [
  'index' => [Handlers\IndexHandler::Handle($dataSource)]
];
```

This adds a new `index` key to our `$handlers` array. which is just an array of functions. In this case, it is a single function. It calls the `Handle()` function we wrote above and passes it the `$dataSource` function it needs. To save space, I am not going to show the bootstrap code that creates the `$dataSource` function. Feel free to look at the source code for that part.

Update and Add Handlers

Next we need to define handlers for our add and update routes. These will let users add new to-do items and update existing ones.

```
// index.php

$handlers = [
  'index' => [Handlers\IndexHandler::Handle($dataSource)],
  'add'   => [Handlers\AddTodoHandler::Handle($dataSource)],
  'update'=> [Handlers\UpdateTodoHandler::Handle($dataSource)]
];
```

I am adding two new handlers for the add and update routes. I defined these the same way the index handler is defined but will not show them here for brevity's sake. Check the source to read them in all their glory. I hope you can start to see a pattern forming here.

There is one problem so far though. As mentioned earlier, we want to create *pure* functions as much as possible and, so far, all our functions have been impure. Our main framework class, rendering, and database functions are going to be impure because they deal with IO. There is no way around that. Our `IndexHandler`'s `Handle` function however could be turned into a *pure* function if we move the database call to somewhere else.

IndexQuery.php

What do you know? We have an `IndexQuery` class. This class will do the query for us. It will be an impure function because it needs to do database IO. Quick reminder: it is not considered *pure*, because if we called it multiple times, it may return different results as the data stored in our database changes. See Listing 5 for that good stuff.

LISTING 5

```
01. <?php
02. // IndexQuery.php
03.
04. class IndexQuery {
05.    public static function Query($dataSource) {
06.       return function($model, $params) use ($dataSource) {
07.
08.          $todos = $dataSource('SELECT * FROM todos WHERE
09.             deleted IS NULL ORDER BY created');
10.
11.          return [ 'todos' => $todos ];
12.       };
13.    }
14. }
```

This is fairly simple. It has the same basic structure as a handler class does, but all it does is query our database and return the raw results of that query. With this in place, we can now remove the database query from our `IndexHandler` as in Listing 6, and it turns into a nice pure function.

LISTING 6

```
01. <?php
02. // IndexHandler.php
03.
04. class IndexHandler
05. {
06.    public static function Handle() {
07.       return function($model, $params) {
08.
09.          $count = array_reduce($model['todos'],
10.             function($carry, $todo) { // [4]
11.                return $carry + (!$todo['completed'] ? 1 : 0);
12.             }, 0);
13.
14.          return [
15.             'title'=>'Todo App',
16.             'todos' => $model['todos'],
17.             'total' => $count
18.          ];
19.       };
20.    }
21. }
```

Yes. The pundits all agree. Now it is only using the data it receives to produce the model. We can call this 100 times with the same list of to-dos and always get back the same model. Let's wire this up.

```
// index.php

$handlers = [
   'index' => [
      Queries\IndexQuery::Query($dataSource), // [1]
      Handlers\IndexHandler::Handle()
   ],
   // ...
];
```

[1] All we needed to do was add this new query class as a new item in our handler array for the `index` key. We put it first so it will run before the `IndexHandler`, and we have a nice separation of database IO to business logic. Notice how we removed the `$dataSource` dependency from the `IndexHandler` class? It does not need this anymore, so no need to pass it in. I also changed the update and add handlers to something similar, but I won't show that here.

Summary

Our application is done! We have a single `index.php` file that wires up the entire application, and we can see exactly which classes are depending on which and what order the code is executing in.

You might be thinking that this `index.php` file will soon become a maintenance nightmare at 1,000 lines of code because of all the classes and dependency hierarchies you have. Well that's kind of the point. You have to type out all those levels of dependencies. Maybe when your hand starts cramping up, you will realize how complex your application has become. It should make you step back and think more about your application architecture. Perhaps there is a better way.

You may also argue that we are losing some of the strongly typed benefits of our object-oriented PHP code by using functions and static methods. This is just the way I chose to write this example. There are other options as well that you should feel free to explore. The ultimate goal remains the same.

One more thing: Keep in mind that I am not trying to tell you to stop using your libraries and frameworks. I don't have any problems with most of them. All I hope is that the next time you find yourself reaching for that library to add into your application, you ask yourself two questions first.

- What problem am I having that this library is going to solve?
- Can I change my problem around so I don't need the library?

I did not want to use a complex dependency injection framework. As such, I changed the way my application was architected so I could manually wire up dependencies. I did not need a large ORM, so I wrote my queries manually. I hope this will get you thinking of new ways to keep your applications simple and that your future code is cleaner because of it. Your future self will be grateful.

David has been programming since he was a wee laddie and has been building applications and web sites for almost 20 years in many different languages. He is currently working as a software developer with the great people at Buddy.com building IoT data solutions. When he is not working or hacking away on a personal project, he enjoys playing soccer, getting outdoors, or binging on snacks and Netflix. @davesters

RegEx is Your Friend

Liam Wiltshire

Regular expressions are not the scary, hard-to-use, slow things they are sometimes made out to be. In fact, in the right hands, they are a powerful tool in your toolkit. In this article, you'll see when they are the right tool for the job and when they are not.

Regular expressions are scary. At least, if you ask Google, that's what you might think (500,000 results)—and slow (1,460,000). For many developers, they are something best avoided, something unmaintainable and difficult to understand. From personal experience, I've seen some amazing attempts to avoid using them—from 10 chained `explode`s to `str_replace`ing everything except the bits you need, piece by piece. However, this doesn't have to be the case—regular expressions are neither scary or slow; indeed, when used correctly, they are a powerful tool in your utility belt.

Yes, there are alternatives, and sometimes, these will do the job better than regular expressions could (more on that below). Often, though, a regular expression *is* the right tool for the job.

So What Can We Use Regular Expressions For?

While it's easy to find plenty of examples of regular expressions in use, and it's easy to list all sorts of things they can be used for, most uses boil down to one of the following four scenarios.

Data Validation

If you are handing user data, you almost certainly want to check that the data matches what you are expecting—that if you've asked for a phone number, then the user has entered a phone number, or that you're not about to add "Little Bobby Tables"[1] to your database. Checking user-provided data (for example, from a form, or `$_SERVER` variables that the user could have spoofed) against a regular expression pattern that matches the format you are expecting can be a very effective way of validating data you receive. Equally, a regular expression can be used to validate that some data *does not* contain something—for example, checking that the bio that the user has provided doesn't contain HTML if you don't want it to.

Text Replacement

Find and replace is a tool that most of us use regularly. That could be in your OS (for example, `sed`), email client, or word processor, among many other things. Regular expressions are often used (including in `sed` mentioned above) to allow you to do text replacement when you don't know the exact text you want to replace. Examples include if you want to strip out HTML from some text (where you might not know what the tags are) or if you want to strip all URLs out of a posted comment (where you don't know what the URL might be). The key benefit of regular expressions here is the *unknown* bit—you can replace based on text that "looks like" something, rather than requiring an exact match.

Text Extraction

In our jobs, we tend to deal with a lot of data manipulation. Hopefully, all of your data is well formatted, well structured, and easy to deal with … but let's face it, everyone has had to pull data out of a webpage at some point! Regular expressions can be used to extract a block of text from within a larger block by using simple patterns. What's more, if you want to pull out multiple pieces of data (perhaps you work for "S. Pammer" and want to extract all of the email addresses, for example!), you could use regular expressions to do that, too.

Pattern Matching/Configuration

If you've ever had to set up redirects on a web server (be that Apache or NGINX), you've probably had to work with regular expressions. Many different servers use regular expressions as a test for conditional rules, such as IP address matches, partial URL matches, user agent matches, etc. Understanding how regular expressions work makes this process substantially easier!

When Shouldn't We Use Regular Expressions?

Wait, so an article about regular expressions is telling us *not* to use regular expressions? As with anything in PHP (or even PHP itself), it is only one tool in your arsenal, and as always, it's a case of using the right tool for the job—there are certainly alternatives to regular expressions, and they may do a better job in some situations.

PHP filter_* Functions

PHP has a series of functions `filter_`[2] that provide simple ways to sanitize and validate incoming data to match certain patterns. These patterns are pre-defined constants that provide some commonly used filters for validation (email, IP address, for example) and sanitization (escape special chars, remove non-numerics, etc.)

The biggest benefit of using these functions is that it's substantially easier than writing a regular expression. Looking at emails as an example, there is a regular expression for validating RFC 822 email addresses—this expression is over 6,000 characters long (seriously!). While no one would recommend using that expression in practice, any shorter ones result in tradeoffs between what is practical and the "official" rules on what an email address can be. As an alternative, `FILTER_VALIDATE_EMAIL` also validates RFC 822 email addresses (with two minor exceptions), and I can't see a single reason why you wouldn't use this — it's much simpler, more future proof, and easier to maintain. In much the same way, `FILTER_VALIDATE_IP` is probably easier to read than:

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

In many cases, if you can use `filter_var`, it may

1 *Little Bobby Tables:* https://xkcd.com/327/

2 *PHP Filter functions:* http://php.net/ref.filter

additionally be faster than a regular expression. For example, a basic test of `FILTER_VALIDATE_IP` against the above pattern shows that `filter_var` is about 70% quicker. Additionally, the `filter_*` functions give you additional options that would require more complex patterns in regular expressions—for example, using `FILTER_FLAG_NO_PRIV_RANGE` to filter out private IP address ranges would be very complicated to do. Something like the following would do the trick (all on one line)!

```
^((?!(10)|192\.168|172\.(2[0-9]|1[6-9]|3[0-2]))[0-9]{1,3}\.
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})$
```

PHP str_replace & str_ireplace functions

If you are only trying to replace an exact string, then using PHP's string replace functions is substantially quicker. Running a basic (although wholly un-scientific!) test involving performing 5,000 `str_ireplaces` against 5,000 case-insensitive `preg_replaces` shows that `str_ireplace` can be twice as fast:

```
[liam@lwlaptop ~]$ php replace.php
str_ireplace Time taken: 0.013195991516113
preg_replace Time taken: 0.025457859039307
```

If, on the other hand, the text you replace contains unknowns, then using a regular expression is your best option.

Introduction to Regular Expressions

OK, so how about some actual examples? Regular expressions are massively powerful, and it's impossible to cover everything they can do in one article, but we will work through some common uses and explain what they are doing. To start, no article on regular expressions would be complete without the "hello world" of regular expressions — validating a phone number!

For this example, we are assuming that we are trying to validate an input for a U.S. phone number using the format `(123) 456-7890`, `(123) 456 7890`, or `123-456-7890`

This is a fairly straightforward regular expression:

```
^\(?[0-9]{3}\)?( |-)[0-9]{3}( |-)[0-9]{4}$
```

Believe it or not, this isn't Q*Bert swearing (honest!), so what does it all mean?

First off, let's start with the `^...$` around the outside. These are string anchors, and they indicate that the regular expression must match the start(`^`) and end (`$`) of the string (i.e., the whole string). Without these, the regular expression would match a phone number *within* a larger block of text. That may be what you want if you are extracting phone numbers but not if you are validating user input!

Next, we have this little combination: `\(?`. As with most languages, the backslash is an escape character (in regular expressions, `[\^$.|?*+()` are all reserved characters), and the question mark makes it optional. In other words, it may optionally have a literal open parenthesis.

Now, we start matching our numbers with `[0-9]{3}`. The square brackets indicate a range (in this instance, anything in the range 0-9), and the curly braces indicate the number of characters. This can either be an exact number (exactly 3 in this case) or between two numbers: `{2,4}` would mean between 2 and 4, for example.

Next, we have an optional closing parenthesis and then this combination `( |-)`—the *unescaped* brackets act as a grouping in terms of executing the match and also in terms of the data that gets returned at the end, but more on that later. Within that, we have a pipe character—this is an "or" character. In other words, this says, match either a space *or* a hyphen.

The rest of the pattern is repeating what we've already seen, so the whole pattern could be described as the following:

At the start of the string, optionally match an open parenthesis, then match exactly 3 characters between 0 and 9. Next, optionally match a close parenthesis, then match either a space or hyphen character. Next, match exactly 3 characters between 0 and 9, then match either a space or hyphen character. Finally, at the end of the string, match exactly 4 characters between 0 and 9.

Replacing Text Using Regular Expressions

As we mentioned before, if you know the exact string you want to replace, then use `str_replace`. However, a regular expression is perfect for when you don't know exactly what is being replaced. In this example, we want to replace any `<script>` tags (and their contents) from a block of text with a message "NO JAVASCRIPT ALLOWED!". PHP has a function for stripping HTML (`strip_tags`), but it has a limitation in that while it will remove the tags, it won't remove the contents between them. Listing 1 is an example of regular expressions being more effective than the alternative.

Short but sweet. How does it work?

The forward slashes are delimiters, which show where the regular expression starts and ends (the characters after the final slash are flags, but we'll come to those later).

After the initial literals, we have `.*?`. In this sequence, the period means match any character. By itself, it just matches a single character, but the asterisk means "0 or more." Together, it means match 0 or more of any character. The asterisk can follow any character. For example, `a*` would mean "match 0 or more a characters." By default, the asterisk will keep matching characters

LISTING 1

