www.phparch.com

February 2017
VOLUME 16 - Issue 2

php[architect]

ESCAPE THE SERVER

ALSO INSIDE

Strangler Pattern, Part Five: Producer-Consumer Programming in CakePHP/RabbitMQ

Education Station: Instrument Your Apps and Make Them Fly—With Tideways!

Leveling Up: Building Better Objects

Community Corner: Learn to Say No

Security Corner: PHP 7—a Step Closer to a More Secure PHP

finally{}: The Value of Moving Forward

Learning to Code with Minecraft, Part One

Creating a Cross-Platform App With Apache Cordova

Mocking the File System with VfsStream

PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

May 24-26 — ATLANTA

Keynote Speakers:

Gemma Anible
WonderProxy

Keith Adams
Slack

Alena Holligan
Treehouse

Larry Garfield
Platform.sh

Mitch Trale
PucaTrade

Samantha Quiñones
Etsy

Sponsored By:

php women

ShootProof

platform.sh

Save $200 on tickets
Buy ***before*** Feb 18th

tek.phparch.com

php[architect]

CONTENTS

FEBRUARY 2017
Volume 16 - Issue 2

ESCAPE THE SERVER

Features

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

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

Escaping the Server

Who says PHP is only good for serving web pages? Of course, we've evolved past building just simple HTML pages with a dash of Javascript to make it "interactive". PHP's a great fit for powering the APIs on the backend. However, as the core team continues refining and adding to the language, new frontiers and applications open up to us.

As much as we may like the work we do, if all you're doing day in and day out is building the same kinds of websites, you're letting some programming muscles atrophy. Taking on a different project or seeing how someone else applies PHP in a novel environment is a great way to add to your bag of tricks. Who knows? Maybe a couple of months down the road it'll spark the inspiration to solve your latest challenge.

In *Learning to Code with Minecraft, Part One*, Christopher Pitt will show how to build a cooperative code school. If you've ever wondered how you could control a Minecraft server and affect how the players interact with its blocky environment, he'll get you started down that path.

Ahmed Khan looks at *Creating a Cross-Platform App With Apache Cordova* which provides a way to build mobile applications using HTML, CSS, and JS. He'll guide you in hooking one up to a PHP API for managing To do tasks.

Edward Barnard concludes his *Strangler Pattern* series by explaining the code for a worker in *Part Five: Producer-Consumer Programming in CakePHP/RabbitMQ*. He'll also share some lessons learned from deploying microservices to production and a wealth of links for further reading.

Have you wondered how you can test file system operations? In *Mocking the File System with VfsStream*, Gabriel Zerbib introduces us to a package for safely simulating reading, writing, and working with files.

Matthew Setter looks at a new service for monitoring your PHP application's performance in *Education Station: Instrument Your Apps and Make Them Fly—With Tideways!*. He's detailed how to install Tideways to monitor your application to profile it and identify performance bottlenecks. In *Leveling Up: Building Better Objects*, David Stockton explains how to construct objects but—more importantly—the benefits of using an Object-oriented approach. Cal Evan's advises us to *Learn to Say No* in this month's *Community Corner*. It's not an easy lesson to learn, but it is an important one to prevent burn out or worse.

In this month's *Security Corner: PHP 7—a Step Closer to a More Secure PHP*, Chris Cornutt writes about the improvements in PHP 7 which encourage more secure programming practices. Make sure to check it out so ensure you're not still using some outdated habits. Along the same lines, in *finally{}: The Value of Moving Forward* applauds the changes being made and proposed in future PHP releases—including deprecations and removal of features.

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!