```
01. <?php
02. $text = <<<EOM
03. This is my profile, where I am going to try and break your
04. site!!
05. <script>
06. document.getElementsByTagName('body')[0].innerHTML
07.   = 'Override page';
08. </script>
09. EOM;
10. $noscript = preg_replace(
11.     "/<script.*?>.*?<\/script.*?>/is",
12.     "NO JAVASCRIPT ALLOWED!",
13.     $text
14.  );
15. var_dump($noscript);
```

until it can't match any more. For example, if you had a string "regular expressions" and used the pattern `.*s`, the match would be the whole string up to the last `s`. It wouldn't stop at the first `s`. This is a "greedy" regular expression. By contrast, the question mark makes it "lazy." It will stop matching after the first character that satisfies the next part of the rule. In the example above, `.*?s` would match `"regular express"`. Coming back to our `<script>` example, we will match any character up to the next closing angle bracket.

HTML sanitization is a whole topic on its own. Regular expressions can help you sanitize and clean up markup coming from a trusted source but for cleaning arbitrary HTML, consider using an existing project like HTML Purifier[3] or similar.

After the angle bracket, we will again match anything until the next occurrence of `</script`, where again we will match anything until the next closing angle bracket.

We mentioned above that there are two flags, `i` and `s`, after the end of the pattern. The `i` makes the pattern case *insensitive* (so the literal `script` will also match `SCRIPT` or `ScRiPt` or any other variation), and the `s` means that the period character will match newlines (by default, the period character doesn't match a newline).

Using this pattern in the `preg_replace` function will replace any text matching the pattern with the string we provided.

Matching Content with Regular Expressions

Up to this point, we've just used regular expressions for matching patterns; however, they are also very useful for extracting data from a larger block of content. To do this, we make use of parentheses. When you make groupings of patterns in a regular expression, those specific parts will be pulled out as part of the result set, allowing us to just extract the exact data we need.

As an example, we are going to pull out today's temperature from a location page on Accuweather—specifically for my home, Nottingham.

First, if we check the source code of http://www.accuweather.com/en/gb/nottingham/ng1-7/weather-forecast/330088, we can find out how the temperature is displayed:

```
<strong class="temp">13<span>
```

Using what we know of regular expressions, we can then write a quick pattern to extract the number from within that. Notice, in this instance, how we are not using start and end anchors, as we are not matching the entire block of text. We just want to match something inside it:

```
$data = file_get_contents("http://www.accuweather.com"
      . "/en/gb/nottingham/ng1-7/weather-forecast/330088");
preg_match_all(
    "/<strong class=\"temp\">(-?[0-9]+)<span>/i",
    $data,
    $matches
);
var_dump($matches);
```

In this example, I've used `preg_match_all` just to show how it can pull out multiple matches of the same pattern. If, however, you only need the first result, you can equally use the non-global version `preg_match`, which stops after the first match. This pattern also introduced one new symbol. The plus works similarly to the asterisk we used before, but rather than looking for 0 or more, the plus looks for 1 or more.

As you can see from the `var_dump` below, within our results variable, we have the full matched pattern, as well as our bracketed group as an additional element of the array. If we had multiple bracketed groupings, we would have a separate array element for each one:

```
array(2) {
  [0]=>
  array(4) {
    [0]=>
    string(29) "<strong class="temp">13<span>"
    [1]=>
    string(29) "<strong class="temp">24<span>"
    [2]=>
    string(29) "<strong class="temp">12<span>"
    [3]=>
    string(29) "<strong class="temp">22<span>"
  }
  [1]=>
  array(4) {
    [0]=>
    string(2) "13"
    [1]=>
    string(2) "24"
    [2]=>
    string(2) "12"
    [3]=>
    string(2) "22"
  }
}
```

Regular Expressions Performance

Understanding the key concepts of regular expressions is great, but what if you need to dive deeper? Many of the applications we are running might be getting huge traffic, and these matches might be being run thousands, even hundreds of thousands, of times a day, so how can we be sure that we are getting the best performance out of it?

Regular expressions are very often more complicated to test than they seem on the surface. Commonly, this is due to the number of lookaheads or lookbehinds that the engine needs to do to evaluate your test.

Take the following pattern as an example:

```
(.*?)!
```

If you were testing that against a string `"I really like PHP!"`, then there are 39 steps that the engine needs to go through, as it has to check that each character in turn matches the first step, then check the following character to see if it satisfies the literal '!'. If that test fails, it checks the previous character (a backtrack) to see if that can satisfy the next part of the match.

That's a lot of steps for a simple pattern, so what can we do to make it more efficient? Regular expressions can be expensive, particularly when the matches are very unspecific, as in the above example, as it has to keep performing lookaheads and backtracking to determine if the next/previous character satisfies the next part of the rule. Making your expression as specific as possible will help ensure it doesn't have to perform more passes than it needs. For example, if you only want to match numbers, a `[0-9]+` pattern will quit much more quickly than `.+` if the string is `"012abc"`.

In the example above, we have to go through a number of steps, as we are using positive matches. Often, using a negative character class is faster. We can rewrite the above "match anything until an exclamation mark" to "match everything except an exclamation mark" like this:

```
([^!]+)
```

Our square brackets create a character set, and the starting carat means not (i.e., the character set is "not exclamation mark"). Because it will automatically stop at the first exclamation mark, we

3 *HTML Purifier:* http://htmlpurifier.org

also don't need a "lazy" question mark. This only requires 4 steps because the engine can scan for the first exclamation mark and then work backward. This is a massive improvement over our original pattern.

As shown in the examples above, brackets are used regularly for grouping. That's great, but it can also result in the expression returning a lot more data than you actually need. By default, groups in regular expressions are capturing; they store the result of each group for further use. Consider the following example:

```
([0-9]+[MKG])\s+([a-z0-9:]+\s+){3}([^\n]+)
```

This could be used for extracting the filesize and the filename from a *nix `ls -lah` output, but it would also extract the date information, which we don't want. If we are matching on a large list of files, that could be a fairly substantial amount of extra data we are holding.

However, regular expressions give a simple way to tell the engine to make a group non-capturing by adding `?:` to the start of the group—for example:

```
([0-9]+[MKG])\s+(?:[a-z0-9:]+\s+){3}([^\n]+)
```

This doesn't reduce the number of steps needed, but it will reduce the memory required to store the data.

As we mentioned before, regular expressions are greedy by default. However, this can result in wasted lookups. If we consider the pattern `(.*)PHP` and the string `"I really like PHP, it's much better than Perl"`, it requires 132 steps. Just adding `?` to make the match-any character lazy doesn't change the result, but it brings it down to 96 steps. Why?

With a greedy selector, even though *we* know the string `"PHP"` doesn't appear again, the engine has to check every character position to the end of the string in order to know that's the case, which wastes a number of lookups. Making the match lazy, however, means that it can stop as soon as it hits the first `"PHP"` string, saving these additional lookups.

Another potential source of slowdowns is multiple options in a match. In regular expressions, we can test that a part of the pattern matches the start or end of a word with word boundaries. These are indicated by `b`. For example, consider the pattern `b(package|packed|pack)b`. In this pattern, we are looking to match a whole word (so packaging *wouldn't* be matched). As such, if we consider the phrase "Delivery rejected - badly packaged", we know that there are no matches. However, the engine would require 27 steps to figure this out. This is because the engine will check every option within the group and then check if that represents an entire word (which it doesn't), even though we know that if `"package"` matches the group but isn't the full word, then it's impossible for `"packed"` or `"pack"` to do so.

So what can we do about it? Regular expressions have *Atomic Grouping* to help with this. In short, an atomic group tells the engine, "Once you've found a match in this group, forget any backtracking you have for the group, and continue with the next part of the pattern." This is represented by `?>` at the start of the group: `\b(?>package|packed|pack)\b`. This would bring the number of passes down to 16 because once the engine recognizes that "package" satisfies the group, even though it fails at the next step, it won't try packed or pack. We've effectively been told it that if package satisfies the group rule, then there is no value in checking the other options.

The sequence of the options is important. Remembering that the engine will stop checking the group after the first match, consider this pattern:

```
\b(?>pack|packaged|packed)\b
```

Looking at the same string above, we know that packaged *would* satisfy the entire pattern (as we have the word `"packaged"` in the string above), but as "pack" will match within the group, the atomic grouping means that the engine won't check the other options. Swapping the order of the options around to `\b(?>packaged|packed|pack)\b` fixes this. As a general rule, putting the options in length order—longest first—means that your atomic groups will work effectively.

As a side note, atomic groups are non-capturing. In other words, they don't return the match within the group in the same way as `?:` above.

Conditionals in Regular Expressions

It's not very well known that regular expressions support conditionals, but it's a very powerful part of the engine. Particularly when combined with lookaheads (which allow you to check for something later in the subject string first), it can make life much easier.

The general pattern for a conditional is as follows:

```
(?(Condition)Then|Else)
```

As an example, we are going to extract the postcode (or zip code) from an address. We will assume that we know it is either a US or a GB address, so the country in the address affects the regular expression we want to match for:

```
(?(?=.*GB)([A-PR-UWYZ](
   [0-9]{1,2}
   |([A-HK-Y][0-9]|[A-HK-Y][0-9]([0-9]|[ABEHMNPRV-Y]))
   |[0-9][A-HJKPS-UW]
   ) ?[0-9][ABD-HJLNP-UW-Z]{2})
|([0-9]{5}))
```

This is quite a lengthy one, so let's break it down:

We have the initial conditional `(?` and then our condition. This has been defined as a lookahead (identified by the `?=`), which basically says "scan the subject string for anything followed by GB." A lookahead effectively returns true or false (i.e., it's not collected as a group, and it doesn't change which character we are testing against in the subject string).

After this lookahead, we have our true condition. This is a fairly lengthy one; however, it's just matching a UK postcode followed by our "false" condition matching 5 digits for a zip code.

If we use this to test the following string:

```
Buckingham Palace
London SW1A 1AA
GB
```

First of all, it will perform a lookahead for the characters `GB` after any other characters. This will return true, so we then will use the UK postcode pattern to extract the full postcode.

If, on the other hand, we tested this string:

```
1600 Pennsylvania Ave NW, Washington, DC 20500, US
```

The lookahead will not return a match (i.e., it will be false), so we will check the "else" of the conditional for 5 digits, which will return the zip code 20500.

Another use for conditionals in regular expressions is to check if a previous collected group was matched. In regular expressions, collected groups are numbered from 1 onward. These can be used in replacements and as part of a pattern, as well as in conditionals.

Let's take this pattern as an example:

```
^(?:(START)|(BEGIN))(.*?)(?(1)STOP|END)$
```

What this says is "Match either START (group 1) or BEGIN (group 2) at the start of the string. Then match anything up to either STOP if group 1 matched (i.e., if the string began with START) or END otherwise (i.e., the string began with BEGIN)."

This could be useful if you were looking to get text out of files with different but matching delimiters.

As an aside, referencing collected groups does take on different syntaxes. Most commonly outside of conditionals, it's the escape character (a backslash) and the number of the group. A common use for this is to refer to an already matched group later on in the pattern. For example, if you wanted to match the heredoc syntax (three open angle brackets, then one or more a-z characters, followed by a newline, any characters, followed by a newline, and then the contents of the first collected group (i.e., one or more a-z characters), you could use this pattern:

```
<<<([a-z]+)\n(.*?)\n\1;
```

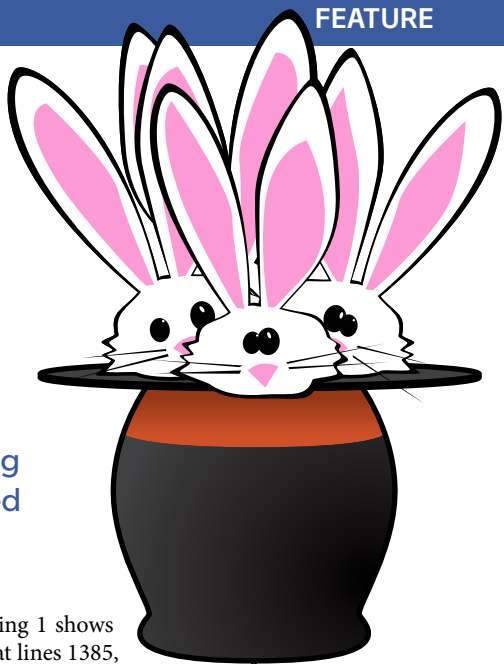
Summary

Regular expressions are a powerful tool in your tool belt. However, with great power becomes great responsibility. It's important to understand how patterns work and how to make them run efficiently. Whenever you are about to use a regular expression, do consider why. There are alternatives for certain use cases, and often, these will be easier to maintain and faster, but as with most things, benchmarking will help identify what the right option is.

One of the strengths of regular expressions is portability. Almost every operating system and programming language, as well as many servers, editors, etc., use regular expressions in one way or another. Therefore, being able to apply it to different systems will make life easier. Different implementations do exist, and each flavor may have its own quirks. However, for the most part, the patterns, anchors, and symbols are global.

Liam is a software developer and director of development at Absolute Design in Nottingham. After managing to escape the clutches of developing in Perl, he's spent the last 9 years tackling all manner of projects, from Magento sites for multi-national organizations and bespoke Laravel projects, to the plumber down the road looking for "mates" rates.

Liam is still "hands-on" in development, particularly focusing on third-party integration and eCommerce development, and in his spare time (once he's done hanging out on Stack Overflow) is trying to learn the secret to actually having some spare time.



Reference Counting : The PDO Case Study

Gabriel Zerbib

Like many dynamic languages, PHP uses a garbage collection mechanism to safeguard developers from the hassle of managing memory. Reference counting is at its core, and it is recommended to understand how to work with it so as to avoid certain design problems in your applications.

In this article, we will study the behavior of the reference counting mechanism in PHP and the impact on the life cycle of the well known `PDO` object.

The PDO Class

There is no longer any need to introduce PDO (PHP Data Objects), the abstraction layer for manipulating databases. It has come to replace the family of `mysql_*` or `pgsql_*` functions and so on.

Unlike the old extensions which all exposed a `*sql_connect` function to initiate the link to the database server, the `PDO` approach is to create the connection when constructing an object instance:

```
$pdo = new PDO($datasource, $username, $password);
```

However, the PDO class does not propose a counterpart to the procedural `*sql_close` function in order to terminate a session and free the associated resources. The manual states that this only occurs when the PDO object is automatically destroyed (in fact, in PHP, the freeing and destruction of objects are two distinct phases internally, which we can mentally unify in this article for the sake of simplicity). Alas—some may say—there is no *delete* keyword in PHP. An object is automatically freed by the engine when it considers that the instance is not used anymore, which is to say when no reference to it is left accessible.

Destruction of a PDO Instance

A quick glimpse at the source code of the PDO class[1] confirms the above statement. Listing 1 shows the relevant sections at lines 1385, 1502, and 1547.

LISTING 1

```
01. static zend_object_handlers pdo_dbh_object_handlers;
02. // ...
03. pdo_dbh_object_handlers.free_obj = pdo_dbh_free_storage;
04.
05. static void dbh_free(...)
06. {
07. /// ...
08. }
09.
10. static void pdo_dbh_free_storage(...)
11. {
12. /// ...
13.       dbh_free(dbh, 0);
14. }
```

The call to `dbh_free` occurs only when the object is freed via the automatic invocation of the `zend_object_handlers`'s `free_obj` function). We can check in the source code that there is no exposed way of reaching the `dbh_free` internal function.

Releasing a PDO instance happens, for example, if we explicitly tell the program that it can reclaim the last variable which holds the object.

```
$pdo = null;
```

This is also the case when the variable was local inside a function:

```
function doSomehtingWithDb() {
  $pdo = new PDO(...);
  $pdo->query(...);
}
```

In this listing, the `$pdo` variable is created in the function's scope and is automatically released when the function returns, leading to the disconnect from the database. Additionally, as usual, all variables and resources are freed at the end of the script's execution; this applies to PDO connections too (except for persistent connections, which we will not cover in these lines).

The Program's Life Cycle

The PHP language has left most developers under the impression that a script's life cycle is necessarily short because it is tied to the HTTP request-response cycle. And, because significant efforts are generally invested in keeping the response time as short as possible, it often means that the script's life itself is normally expected to be short, along with the resources that it allocates.

Yet, a number of new paradigms have emerged over the past few years in the PHP ecosystem, such as long-lasting command-line tasks, background *daemon*-like services[2], or even full-fledged application servers in pure PHP[3]. The disruptive difference of these usages is that the script's termination in normal conditions could well occur only after a very long time, if at all.

But in simpler programs too, we may need to explicitly control when our code disconnects from a database, as we will see below.

1 *pdo_dbh.c:* http://phpa.me/pdo_dbhc

2 *mac-cain13/daemonizable-command:* https://github.com/mac-cain13/daemonizable-command

3 *appserver.io:* http://appserver.io

Concurrent Connections

Let's consider a program whose job is to read a few fields in a table (such as a Twitter username or a YouTube address) and then to invoke a number of remote APIs (maybe to extract the last 1000 tweets of every follower of a particular user, or to slice still images out of a YouTube video, etc.). Suppose that the initial query runs fast (sub-second), but that the entire execution of the script is lengthy (several seconds) and is mostly spent in network latency and external API processing.

> **Note on architecture**
> *A good solution to this type of requirement would probably involve several components (workers) decoupled from one another, with a message queue in between, rather than a monolithic script which processes the whole chain linearly. However, let us focus on the latter, single script approach in this article, in order to illustrate the problem raised here.*

The script, which begins with connecting to the database and continues with the long-running task, can be represented in this simple form:

```
$pdo = new PDO(...);

// query the details about the job (fast)
$data = $pdo->query('select ...')->fetch();

// invoke external APIs (long)
execute_long_lasting_api_calls($data);

// end of script..
```

Theoretically, the application could deal with dozens of concurrent accesses because most of the work is done outside. Yet, if we are not careful about closing the `$pdo` connection, the number of simultaneous jobs is only limited to the maximum number of concurrent connections accepted by the database server for the applicable account. This is because, even after the query has returned results, the connection remains open, although sleeping, until the entire program terminates (see Output 1).

It is generally advised to keep this max connections number low. Although your systems will easily know how to cope with several hundred simultaneous connections, each one still uses up socket and memory resources. Moreover, the apparent need to increase this number indicates in general a misdesign of the application, rather than a real ceiling in scalability.

```
CREATE USER 'app-user'@'localhost'
   IDENTIFIED BY 'app-password'
   WITH MAX_USER_CONNECTIONS 10;
```

So we need to modify the script to release the PDO instance as soon as possible.

```
$data = $pdo->query('select ...')->fetch();
$pdo = null; // <-- Explicitly terminate the connection
// call the APIs, which take a certain time to respond.
```

Once the data are read from the table, the script does not need to keep the link active any longer, since the rest of the execution time is spent connecting to the external APIs. Nullifying the `$pdo` instance instructs the program that it can disconnect from the database, thus freeing up the slot for more simultaneous executions as in Output 2.

The access time of the initial query becomes so short that the connections don't show up in the process list anymore.

References in PHP

The PHP language operates a reference counting mechanism to track the number of "copies" of a variable in memory. Consider the following assignments:

```
$pdo1 = new PDO($dsn);
$pdo2 = $pdo1;
```

The object of class `PDO` is referenced twice. Subsequently, assigning null to one of the "pointers" has no impact on the actual instance. This amounts to removing a handle from a suitcase that had two: afterwards, you can still carry the baggage.

```
$pdo1 = null;
do_something_with_db($pdo2); // still ok
```

It is only when the last reference to an object is lost that the PHP engine destroys the underlying object.

```
$pdo2 = null; // leads to the destruction of the PDO instance
```

Therefore, one should be careful regarding the use of the PDO class in long-running scripts. The inadvertent increase of the PDO instance's reference counter could cause the connection to be held until the end of the program. But, it can be tricky to understand exactly where the references hide themselves in a dynamic language script's source code.

OUTPUT 1

```
mysql> show full processlist;
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
| Id  | User     | Host      | db    | Command | Time | State    | Info                  |
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
| 112 | app-user | localhost | appdb | Sleep   |   31 |          | NULL                  |
| 113 | app-user | localhost | appdb | Sleep   |   28 |          | NULL                  |
| 145 | app-user | localhost | appdb | Sleep   |   24 |          | NULL                  |
| 159 | app-user | localhost | appdb | Sleep   |   12 |          | NULL                  |
| 165 | app-user | localhost | appdb | Sleep   |    2 |          | NULL                  |
| 170 | app-user | localhost | appdb | Sleep   |    1 |          | NULL                  |
| 174 | root     | localhost | NULL  | Query   |    0 | starting | show full processlist |
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
```

OUTPUT 2

```
mysql> show full processlist;
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
| Id  | User     | Host      | db    | Command | Time | State    | Info                  |
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
| 244 | root     | localhost | NULL  | Query   |    0 | starting | show full processlist |
+-----+----------+-----------+-------+---------+------+----------+-----------------------+
```

Closures and Disconnections

Let's now consider the following situation, shown in Listing 2, in which we use a routing framework based on anonymous functions, such as Slim[4] or Lumen[5].

LISTING 2

```
01. // Prepare dependencies
02. $pdo = new PDO(...);
03.
04. // Declare routes
05. $app->get('/previous-example', function () use ($pdo) {
06.     $data = $pdo->query(...)->fetch();
07.     $pdo = null; // <-- tentative disconnection
08.     // ... long-running task here
09. });
10.
11. // (More routes here...)
12.
13. // Start URI parsing and invoke the matching route handler
14. $app->run();
```

The clause `use ($pdo)` tells the anonymous function that it should capture the said variable, so as to make it available in the function's inner scope when it is invoked. It leads to incrementing the reference counter. Thus, in the handler's body, the attempt to disconnect on line 16 `$pdo = null;` has no effect. The local copy of the `$pdo` variable is correctly nullified, but the reference captured by the closure remains active. The anonymous function, as an argument to the handler registration method `get()`, continues to exist until the end of the `$app->run()` function. Therefore, the PDO instance is not released before the end of the program.

The Destructor

Although it is not possible to force the destruction of an object as long as its reference counter is greater than zero, PHP nonetheless offers a way to write a destruction handler: a piece of code that the engine executes automatically when the instance is released. This is achieved by implementing the special method `__destruct()` in a class. It is recommended to carefully read the manual[6] to understand the specifics of this feature.

We are going to use it now to illustrate the observations studied above (see Listing 2). This class reports its construction and its destruction phases by indicating the elapsed time since the program was launched (using the superglobal `$_SERVER['REQUEST_TIME']`).

Let's use the class in three different situations: simple explicit release, instance capture in a closure, and automatic destruction upon program termination. See Listing 4.

The execution displays:

```
0 : __construct q
0 : __construct k
0 : __construct x
0 : __destruct q
6 : __destruct k
6 : END
6 : __destruct x
```

4 *Slim Framework:* http://slimframework.com

5 *Lumen:* https://lumen.laravel.com

6 *Constructors and Destructors:* http://php.net/language.oop5.decon

LISTING 3

```
01. <?php
02. class K
03. {
04.     private $name;
05.
06.     public static function timer() {
07.         return time() - $_SERVER['REQUEST_TIME'];
08.     }
09.
10.     public function __construct($name) {
11.         echo self::timer() . ' : ' . __FUNCTION__
12.             . ' ' . ($this->name = $name) . PHP_EOL;
13.     }
14.
15.     public function __destruct() {
16.         echo self::timer() . ' : ' . __FUNCTION__
17.             . ' ' . $this->name.PHP_EOL;
18.     }
19. }
```

LISTING 4

```
01. <?php
02. include ("K.php");
03.
04. $q = new K('q');
05. $k = new K('k');
06. $x = new K('x');
07.
08. $c = function () use ($k) {
09.   $k = null;
10. };
11.
12. $c(); // no effect
13. $k = null; // no effect
14. $q = null; // q is destroyed because not captured
15.
16. sleep(6);
17. $c = null; // The closure is freed, hence k is destroyed
18.
19. echo K::timer() . ' : END.' . PHP_EOL;
```

Let's analyze what happened. Three instances are constructed at the program's start (lines 4 to 6). The closure `$c` on line 8 (in fact, the `use` clause there) captures a reference to `k`, which is automatically released when we free the reference to the anonymous function on line 17 after the `sleep`. The `q` instance is not involved in any `use` clause, therefore it is intuitively released as soon as `$q` is nullified in line 14. Finally, the `$x` instance is never explicitly released: it is then destroyed automatically when the script shuts down (after the end of the last instruction, line 19).

Encapsulation

The solution (shown in Listing 5) is to encapsulate the PDO object using composition.

Then the previous program becomes:

```
// Prepare dependencies
$pdoWrapper = new PDPWrapper($dsn);