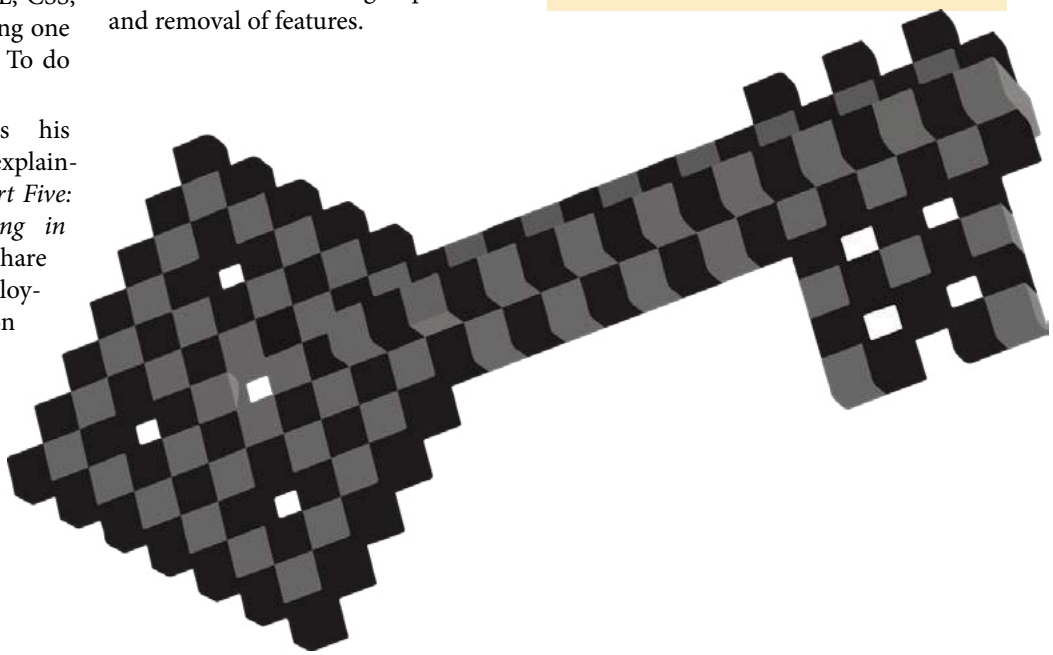
Stay in Touch

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

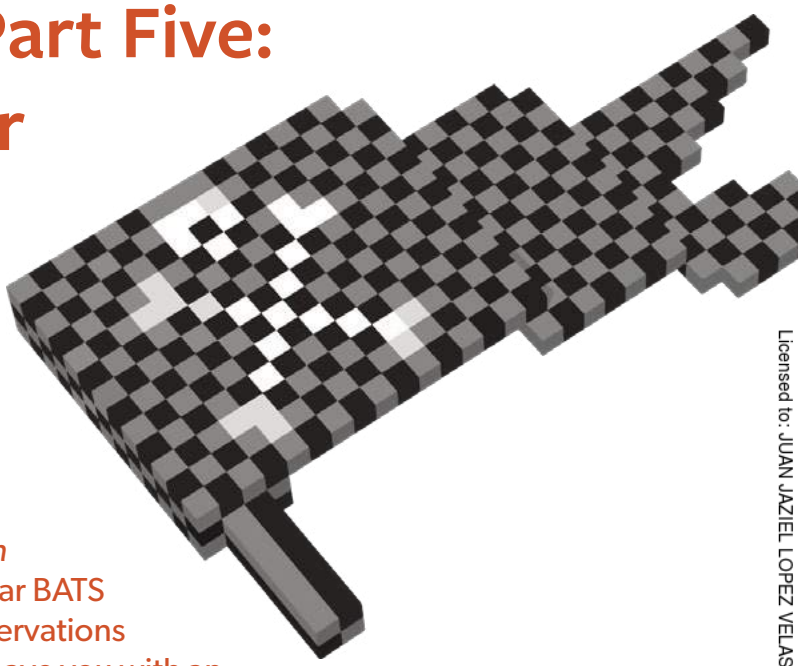
- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/February2017_code



Strangler Pattern, Part Five: Producer-Consumer Programming in CakePHP/ RabbitMQ



Edward Barnard

In this final installment of *The Strangler Pattern* series we walk through the code for a particular BATS worker. We wrap up the series with some observations about production microservices. Finally, we leave you with an extensive reading list for designing distributed messaging systems. We include the reason for each recommendation.

In *Part Five* we are looking at one specific consumer. Be sure to take a close look at the official RabbitMQ tutorials in PHP1. We won't be covering that information here. Instead, we'll focus on the work done by this consumer process. We also assume you're familiar with the earlier articles in this series. We'll wrap up the series with my reading list.

Web Service Call Log

At InboxDollars we record information about each web service request and response in our MySQL *web service call log* table. Given this information is not part of the web service response, it's an excellent candidate for offloading to BATS.

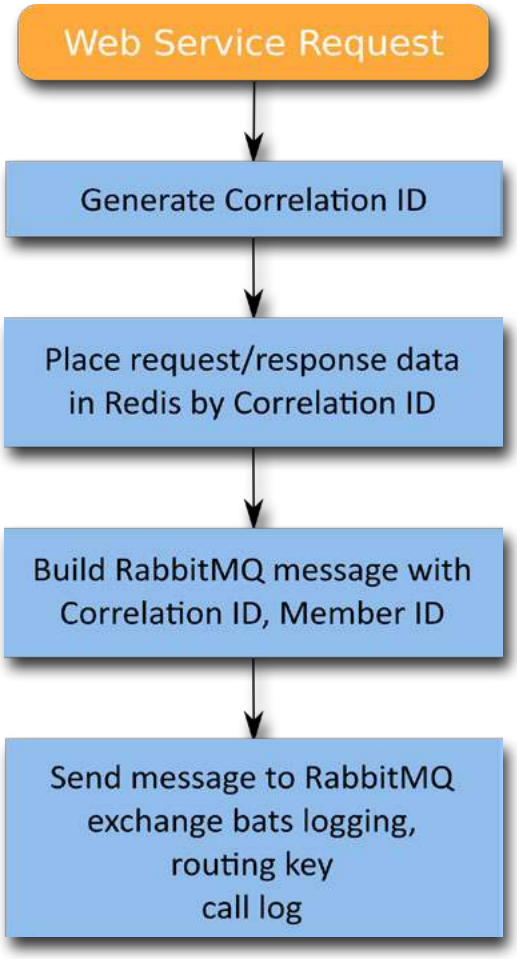
The *producer* code is executed at the end of every web service call. We "fire and forget" the call information into our BATS system.

The *consumer* code is a callback within our *BATS Call Log Worker* which receives incoming messages from RabbitMQ.

Producer

Figure 1 shows the processing flow. This code executes on any of our member-facing load-balanced web servers, at

Figure 1. Producer processing flow

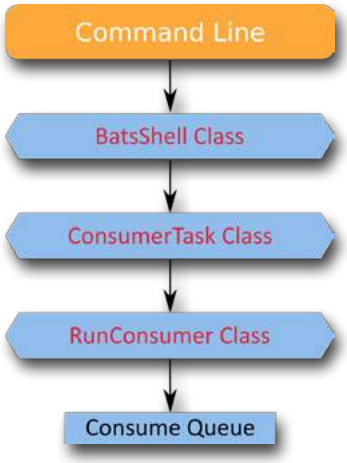


1 *RabbitMQ tutorials in PHP:* http://www.rabbitmq.com/getstarted.html

the end of processing any web service request.

```
exchange = 'bats_logging';
$routingKey = 'call_log';
$data = array( /* ... */ );
$record = json_encode($data);
$client = new PredisCli-
ent(PREDIS_SERVER);
$token = GenerateToken::token();
$client->hset($routingKey, $token,
$record);
```

Figure 2. Class Invocations



The above code generates the Correlation ID as `$token`. We use the routing key as both the RabbitMQ routing key (below) and the Redis hash name (above). The Redis hash field is our Correlation ID and value with that field is the JSON-encoded string containing our call log data.

The code in Listing 1 connects to RabbitMQ and builds a message payload containing the Correlation ID and member ID as a JSON-encoded string. We publish the message to our RabbitMQ exchange with our routing key. Then, clean up with `close()`, `close()`, and `quit()`. That's it!

Consumer

Our *BATS Call Log Worker* is written as a CakePHP 3 Shell[2]. Any framework which supports PHP's Command Line Interface (CLI) would work as well.

Class `BatsShell` (see Listing 2) handles command line parsing and passes control to the `WssCallLog` task:

You'll notice `BatsShell` supports two other workers. I used `Exercise` to load-test the entire BATS system during development. `PermTracker` is another worker we won't mention further here.

The `WssCallLogTask` collects (Listing 3) the command line parameters, adds a few more, and passes control to `RunWssCallLog` which contains the consumer code. I use this same three-step pattern for all BATS workers.

- The BatsShell supports all the BATS workers which are part of this project.
- The XxxTask passes control from the CakePHP ecosystem to the BATS ecosystem.
- The RunXXX class contains the consumer-specific logic, using the BATS ecosystem in support of the distributed messaging model.