// Routes
$app->get(..., function (...) use ($pdoWrapper) {
   $pdoWrapper->getPDO()->query(...); // some SQL here
   $pdoWrapper->terminate();
});
```

LISTING 5

```
01. <?php
02. class PDOWrapper
03. {
04.     /** @var PDO */
05.     private $pdo;
06.
07.     public function __construct( ... ) { $this->pdo = new PDO( ... ); }
08.
09.     /** @return PDO */ public function getPDO() { return $this->pdo; }
10.
11.     public function terminate() { $this->pdo = null; }
12. }
```

This time, the PDO instance is correctly destroyed when calling `$pdoWrapper->terminate()`, because the closure has jailed a reference to `$pdoWrapper` without incrementing the counter for the PDO object which is inside.

But, caution: for this solution to work, one has to avoid capturing an explicit handle to the encapsulated PDO object. In other words, never store it in another variable:

```
$pdo = $pdoWrapper->getPDO();
```

Doing so, you lose the whole benefit of the encapsulation. For example, when using the Doctrine ORM, the following assignment would break the logic:

```
$pdo = $entityManager->getConnection()->getWrappedConnection();
```

(See Doctrine's documentation[7]). To prevent breaking the logic, we would need to take the design effort one step further by implementing the Proxy pattern, thus completely removing the `getPDO()` method, and replicating every method in the PDO class as in Listing 6.

By doing this, we make it impossible to capture an explicit reference to the `$pdo` instance. Notice the `extends PDO` declaration: although not required, it can help reusing the class in methods that accept only PDO objects. But, of course, all the public methods are overridden so the native ones will not run on the wrapper object but only on the proxied PDO instance.

Unfortunately, implementing this kind of wrapper is not always possible, especially in an application that uses a standard ORM such as Doctrine. See, for example, the Driver documentation[8], where we read that the Database Abstraction Layer in Doctrine will use an internal `PDOConnection` class to wrap the connection:

```
public function connect(array $params, $username = null,
    $password = null, array $driverOptions = array())
{
// ...
  $conn = new PDOConnection( ... );
// ...
   return $conn
}
```

LISTING 6

```
01. <?php
02. class PDOProxy extends PDO
03. {
04.     /** @var PDO */
05.     private $pdo;
06.
07.     public function __construct($dsn, $username = null,
08.                                 $password = null,
09.                                 $options = array()) {
10.        $this->pdo = new PDO($dsn, $username,
11.                             $password, $options);
12.     }
13.
14.     public function exec($statement) {
15.        return $this->pdo->exec($statement);
16.     }
17.
18.     public function lastInsertId($name = NULL) {
19.        return $this->pdo->lastInsertId($name);
20.     }
21.
22.     // ... (all the other methods in the PDO class)
23.
24.     // and of course our terminate function
25.     public function terminate() {
26.        $this->pdo = null;
27.     }
28. }
```

But if we look at Doctrine's own `PDOConnection` object[9], we see that the `PDOConnection` class itself wraps the native `PDO` class, and there is nothing we can do about it.

```
class PDOConnection extends PDO //... {
   public function __construct($dsn, $user = null,
                               $password = null,
                               array $options = null) {
     parent::__construct($dsn, $user, $password, $options);
// ...
```

Conclusion

Personally, I think that the PDO class is really missing a `close` method. Most of the other standard extensions do expose a way to explicitly release the resource, such as `Memcache` or `ZipArchive`, for example. To leave the lifecycle of resources up to the sole decision of the garbage collector is only one choice among several possible approaches, and it would not be my favorite one. I have discussed this with the maintainers[10], who explained that, since the problem can be solved in "userland," there is no need to modify the core extension. What is important, anyway, is to be aware of the potential problems that it can cause.

Gabriel Zerbib is a full-stack engineer and cloud architect, who enjoys programming in various languages since the 80s. Currently based in Tel Aviv, he specializes in high frequency applications and large scale data volumes.

7 *dbal/Connection.php at v2.5.4 :* http://phpa.me/doctrine-conn-254

8 *dbal/Driver.php at v2.5.4:* http://phpa.me/doctrine-driver-254

9 *Doctrine's PDOConnection object:* http://phpa.me/doctrine-pdo-254

10 *Re: Re: PDO Close Connection:* http://news.php.net/php.internals/90840

Nacho Cheese

Cal Evans

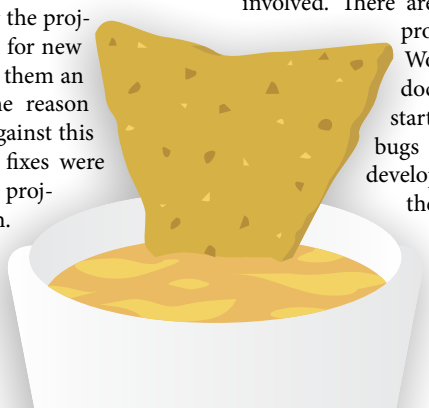
Q: What type of cheese can never be yours?
A: Nacho Cheese
It's Father's Day as I write this, so yes, I'm telling Dad Jokes. :)

A couple of years ago, I decided to get involved in developing WordPress. I followed the instructions—yes, they have very good instructions on how to get started. I went to the bugs list and found a bug I thought I could understand. I began coding.

The first thing any developer does when working on a new project is to read the code. Make sure you understand it before you try and modify it. So I began reading the WordPress codebase. In the section I was in, I noticed a few things that I knew weren't right. "Yoda conditions," nesting too deep, sometimes it was as simple as a piece of code needing a comment to make it clearer.

To help me learn the codebase, I corrected these "problems" and submitted PRs for them. After all, who doesn't want cleaner code, right? Turns out, the WordPress project doesn't. OK, that's rude. What I mean was that all of the PRs were summarily rejected and I was asked to reread the "getting started" guide because it clearly explained that PRs of a trivial nature were not wanted and not accepted.

I was, of course, incensed. I began writing a scathing comment reply explaining how letting people help clean up the code was good for new committers, and it was good for the project. I stopped short of "Do you know who I am?" but seriously, I unloaded in about four paragraphs. As my self-righteous high began to subside, I paused. I thought about what was actually said to me. (Not the tone it was said in, because, honestly, they don't owe it to me to be nice.) I began to realize that maybe they had had people come before me and done this same thing and discovered that, no, it was not good for the project. Honestly, all it did for new developers was to give them an ego stroke. Maybe the reason the project has rules against this is because these little fixes were more problem for the project than they are worth. Maybe—just maybe—they had rules in place for a reason. So I deleted my comment.



Founders of open-source projects don't owe it to anyone to accept a pull request. It doesn't matter who you are or how much you think it would improve the project. You didn't found the project, you didn't set the rules, it is NACHO PROJECT. (See what I did there?)

The great thing about open source is that if you feel strongly enough about your change, you can fork the project, make the change, and use the new version yourself. NOW IT IS YOUR PROJECT. (There's no cute Dad Joke for that, sorry.)

Yes, if you feel strongly about your PR, you can argue your point, you can reply to every comment, and show just how important your PR is. You can publicly shame the project on social media for not accepting your PR. In the process, you may even convince—or simply bludgeon—those in control of the project into accepting your PR. You have won the battle, but in the end, we will all lose the war. This isn't about just you or your PR; if everybody does this, then eventually the project founder will get tired of it and simply walk away. I've been doing this long enough, and I've seen it happen many times now.

Most developers who release a project as open source aren't looking for world domination. They are looking to solve a problem and are willing to share. If we punish them by insisting that they do things this way or that way, they will eventually just walk away.

Projects, it is in your best interest to be new-developer-friendly. It is good for you to encourage new developers—and new-to-your-project developers—to get involved. There are multiple ways to do promote getting involved. WordPress has good documentation on getting started, Symfony marks bugs as "easy" so that new developers can quickly locate them and use them as the first steps in beginning to contribute. There are other great projects out there that are friendly to new developers and we should all applaud them. We should tell others about them. We should lift them up on social media because these are how other new projects are born. When developers get involved and get in the habit of sharing, when they start their own project, open source is the natural way to go. We all win.

If a project, however, is not friendly toward new developers, that is OK too. It is a choice that the project has made. If you don't like it, you have two choices: fork it (check the license!) and maintain a new developer-friendly version of the project, or use and contribute to another project. You do not have an inalienable human right to have your PR accepted by a project. You do not have the right to belittle those in control of a project because you don't particularly like the governance or lack thereof. And you do not have any right to publicly shame the project because it doesn't measure up to your standards.

Open source is about sharing. If a project wants your talents, skills, and abilities, that is awesome; you get to share. If they don't, remember, it's Nacho Project. Let it go, move on, and find one that does want you, and that you can make your project.

You may think the problem is that it's hard to cut nacho cheese, but really, you have a grater problem.

(I'll show myself out.) :)

Past Events

July

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

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

WordCamp Europe 2016
July 24–26, Vienna, Austria
https://2016.europe.wordcamp.org

Upcoming Events

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

SymfonyLive London 2016
September 15–16, London, U.K.
http://london2016.live.symfony.com

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, Rawa Mazowiecka, Poland
http://www.phpcon.pl

PHP North West 2016
September 30–October 2, Manchester, U.K.
http://conference.phpnw.org.uk/phpnw16

Madison PHP Conference 2016
September 30–October 2, Madison, WI
http://2016.madisonphpconference.com

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

International PHP Conference 2016
October 23–27, Munich, Germany
https://phpconference.com/en/

ScotlandPHP 2016
October 29, Edinburgh, Scotland
http://conference.scotlandphp.co.uk

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

Simple is Better

David Stockton

A coauthor of the C language, Brian Kernighan, wrote, "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" With that in mind, it makes sense to try and make our programs as simple as possible. Simple programs are easier to understand, easier to debug, and easier to maintain. It does take effort to write simple code, though, so let's talk about some of the ways to do that.

Problems That Lead to Complexity

One of the most common causes of complex code is not fully understanding the problem. Unfortunately, unless the problem is trivial, this is going to happen to everyone on almost every problem they work to solve. I don't mean to sound insulting or condescending—it's just how it is. Bear with me, though; I'll explain. If you've ever written a piece of code that is still running and has not needed any maintenance, added features, or fixed bugs, the problem you solved was trivial, or it's not being used. Sometimes the understanding of the problem may have been correct, but as time goes on, the problem changed and the understanding of the problem does not.

Much of the code we write, especially for our jobs, needs to be updated. We have bugs to fix, features to add, font sizes to change, and colors to adjust. This can be due to customer requests, business requests, as well as a slew of other reasons. Even long-running systems that have not needed changes can require updating all of a sudden. Software doesn't mature as it gets older. It rots. If everyone stopped writing code, and attackers stopped trying to exploit it, then I suppose I could accept an argument that code rot is not happening anymore. But that's not realistic. Even if the code has run for years, there may be changes in the underlying operating system, or in the system that interacts with the code, or that the code interacts with. Security updates may force changes that adversely affect how the code runs. Perhaps a service that the code calls out to has changed and no longer accepts the payload you're sending, or the services that send in data may have changes in requirements which, in turn, lead to further downstream changes.

In short, any nontrivial software will require changes at some point. The choice, then, is to either maintain the software and make the changes, decommission the software so it can no longer be used, or leave it alone and deal with the consequences and fallout. In software, change is inevitable. It is a certainty. Since we cannot know what all those changes are ahead of time, it is not possible to fully understand the problem. The problem changes as time goes on.

Poor Requirements

Unless you've just started writing code, you've certainly been presented with poor requirements. Building software is hard. Clients don't know exactly what they want and assumptions are made on both sides. Clients make assumptions. When I say "clients," I mean whoever it is that is creating the requirements for software that you build. If you're building a product and you're the one coming up with the requirements, be aware that you are making assumptions—you are playing the role of the client. You may make fewer than someone who hasn't written code, but you *are* making assumptions. You may be assuming the users of your software will use the feature in a particular way (or that they want the feature at all). In some cases, you may be playing three roles—the role of the client deciding what the software needs to do, the role of the developer creating the software, and the role of the user, actually utilizing the software to solve a problem.

There are ways to help make the requirements better—you can point out any inconsistencies, remove as many assumptions as you can identify, wireframe, whiteboard, build prototypes, etc. But even with these tools, assumptions will still sneak in, requirements will change. New features will be requested, bugs will be found, and edge cases identified. The combination of an incomplete understanding the problem and changing requirements leads to more complexity in the software than there should be. It takes a very concerted effort of constant refactoring and redefining the understanding of the problem in order to keep complexity out and keep the software simple.

More Code, More Problems

There's an old joke that there are two hard problems in computer science: naming things, cache invalidation, and off-by-one errors. Coming up with good names for things in code is absolutely one of the tougher problems. Beginner programmers, learning about the differences between `int`s and `float`s, `char`s and `string`s, often choose short names that may indicate what type of data a variable holds. They want to avoid typing, and keeping track of the differences in data types may be difficult. More senior programmers figure that knowing what a variable represents and why it exists is more important. They tend to use longer, more descriptive variable names. Nevertheless, creating descriptive and useful names is difficult.

I often use the nondescript `$i` variable as a counter in a `for` loop because it's easy and doesn't require any thinking about what `$i` actually means within the loop. Using short variables like `$i` and `$j` with loops is a common practice in other languages too. In other naming exercises, I'm better—for function and method names, class names, and namespaces, I do my best to create good names. Sometimes, I'm more successful than others. When naming things, it may help to think of variables and class properties as nouns while methods and functions are verbs. If a function or method returns a boolean value, I usually name it `isAdjective` or `hasProperty`.

Doing Too Much

In almost every OOP talk I've given, I include the quote "If you need to use 'and' to describe what your class or method does, it's probably doing too much." The code can do too much for a number of reasons. Junior developers often don't understand the need to separate concerns and split chunks of code into different methods and classes. Later on, it's often easier to just add another conditional to meet the changing requirements of the software than it is to look at the software as a whole, understand how the new requirements fit in the whole scheme of things, and refactor to keep complexity low and keep the software simple. Over time, if the software is maintained in this way, complexity grows slowly until the system is essentially a big ball of mud with little bits of functionality jammed in wherever it can fit.

Keeping the code simple is not simple work. It means that adding a new feature may require more code and classes than just adding a conditional or a loop. It requires the discipline to understand when to add a conditional versus when to refactor conditionals into classes, and when copy/paste might be the right way to solve a problem. It's not easy, but as I've been saying, building nontrivial systems and keeping them simple is quite hard.

> *Over time, if the software is maintained in this way, complexity grows slowly until the system is essentially a big ball of mud with little bits of functionality jammed in wherever it can fit.*

Knowing Too Much

One common pattern I see that quickly leads to complex software is when the classes and methods know too much, or when they expose too much. Encapsulation is the concept in building OOP that allows objects to hide data and functionality in order to protect them from outside interference and misuse. The *Law of Demeter*[1], also known as the *Principle of Least Knowledge* provides guidelines to help keep software simple. It says that if we have an object with a method, that method can invoke methods on the object itself, the parameters passed to the method, any objects created or instantiated in the object, or objects that were passed in, as well as global variables that the object can reach. This means if an object has a dependency, our code should not reach into that dependent object and do things with it. In PHP, we can often see violations of the Law of Demeter when a line of code contains more than two `->` operators. For example:

```
$query = $this->mapper->getSql()->buildQuery();
```

In this case, whatever code includes the above will break if the mapper changes, as well as if the SQL object in the mapper changes. It means the code knows too much about the internal workings of the objects passed in. Code that doesn't concern itself with how other objects work, or only does that minimally, is simpler than code that has tendrils reaching into objects and going into places it doesn't belong. This kind of code is hard to test, more prone to breaking, and is difficult to debug and understand. One real-world example I see regularly is with ZF2's SQL object when you want to determine what the latest ID generated by the database happened to be. This code would be found within a method that runs an insert query. Let's take a look:

```
$id = $this->sql
          ->getAdapter()
          ->getDriver()
          ->getLastGeneratedValue($nameOfSequence);
```

The issue is that while the SQL object was injected, and it's technically OK to retrieve the `Adapter` from it, the Law of Demeter says we should not be calling anything on the Adapter. But this code requires we retrieve the `Driver` from the Adapter and then ask the Driver to tell us the value it received on the last insert for the generated ID. In the case of MySQL, the sequence name is not needed, but in PostgreSQL, it is.

In order to unit test this code, the test double for the SQL object would require the `getAdapter` method to return a test double for the Adapter. That test double would need to return a test double for a driver object when `getDriver` was called. And the `getLastGeneratedValue` method on the double for the driver would need to return a value or emulate the behavior you're trying to test. Rather than just injecting a double for the SQL object, we'd also need test doubles for an `AdapterInterface`, and a `Driver`, all of which must have behaviors and return values defined, even if you really don't care about the returned ID value. And if any of those objects changed behavior, every bit of code that retrieves the generated identifier would need to change as well.

It would have been better for the SQL object itself to have a `getLastGeneratedValue` method, which could in turn delegate to an Adapter, which would delegate to a Driver. In that case, the code could have been:

```
$id = $this->sql->getLastGeneratedValue($nameOfSequence);
```

With this design, the behavior of the adapter and driver is unknown and of no concern to our class. Unit testing would only require the injection of an SQL test double.

// Conclusion

While the naming of things is hard, and sometimes some level of complexity is needed, it's best to write code that is self-explanatory. This is a lofty goal, and it's not possible to do all the time. So we have comments in the code that can explain the how, or the what, or the why. The best comments provide insight into "why" a piece of code does what it does.

On the other side, it's not uncommon to see a comment written when the code is initially built. However, as the code changes and the method is updated and requirements are changed, oftentimes, comments are left behind, unmaintained. An incorrect, invalid, or misleading comment is worse than no comment at all. Comments should be avoided where you can. To be clear, I don't mean don't comment your code. I'm all for commenting your code. The code will require comments, but if you can make your code simple and clear enough that it is obvious what's happening, that is much better than a comment explaining what is otherwise an opaque bit of logic. Tracking through complex code is difficult enough without having to do so when the comments are lying or misleading. So if you change some code that has comments, be sure to keep them updated when updating code. Just as the comment beginning this section was misleading, bad comments lead to incorrect assumptions.

1 *Law of Demeter:* http://phpa.me/wiki-law-of-demeter

Too Many Loops and Conditionals

If you run PHPUnit against your code (and you should), one of the metrics you can generate is called the *CRAP* index. The conveniently named acronym for "Change Risk Anti-Patterns" index is a combination of cyclomatic complexity and the amount of code coverage a method has. If your code coverage increases, the CRAP score decreases. If your code complexity increases, the CRAP score goes up. Cyclomatic complexity[2] is a measurement of how many linearly independent paths through a piece of code there are. Each additional `if` statement and every added loop construct means there's a new, different path through the code. For `switch .. case` statements, each case, including `default`, would represent another unique linear path through the code.

To properly unit test code fully, there should be tests that exercise each of the paths. Cognitively, it becomes more difficult for a developer to reason through and keep track of all the ways through a bit of code as well as any changes to state and variables as the code progresses. Complexity leads to bugs. Fortunately, there's a great way to reduce complexity by removing loops and conditionals.

Removing Loops and Conditionals

If we can treat everything the same, conditionals become unnecessary. You may have seen members of the community pushing for functions and methods that only return one type of thing. PHP's own return type hints help with this. But PHP allows us to return anything we want from the same function. It's not uncommon to see a bit of code that tries to return a record from the database. If the record isn't found, it returns `false`, or `null`. This means that anything which calls this method must have a conditional to deal with either case.

```
$record = $this->getRecordById($id);
if ($record === false) {
   // Deal with no record found
} else {
   // Deal with the record
}
```

In the code above, we have two distinct paths, one if the record is found, one if it is not. To test properly we need at least two tests. You might wonder how to avoid returning something different if no record is found, and indeed, this does seem to be different. This can be accomplished by indicating the method will return a particular interface. You then have two (or more) objects that implement that interface—one object that acts normally and contains the data, and another, commonly referred to as a "Null Object," which has all the methods specified by the interface, but doesn't need to actually contain the data (since there isn't any). Typically, every method in the null object doesn't do anything.

Let's take a look at another strategy. Again, with the database, rather than a method that returns a single record (or nothing), imagine a method that gives back multiple records (or nothing), such as a filtering search over a table. A common return value would be an array of records. In this case, we get back an array regardless of whether the filter returned some records or excluded all of them. The caller of this method can safely loop over whatever is returned. It will work whether the array is empty or contains a bunch of records.

2 *For a full explanation of Cyclomatic Complexity, see Level Up in the February 2016 issue.* http://phpa.me/2016febzine

Declarative Programming

When we write loops in order to do things, whether it is totaling up values, transforming values, searching through an array to determine if something exists, or any other reason, we're telling the computer "how" to do something. It requires more mental gymnastics to read that code and determine what it is doing. If, instead, we can tell the computer "what" we want and let it work it out, it's much easier to understand. Easy-to-understand code is simpler. Compare these two examples:

```
$result = [];
foreach ($people as $person) {
   if ($person['name'] == 'chad') {
      continue;
   }
   if ($person['cash_on_hand'] > 20) {
      $result[] = $person;
   }
}
```

vs

```
SELECT * FROM people
   WHERE name != 'chad'
      AND cash_on_hand > 20;
```

Both essentially do the same thing, but in the first example, we've told the computer how to get the list we want. We have a loop and two conditionals. But at the end, `$result` should contain just the people who have more than $20 in their pocket and who are not named "chad".

The SQL example does the same thing, but rather than telling the computer how to give us what we want, we just tell it what we want and let it figure out how to do it. The code is more expressive and clear.

Next, On "Hoarders"

Earlier, I mentioned that treating everything the same can help eliminate loops and conditionals. Using a concept of a "Collection," we can treat sets of data more uniformly and we can write code that is more declarative. We can tell the computer what we want and let it sort out the details. I could go into how to derive your own collection, but there's a pretty excellent one in the `Illuminate/Support`[3] package. For a more in-depth look at Collections and how they are awesome, I'd highly recommend picking up Adam Wathan's *Refactoring to Collections* book[4].

Let's take a look at how we can rework the PHP example to utilize declarative programming and collections. Last month, I talked about Transducers, so the map, filter, and reduce concepts should be somewhat familiar. First, let's build some functions to use in those methods.

```
$isChad = function ($person) {
   return $person['name'] == 'chad';
};
```

That's simple enough, but it's limited. It will return true if we pass in a `$person` which has a `name` of `"chad"`. With a tiny bit more code, we can make something that will return a function which will match whatever we send in.

3 *Illuminate/Support:* https://github.com/illuminate/support

4 *Refactoring to Collections:* http://adamwathan.me/refactoring-to-collections/

```
function matchesName ($name) {
   return function ($person) use ($name) {
      return $person['name'] == $name;
   };
}
```

This means that through the use of a `closure` on `$name` we can create functions that match any name. The `$isChad` function can be created like this:

```
$isChad = matchesName('chad');
```

Pretty simple, right? And it means we don't have to write a new function just to match a new name. With a few minor modifications which I'll leave to you, we can make a function which will create a function that matches if a given field matches a given name.

You may notice that in my original examples, we wanted to match people who didn't have the name `"chad"`, but in this code, we're returning `true` if the name is `"chad"`. We'll get to why that is in just a moment.

First, let's build the next helper function for filtering based on people with more than $20 in their wallet. Since we've already seen how to make a function that builds a function, we'll start with that here:

```
function moneyOnHand ($money) {
   return function ($person) use ($money) {
      return $person['money'] > $money;
   };
}
```

This means we can define a function which returns true for people with more than $20 like this:

```
$moreThan20 = moneyOnHand(20);
```

We now have what we need.

```
$result = collect($people)
          ->reject(matchesName('chad'))
          ->filter(moneyOnHand(20));
```

Here's what's happening. First of all, the `collect` function is just a helper function which returns a new instance of the `\Illuminate\Support\Collection` object with the contents being whatever was in the `$people` array. Next, we call the built-in `reject` method, which takes a function that should return `true` if the record should be excluded. So we're telling it to reject any person in the collection named `"chad"`. This returns a brand-new `Collection` object with that rejection in place. The original `$people` is not touched. Next, that `Collection` runs the filter method, which keeps records which return `true` with the supplied function. The `reject` and `filter` methods are the opposite of each other. At the end, the `$result` variable will contain a collection which has just what we asked for. There are no loops or conditionals in our code and it clearly expresses what we want and the intent of it. You can quickly read the code see what's going on, as opposed to the `foreach` loop which takes a bit of time.

There are many other built-in functions we can use on collections to work with the values. If we wanted to just have a list of the names of the people who have more than $20, we could do `$result->pluck('name');`. If we wanted to sum up all the money from these people, then `$result->sum('money');` will give us what we want. We could average it, or slice and dice the data in any other way we want in a simple, clear, and declarative way. This means we have less code and it's clearer, which helps the code stay simple.

Conclusion

Keeping code simple and easy to maintain requires work. It's not easy and it requires constant vigilance when we're changing the code. Keeping the requirements in mind, avoiding assumptions, and keeping responsibilities limited and class implementations ignorant of the inner workings of other things help to keep the code simple. Understanding the problem fully is also one of the best ways to simplify the code, but fully understanding only comes after time. Reducing conditionals and loops by using collections can simplify code and make it more expressive. While it's not easy, working to keep code simple pays off dividends. 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.

Running Mailing Lists with MailChimp and PHP

Matthew Setter

Despite how old email is—the first email was sent in 1971—and despite the popularity of social media, reports repeatedly show that email is as prevalent as ever. Not only is it still prevalent, but it's far and away more profitable than social media ever was, or likely will ever be. Mailing list services are nicely bundled up in slick web interfaces so that users are required to do as little as possible. That's OK for power users—but we're developers! We like to work with APIs, integrating them into our applications, or sometimes just being able to script up repetitive tasks. Today I'm going to show you how to do that with my current favorite mailing list service—MailChimp.

Given email's popularity, it's not surprising that there is a plethora of online services for managing email campaigns, including such luminaries as MailChimp[1], SendGrid[2], and MailGun[3]. Each of these services, as well as the others in the space, have a range of options, including such features as *lists*, *list segmentation*, *campaigns*, and *automated emails*. Some can even send emails at the same time across multiple time zones—and this is just scratching the surface of what's on offer.

What is MailChimp?

I've been using MailChimp for several years. I started about four years ago, after hunting around for a service that provided the features I needed, yet at a price point that I could afford. MailChimp did just that.

MailChimp offers all those features, along with customizable email templates, a range of powerful reports, mobile integration, merge tags, integration with WordPress, Facebook, and Eventbrite, and more, all at a very competitive price. Unlike some of its competitors, for the first 2,000 subscribers and up to 12,000 emails per month, you don't pay a cent.

After that, you can elect to start paying for other features, starting at $10 per/month. Not a bad deal. I've enjoyed using the interface to create campaigns for *Master Zend Framework*, the site I started to share my knowledge of Zend Framework.

The UI makes it pretty simple to create lists, send email campaigns, and run reports, but I've been wondering about automating the process for a while. Instead of logging in and filling out the various fields, testing, and scheduling campaigns, what about being able to script it from the command line?

MailChimp has an API that offers a broad range of functionality. This month I want to step you through some of the core functionality and show you how to automate your email campaigns.

Here's what we'll cover:

1. Create an account
2. Create an email list
3. Add several users to the list
4. Update a user's details
5. Create and send an email campaign to our list

To make all this easier, I'm going to use a package I came across recently: mailchimp-api[4], by Drew McLellan. Mailchimp-api is self-described as:

> *A super-simple, minimum abstraction MailChimp API v3 wrapper in PHP.*

I've been experimenting with the library for a little while now and enjoy using it. There's not a lot to remember, with just a few method calls available. This might seem like a shortcoming. But stick with me: you'll see just how flexible the library is as a result.

The MailChimp API

Before we go too much further, let's have a quick look at the API documentation, so that you get a taste of what's on offer. Navigate, in your browser, to the API documentation overview[5]. There you'll see the details for each endpoint, covering:

- The endpoint regular expression
- The request methods that they accept
- A brief description of each one

1 *MailChimp:* http://mailchimp.com

2 *SendGrid:* https://sendgrid.com

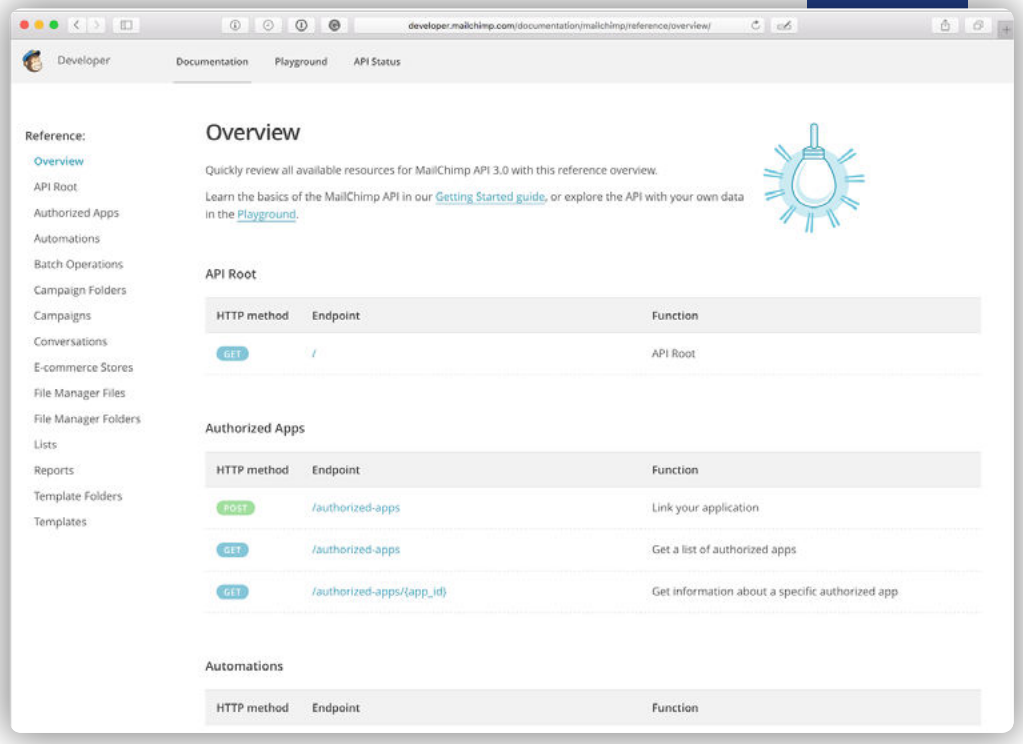
3 *MailGun:* https://www.mailgun.com

4 *Mailchimp API:* https://github.com/drewm/mailchimp-api

5 *Mailchimp API documentation overview:* http://phpa.me/mailchimp-api-overview

You can see an example of the Lists endpoint in Figure 1. It summarizes the purpose of the endpoint and the available methods and sub-resources. Then, as you scroll down to each method, you get a detailed description of the arguments that each endpoint accepts, along with properties, if an argument accepts a complex type. This format applies to all the endpoints within the API.

The MailChimp API documentation **FIGURE 1**



Installation

Like all modern PHP libraries, mailchimp-api can be installed using Composer. To do so, from the root of your project directory run the command:

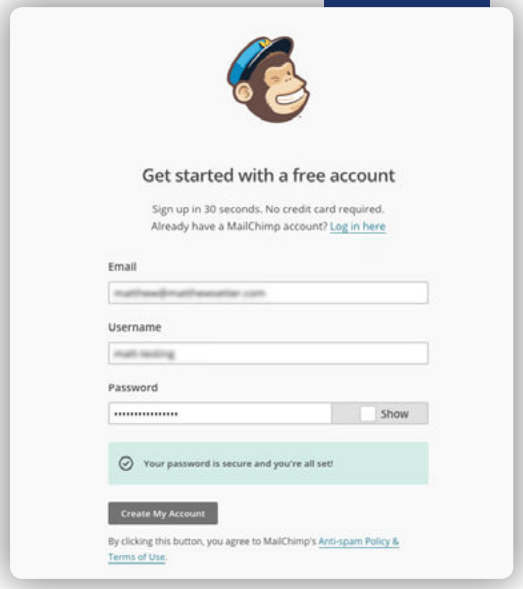
```
composer require drewm/mailchimp-api
```

If you have an existing project, this will add mailchimp-api as a dependency to your project; alternatively, create a `composer.json` file and add it as the first dependency if this is a new project. Regardless of whether this is a new or existing project, the library will be added to the `vendor/` directory.

Creating a MailChimp Account

With the library installed, you now need a MailChimp account to interact with the API. I'll assume that you don't already have one. If you do, feel free to skip this section. That said, go to the MailChimp signup page[6] in your browser, which you can see in Figure 2, and enter your email address, username, and password.

MailChimp signup page **FIGURE 2**



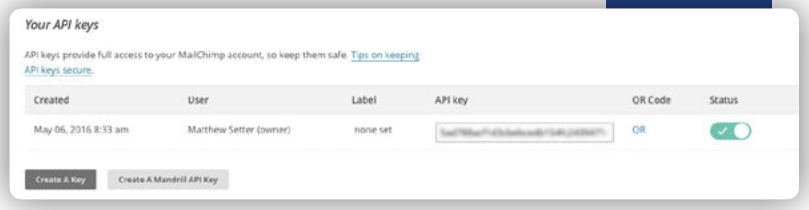
After you've submitted the form, check your email, where you should have received a confirmation email with a link to activate your account. After clicking the link and activating your account, log in[7](https://login.mailchimp.com). You'll see that the account dashboard is pretty bare. You have no campaigns, templates, lists, report data, or automation steps.

Retrieving Your MailChimp API Key

With your account created, before you can interact with the API you need to create an API key. To do this, click the drop-down in the top right-hand corner, where you see your account image and name, and under that, click **Account**.

On the page that you're directed to, click on **Extra** and choose **API Keys**. After the page refreshes (you should be at `/account/api/`), you will see a button labeled **Create a Key** under the section **Your API keys**. Click that and you'll be redirected back to the account page, where an API key will have been created, as you can see in Figure 3. You can click in the cell in the **Label** column to give your API key a descriptive name. This also allows you to use different keys with different applications, which comes in handy if you ever need to revoke access to one.

MailChimp New API Key **FIGURE 3**



6 *MailChimp signup page: https://login.mailchimp.com/signup*

7 *Mailchimp log in*

Copy the key in the *API key* column into the following code below, replacing `your-api-key`. This code will form the basis for all the code we'll be running in the code examples.

```
<?php

require_once ('vendor/autoload.php');

use \DrewM\MailChimp\MailChimp;

$MailChimp = new MailChimp('your-api-key');
```

Creating a MailChimp Mailing List

To create a list, we will send a `POST` request to the `/lists` endpoint. You can see in the documentation[8] that there are a number of options available, including: `name`, `contact`, `permission_reminder`, `campaign_defaults`, and `email_type_option`.

LISTING 1

```
<?php
require_once ('vendor/autoload.php');
use \DrewM\MailChimp\MailChimp;

$MailChimp = new MailChimp('your-api-key');

$result = $MailChimp->post(
   "lists",
   [
      'name' => 'First List',
      'contact' => [
         'company' => 'My Fun Company',
         'address1' => '123 Anywhere Street',
         'city' => 'Berlin',
         'state' => 'Berlin',
         'zip' => '12203',
         'country' => 'Germany'
      ],
      'permission_reminder' => 'true',
      'campaign_defaults' => [
         'from_name' => 'Matthew Setter',
         'from_email' => 'matthew@example.com',
         'subject' => 'Hey People...!',
         'language' => 'en'
      ],
      'email_type_option' => false,
   ]);

print_r($result);
```

Using just these options, the code in Listing 1 provides:

- A name for the list
- The list's contact details
- Campaign defaults, so that I don't have to provide them every time I create an email campaign
- A reminder about where the users signed up to the list, so that they don't think it's spam
- That the list doesn't support choosing the email format

All being well, you'll see a response similar to the output below, which I've truncated for readability:

```
Array
(
    [id] => 73feddfd00
    [name] => First List
    [contact] => Array
        (
            [company] => My Fun Company
            [address1] => 123 Anywhere Street
            [address2] =>
            [city] => Berlin
            [state] => Berlin
            [zip] => 12203
            [country] => US
            [phone] =>
        )

    [permission_reminder] => true
    [use_archive_bar] => 1
    [campaign_defaults] => Array
        (
            [from_name] => Matthew Setter
            [from_email] => matthew@matthewsetter.com
            [subject] => Hey People...!
            [language] => en
        )

    [notify_on_subscribe] =>
    [notify_on_unsubscribe] =>
    [date_created] => 2016-05-09T14:56:20+00:00

)
```


8 *Mailchimp lists API:* http://phpa.me/mailchimp-lists-api

Add Users to the Mailing List

Now that we have a mailing list, we need to add some subscribers to the list. Given the modern practice of double opt in [9], this might seem a strange thing to do—that you can add someone directly to your list.

MailChimp gives you the option to do so. Before we jump into the code sample, please make sure you treat this option with the due care it deserves. Otherwise, you might end up with some disgruntled users. With that said, here's how to add a user:

```
$result = $MailChimp->post("lists/bee5472aef/members", [
   'email_address' => 'matthew@matthewsetter.com',
   'email_type' => 'html',
   'status' => 'subscribed',
   'vip' => true
]);

print_r($result);
```

As the list's endpoint documentation shows, we make a post request to the list's endpoint, specifying the list id, and the suffix of `/members`. For the request options, we're going to pass a few of the available options, mainly just the most pertinent: *email_address*, *email_type*, *status*, and *vip*. This will result in a new VIP list member, who will only receive HTML emails.

Update Users Details

Now, let's imagine that we only had some basic user information, which we provided in the last section, but since then they've provided us with their first and last names. So we're going to add them to the list. To do that, we need to make a patch request and update their details.

As you'd likely expect, the `PATCH` endpoint is only slightly different from the `POST` endpoint, with an MD5 hash of the member's email address being appended to the end (see an example below).

```
$url = sprintf("lists/bee5472aef/members/%s",
                md5('matthew@matthewsetter.com'));
$result = $MailChimp->patch($url, [
   'merge_fields' => ['FNAME'=>'Matthew', 'LNAME'=>'Setter']
]);
```

Here, I've used PHP's `md5` function to create a hash of the email address. If we wanted to, we could also include the hash directly, after retrieving it from the list of members' details, retrieved through a get request as follows

```
$result = $MailChimp->get("lists/bee5472aef/members");
```

Note that in the patch request we only provided the information that we wanted to change. The API is constructed in such a way that `patch` and PUT requests require only the information to be added or changed. All other information will be left intact if not supplied.

Create and Send an Email Campaign

Now that we have a list with a member, it's time to create a campaign and send an email to it. This is going to take a little bit of work—but not too much. First, we create the campaign, by making a post request to the campaign endpoint (see Listing 2).

Here, I've specified that it will be a normal campaign, and will be sent to the mailing list we created earlier. After that, I specified some settings for the campaign, including the subject line, title, who the email will be from, and a reply to address. The subject line and title are not that compelling. In a real campaign, I'd try to make them something that you'd want to open and read. Finally, I've specified some tracking options.

LISTING 2

```
<?php
require_once ('vendor/autoload.php');
use \DrewM\MailChimp\MailChimp;

$MailChimp = new MailChimp('your-api-key');

$result = $MailChimp->post("campaigns", [
    'type' => 'regular',
    'recipients' => [
      'list_id' => 'bee5472aef'
    ],
    'settings' => [
      'subject_line' => 'Here is my subject line',
      'title' => 'here is my title',
      'from_name' => 'Matthew Setter',
      'reply_to' => 'matthew@matthewsetter.com',
    ],
    'tracking' => [
      'opens' => true,
      'html_clicks' => true,
      'google_analytics' => 'regular_campaign_09052016',
    ]
]);
```

Whenever I send out a campaign to my lists, I'm always keen to know how many people opened them, and how many took action on the links in the email. Given that, I've set `opens` and `html_clicks` to true. I'm also interested in some Google Analytics data, and have provided a unique code for that as well.

Create the Email Body

I find this a little strange, but then perhaps it's logical that you have to create the campaign and the campaign email separately. But no matter. To create the body, we have to send a put request to the campaign content endpoint, as in Listing 3.

LISTING 3

```
<?php
require_once ('vendor/autoload.php');
use \DrewM\MailChimp\MailChimp;

$MailChimp = new MailChimp('your-api-key');

$html = <<<EOF
<html>
  <head>
    <title>HTML Body</title>
  </head>
  <body>
    Here is the body of the email
  </body>
</html>
EOF;

$result = $MailChimp->put("campaigns/6f23245f05/content", [
  'plain_text' => 'Here is the body of the email',
  'html' => $html
]);
```

9 *Double opt-in process:* http://phpa.me/mc-double-optin

Here, we've specified a plain text and HTML body for the email. The content isn't that imaginative, but it's enough for a simple email. To retrieve the id of the campaign, we can make a get request to the campaign's endpoint as follows.

```
$result = $MailChimp->get("campaigns");
```

Test the Email

Now that the campaign is created and has a body, it's time to send a test email so that we can verify that the email will render as we expect. To do that, we send a post request to the action's test endpoint, as follows.

```
$result = $MailChimp->post(
   "campaigns/6f23245f05/actions/test", [
      'test_emails' => [
         'matthew@matthewsetter.com'
       ],
      'send_type' => 'html'
]);
```

Here, we've specified the campaign id in the endpoint, then for the request data, specified who we're going to send the test email to, and the send type. As we have both a plain text and HTML body, it would make sense to make two requests, testing both formats.

But for a simple example, making one request for the HTML format is enough. After a few minutes, the test email, which you can see in Figure 4, arrived. It's not all that interesting. But it does the job.

If you're interested in creating more imaginative and creative emails, check out the API documentation on email templates[10].

10 Mailchimp email templates: http://phpa.me/mc-templates

A test email **FIGURE 4**

Conclusion

That's how to use Drew McLellan's mailchimp-api library to interact with MailChimp's API. It might seem strange, at first, to use such a simple library to work with such a full-featured API. But, not having to remember a large amount of functionality, you will find working with the API surprisingly simple and remarkably flexible. I've enjoyed using it, as well as exploring the API further. I hope that you take the time to explore it and make the most of it as well.

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

After the Breach

Chris Cornutt

They're the three words that no software developer (or anyone in a company, really) wants to hear: "We've been hacked." Those words, especially the last one, have a way of ruining anyone's day. Immediately, everyone flies into a panic, and there's usually at least one person shouting: "Shut it down!" Unfortunately, this is a pretty typical scenario, but with a bit of planning, it doesn't have to be you or your organization. Remember, with security and the Web the way they are, it's better just to assume you'll be hacked at some point rather than the opposite. Even if you think you're a small company with no data anyone could possibly want, there's always something that can be gained: user accounts with password reuse or maybe a platform to use for attacking another service.

Some organizations (usually larger companies) go so far as to have a formal, step-by-step process for how to handle a breach. They have scenarios worked out for different kinds of breaches: network, software, physical. I'd love to talk about all of these in detail—maybe in a future column. Instead, I'm going to focus on the one that's a bit more relevant to us PHP developers: software breaches.

First off, let's start with one of the most difficult things to do when this worst case scenario hits…

Don't Panic

I know this is easier said than done, but trust me: running around like your hair is on fire is not the approach to take. Pause, take a breath, and encourage others to do the same. The usual cause of panic is the unknown. When a breach is first discovered, there's a natural human reaction to want to react quickly and make major changes in an effort to make things better. We as engineers and developers are especially bad about this, as our natural instinct is to "just fix it."

Panicking is one of the worst things you can do, as it often leads to bad decisions. You find yourself in the heat of the moment, discovering what kind of damage has been done and whether it's still happening or not. You're worried about what could happen to the company if the attack is successful. You're worried about your own job if something major happens. Worst of all, you're worried about your customers and what they might do when they find out you've been compromised.

This is why you must come up with a plan before you're even attacked, much less breached. It may not be as large and down to every last detail as some of the "big boys," but having some kind of plan in place goes a long way to keeping the stress levels down across the board. This plan should include things such as:

- who to notify and who to get involved in different types of breaches;
- how to handle breaches on certain types of systems in your environment: web servers, load balancers, etc.;
- what kinds of processes need to be followed post-breach; and
- what kind of information to save in order to provide proof, should it be needed, for other interested parties (e.g., the government).

Having these kinds of decisions and information in place ahead of time brings the stress level down a few notches, and the cleanup process gets a little less ambiguous. The right people get involved in the right places, making the remediation an overall simpler process.

Find the Source

Now that you have a plan and have rallied the troops, it's time to stop the issue from happening. Unfortunately, this is the trickiest task of the whole operation. Sometimes it's pretty obvious what happened—a log file somewhere was in the right place at the right time. The log captured exactly the information you needed to find the flaw and to fix it quickly.

Back here in the real world, though, it's never quite that easy. While it is getting easier to log "all the things" and have a lot of context around breaches in your applications, there's only so much you can log effectively and not be overwhelmed by the amount of data. The key here is balance, finding the sweet spot between too much logging and not enough. In the case of a breach, it's always better to have too much—but too much can also hide the true issue if you're not paying attention.

Logs are only one way to try to find the issue. There are many others, including several things you can do on the file system. With many web application breaches, there's usually some kind of evidence left on the system that's been compromised. Clever attackers will probably remove traces that they've been there, but there are a lot of "script kiddie" scripts that can be run that don't bother with cleanup. This means that there could be files in your application's root directory that could provide some insight.

While it's useful to see the create/modify times on these files, the real goodies they can provide are inside them. Unfortunately, the attackers won't make this easy for you. They'll obfuscate these files, taking advantage of the flexibility of PHP itself to do string replacement or to even just make convoluted logic with oddly named variables and minimized code. If you find these kinds of files, the best thing you can do is move them to some kind of quarantine outside of the web application's document root, making note of any ownership, sizes, and dates on the files before issuing the `mv` command. With the files safely in

quarantine, you can then start reversing them and figuring out what they were trying to do.

Some of the most common scripts that are injected this way are:

- webshells that provide a "command line" to your server that can access and write anything the web server user can,
- a "platform" script that can be used to respond to a botnet and relay attacks to other systems, or
- redirects that are injected prior to any content on your site and force the user to another website (maybe with advertising or malware the attackers own).

Once you find the source and have figured out some of the extent of the damage, then comes the next step: locking things down.

Lock It Down

In the above section, I mentioned quarantining everything you can to reduce the immediate risk. You also need to look at the server and other related directories (like `/tmp` for possible session compromise). There could be other files that are compromised too. Remember, they could have accessed and read/modified anything the user was able to work with, so take that into consideration when looking for other resources.

Next you'll need to start rotating things out. Chances are, unless you've been really secure about how you've stored every piece of configuration information, the attacker could have grabbed information such as:

- connection information and credentials for your databases,
- private key contents used to encrypt data in your application, and
- API keys, secrets, and tokens.

These need to be rotated out **as soon as possible** in order to prevent further issues. Additionally, once you identify connected services, you'll also need to check into the logging on those. See if there's been anything anomalous that could be the attacker poking around on other connected systems.

Also, if you can, try to lock things down at a network level too, blocking the attacker IPs and preventing them from accessing the site. Many times this isn't as effective as it might seem, as attackers usually have more than one system at their disposal and could be coming from any number of IPs.

Save What You Can

Now you've cleaned up some of the mess and moved things to quarantine, blocked the attacker from performing any other malicious actions, and rotated out keys and credentials. You'll also need to look at the content of your site to be sure that it hasn't been compromised as well and to try to salvage what you can if so. Depending on the attacker's motive, they may have just been looking to inject their own content into yours, possibly just performing redirects or firing off JavaScript requests in the background to other more malicious scripts.

Much like the other parts of the process, this can be tedious, but it has to be done to ensure the validity of the application and its contents. Fortunately, most of these injections will happen based on a pattern and, once you've found one of them, should be relatively easy to locate in other entries too. Automated scripts can help a lot here and do a "search and destroy" on the contents. Things like "Update Date" columns in your database can help with things like this too.

Fix It

With your content saved and the bad scripts quarantined, you can track down the issue and correct it. I know that this seems like it should be the first step in the process so that the bug isn't exploited again. But trust me: you'll want all of that information down first before you start breaking things in your app. Many times, "fixing it" also means either disabling portions of the application or breaking things for a certain number of users. User-facing issues like these have to be handled delicately. Surprising your users with "Oops!" messages when they're trying to use your service will either instantly tip them off to something being wrong or just flat out make them angry and immediately call support. So be careful as you're fixing your breach that you're not breaking things along the way.

A Feature Versus a Bug

Finally, I want to leave you with one last thing to think about, somewhat related to the last section. When we're talking about security issues, it's a lot easier to fix a security bug than it is to fix a feature. What I mean is that with a bug, you're fixing a flaw (i.e., a problem) with the current application that was either overlooked or not tested enough. With a feature, however, it's going to be more work. It could be a feature that you've provided to your customers as a part of their package but that might be the cause of the security issue. What then? Do you remove the feature entirely, or do you just modify it to fix the issue, potentially breaking other customer integrations?

Conclusion

As always, it's a tricky balance between security, usability, and customer confidence. Plan carefully and follow these steps to help make a breach of your application a bit more calm and, hopefully, easier for everyone involved.

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

Brought to you in conjunction with <?PHPDeveloper.org

June Happenings

PHP Releases

PHP 7.0.8:
http://php.net/archive/2016.php#id2016-06-23-1

PHP 5.6.22:
http://php.net/archive/2016.php#id2016-06-23-2

PHP 5.5.37:
http://php.net/archive/2016.php#id2016-06-23-3

PHP 7.1.0 Alpha 2:
http://php.net/archive/2016.php#id2016-06-24-1

News

Zend Framework Blog: Zend Framework 3 Released!

On the Zend Framework blog they've posted an announcement about the release of the latest version of their framework, the Zend Framework v3. They also mention updates to the skeleton application for this latest release including the work they've done to make the framework and its components more isolated and have fewer dependencies.
http://phpdeveloper.org/news/24138

*Zend Framework Logo used with permission from the Zend Framework project

Stoimen Popov: PHP: Don't Call the Destructor Explicitly

In a new post to his site Stoimen Popov makes the recommendation to not call the destructor explicitly in your code and provides some alternatives. He talks about `__destruct` and its role in PHP's set of "magic methods" and what they exist to do. He then gets into a few examples of what code could look like that uses a destructor and the difference between normal handling calling the destructor explicitly.
http://phpdeveloper.org/news/24130

PHPDelusions.com: Usability Problems of Mysqli Compared to PDO

On the PHPDelusions.com site there's a post that compares the functionality of mysqli to PDO and looks at the differences in their overall usability. The post then breaks it down into sections comparing the functionality between the two database access methods. This is not to say that Mysqli is technically worse than PDO. However, the developer experience with Mysqli is less than ideal since it usually requires wrapping it's functionality in a helper class to reduce the amount of code needed to accomplish a task.
http://phpdeveloper.org/news/24127

SitePoint PHP Blog: Using Halite for Privacy and Two-Way Encryption of Emails

On the SitePoint PHP blog there's a new tutorial posted showing you how to use the Halite package to encrypt the contents of emails. The Halite library sits on top of the libsodium functionality to provide tested, hardened cryptographic results. The tutorial then starts of helping you get the libsodium package installed on your system (assuming it's Unix-based). They then start on the sample application—a basic "email" client able to send/receive messages between users.
http://phpdeveloper.org/news/24119

Christian Mackeprang: Writing Good Code: How to Reduce the Cognitive Load of Your Code

Christian Mackeprang has a post to his site with some ideas about reducing the "cognitive load" of your code—basically making it easier to follow, read, and understand. His tips center around concepts like: following coding standards for consistency; clarification through modularization; overall readability and application structure; good naming on variables and methods/functions.
http://phpdeveloper.org/news/24113

Scotch.io: Understanding Laravel Middleware

The Scotch.io site has posted a tutorial that aims to help you understand middleware in Laravel applications—how they work and how to create ones based on your custom needs. They start by creating a middleware (theirs is DownForMaintenance) and how to register it with Laravel as a valid middleware option.
http://phpdeveloper.org/news/24093

Freek Van der Herten: Building a Dashboard using Laravel and Vue

Freek Van der Herten has a post to his site showing you how to combine Vue.js and Laravel to make a dashboard, a simple way to display statistics and information in a "quick glance" format. He starts with a bit of history around their need for the dashboard and how previous versions (using Dashing) turned out. He gives a high level overview of what he was trying to accomplish and why he chose Vue.js.
http://phpdeveloper.org/news/24069

Davey Shafik: The Syntax of Tech Communities

Davey Shafik has an interesting post to his site sharing some of his views on the "syntax" of different programming communities—PHP, Ruby, Python and Perl. He talks about conferences—both attending and speaking at them—and how the situation differs from community to community. He did some background research on why things are they way they are and how that has influenced the overall focus of each community.
http://phpdeveloper.org/news/24063

Phil Sturgeon: Why Care About PHP Middleware?

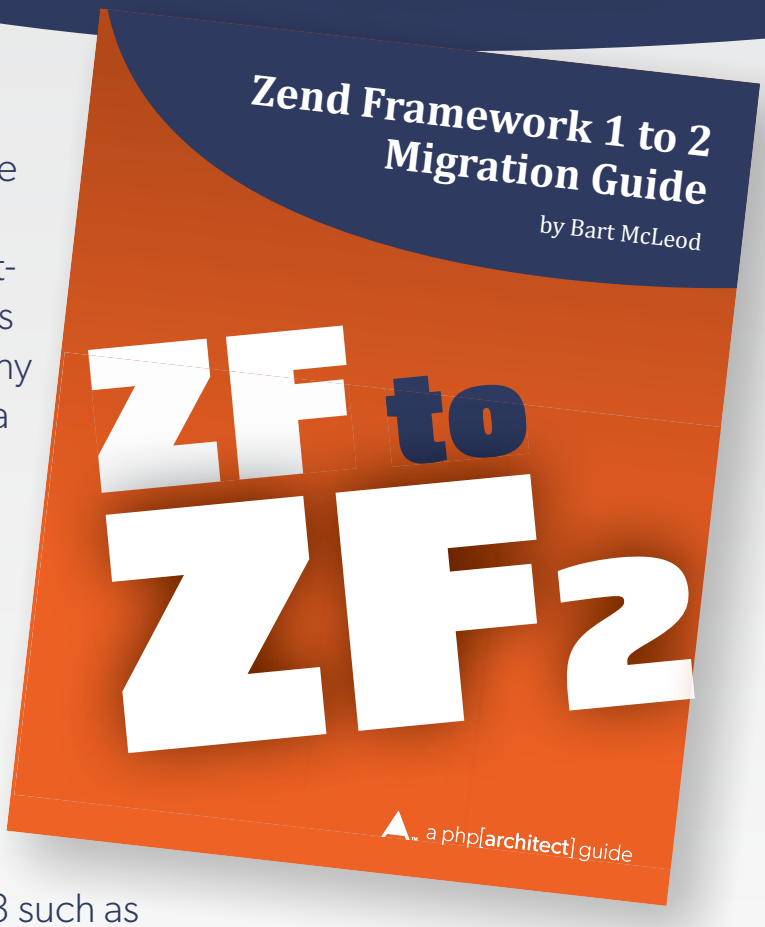
Phil Sturgeon has a post over on his site sharing some of his thoughts on PHP middleware and why he thinks it's worth paying attention to in your applications. He starts with a bit of background about the history of middleware in the PHP ecosystem and where they fit in the overall execution path. He lists out some of the middlewares that have already come out based on this surge in the community including CSRF protection, debugging and rate limiting handling.
http://phpdeveloper.org/news/24046

Frank de Jonge: Finally, File Streams, and Deferred Execution in PHP.

In a post to his site Frank de Jonge looks at a few different topics around the idea of "cleaning up after yourself" when it comes to the use of finally, file streams and deferred execution. He starts by looking at the use of resources for file handling instead of something like file_get_contents. Along with this, however, come "less happy" things to do around cleanup of the resource in case of error or when complete.
http://phpdeveloper.org/news/24054

Zend Framework 1 to 2 Migration Guide

by Bart McLeod



Zend Framework 1 was one of the first major frameworks for PHP 5 and, for many, introduced object-oriented programming principles for writing PHP applications. Many developers looking to embrace a well-architected and supported framework chose to use it as the foundation for their applications. Zend Framework 2 is a significant improvement over its predecessor. It re-designed key components, promotes the re-use of code through modules, and takes advantage of features introduced in PHP 5.3 such as namespaces.

The first release of ZF1 was in 2006. If you're maintaining an application built on it, this practical guide will help you to plan how to migrate to ZF2. This book addresses common issues that you'll encounter and provides advice on how best to update your application to take advantage of ZF2's features. It also compares how key components—including Views, Database Access, Forms, Validation, and Controllers—have been updated and how to address these changes in your application code.

Written by PHP professional and Zend Framework contributor, coach, and consultant Bart McLeod, this book leverages his expertise to ease your application's transition to Zend Framework 2.

Purchase
http://phpa.me/ZFtoZF2

Ageism in the Development Community (from Both Sides)

Eli White

I've been around the block a few times and have the white hair to show for it. With that, I've definitely experienced ageism within my working experience—personally, and having seen it happen to others. Ageism is a problem, and I'm not just speaking about the level where it can become a lawsuit situation, such as firing or hiring someone based on their age, or similar serious accusations. That is a serious problem but isn't exactly the point I want to touch on here.

I want to chat briefly about the much more subtle aspect of ageism in the community, which is simple disrespect between the younger programmers in the field, and those with decades of experience. I'd like to offer a little advice to each side of this feud, in hopes of helping soothe over the differences that appear to exist between these two warring factions.

Advice to the Experienced

To the experienced developers, aka the old farts, the gray hair club, or my people: I know that your reaction is to grab your cane and scream "Get off my Lawn!" But hold off for just a second.

> *In youth we run into difficulties. In old age difficulties run into us.*
>
> *- Beverly Sills*

Typically, the older, more seasoned developers are put off by the fresh-faced newbies who keep pushing the use of new technologies constantly, when you can look at the new technology and go: "Eh, it's just a version of Technology X, we tried that a decade ago and moved on." You know that Technology Y actually solves the problem at hand, is stable, and has been used for a long time.

However, try to keep an open mind and listen to the suggestions that are being made. Sometimes, the next great solution that is BETTER than the software you are used to will be found by the new developers. And they have a strong passion for what they are doing, having not yet been beat down by years of bug-fixing. Yes, you have lots of experience and know what works, but just keep an open mind, and open ear, to what the younger members of your team are suggesting.

Advice to the Newer Developers

To the younger developers, I know that your initial reaction is to tell the grandparents to get out of your way and let you use modern techniques. But again I want to ask you to just pause for a second.

I know that you get frustrated when you find new technologies or new techniques and the older programmers keep shutting down your attempts to use them. They seem fixed in their ways. Wanting to write the same code, use the same tools, and do it all with the same processes that they've used for years.

To you, anything that's five years old is obviously outdated, and there must be a better way to do things now. However, my charge to you is to step back and listen to the experienced devs. While they might not be as quick to pick up a new technology as you, there are reasons. They used to be you. They used to jump onto new concept one after the other, and then over the years got burned by how many of them ended up failing, overly complex, or subpar.

They are battle-scarred, having earned each scar through trials and tribulations that are now hidden deep within them. They have years (decades) of experience, and have seen things start and fail so often that they typically favor a stable, secure, and reliable solution than "yet again" trying something new that might not work out.

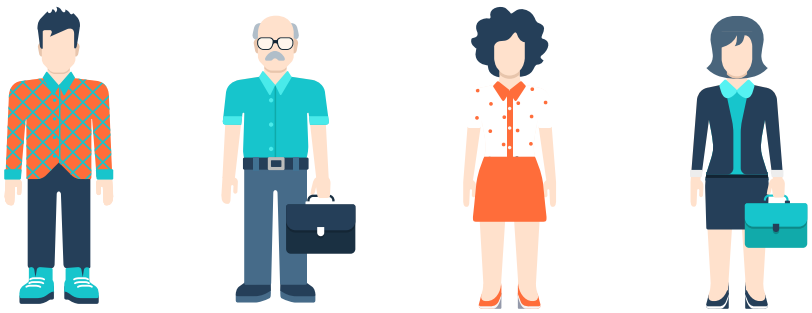
Listen to them, keep an open mind, and use their experience. If they shoot down something that you wanted to do, ask them why. Learn from their experience (and maybe try to convince them otherwise). But realize that their experience is a valuable resource that you can lean on and that will (by definition) take you decades to form yourself.

Conclusion

In the end, the point is that the diversity that exists within our community is a good thing. The more varied backgrounds and experiences that we can rely upon as developers, the more we ourselves learn and the better solutions that we can make.

Talk to each other, learn from each other, and help everyone grow as a developer (and as a person).

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. He had to put away his cane for a short period of time to write this article. @EliW



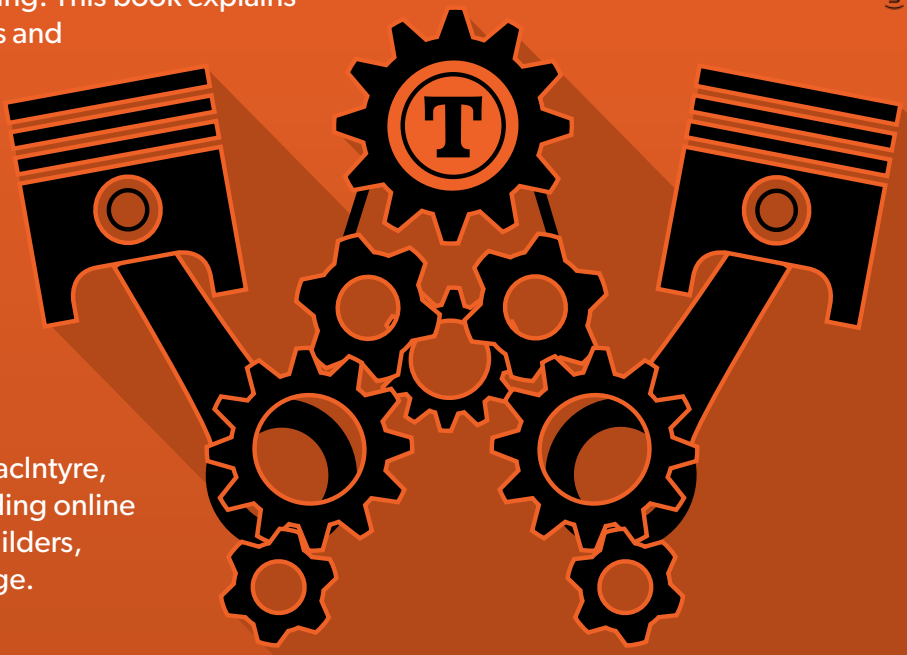
Building Exceptional Sites with WordPress & Thesis

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 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