`runExecute()` is the entry point and can be seen in Listing 4. The XxxTask calls `execute()`, and `execute()` calls `runExecute()` inside a try/catch block:

The `_initialize()` method does what you'd expect. We open Redis connections to both the Redis server and the local Redis instance. We open MySQL connections and acompile PDO prepared statements.

The calls `connectListen()`, `connect()`, `consume()` connect to, and begin consuming, our RabbitMQ work queue.

The calls `myControlQueue()`, `connectControlQueue()`, `registerMe()`, `consumeControl()` establish our BATS command-and-control connection, and notify the BATS Monitor that we exist.

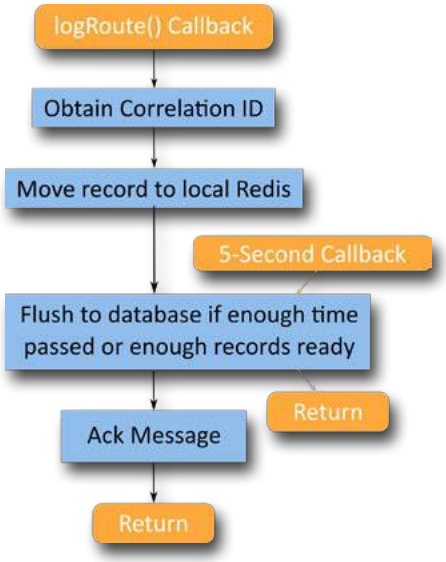
All the work happens during `channelWait()`. We won't ever actually execute `disconnect()`.

So where does the work happen? When we sent the message to RabbitMQ, we specified `['method' => 'log']`. Meanwhile, our current code includes this static array:

```
protected static $routes = [
   'log' => 'logRoute',
];
```

The BATS infrastructure has a callback executed whenever RabbitMQ delivers a message. That callback checks the above array. Given that the current message's "method" is "log", the callback will call the `logRoute()` method:

Figure 3. Callback Flow



2 *CakePHP 3 Shell:* http://phpa.me/cakephp-shell

Listing 1

```php
<?php
$connection = new AMQPStreamConnection( /* ... */);
$channel = $connection->channel();
$channel->exchange_declare(
    $exchange, 'topic', FALSE, TRUE, FALSE
);
$messageParms = [
    'head' => [
        'method' => 'log',
        'time' => time(),
        'batsMessageId' => $token,
        'batsCorrelationId' => $token,
    ],
    'body' => [
        'memberId' => $memberId,
    ],
];
$payload = json_encode($messageParms);
$meta = [
    'content_type' => 'application/json',
    'delivery_mode' => 2
];
$toSend = new AMQPMessage($payload, $meta);
$channel->basic_publish($toSend, $exchange, $routingKey);
$connection->close();
$channel->close();
$client->quit();
```

Listing 2

```php
<?php

class BatsShell extends Shell
{
    public $tasks = ['WssCallLog', 'Exercise', 'PermTracker'];

    public function getOptionParser() {
      return parent::getOptionParser()->description(
            'Run BATS (Batch System) shell commands'
        )->addSubcommand(
            'wss_call_log', [
                'help' => 'Write the web service call log',
                'parser' => $this->WssCallLog
                            ->getOptionParser(),
            ]
        )->addSubcommand(
            'exercise', [
                'help' => 'Exercise the call logger',
             'parser' => $this->Exercise->getOptionParser(),
            ]
        )->addSubcommand(
            'perm_tracker', [
                'help' => 'Permissions Tracker logging',
              'parser' => $this->PermTracker
                            ->getOptionParser(),
            ]
        );
    }
}
```

Listing 3

```php
<?php

class WssCallLogTask extends BatsTask
{
    public function main() {
        $parms = $this->params;
        $parms['caller'] = $this;
        $parms['flushCount'] = 1000;
        $parms['flushTime'] = 30;
        $parms['bulkLimit'] = 1000;
        $parms['timeLimit'] = 3600;
        $runWssCallLog = new RunWssCallLog($parms);
        $runWssCallLog->execute();
    }
}
```

Listing 4

```php
protected function runExecute() {
    $this->_initialize();

    $routingKey = 'call_log';
    $listen = $this->connectListen(
        'bats_logging', $routingKey
    );
    $listen->connect();
    $this->consume($listen, $routingKey);
    $this->myControlQueue();
    $listen->connectControlQueue($this->myControlQueue);
    $this->registerMe();
    $this->consumeControl($listen);
    $listen->channelWait();
    $listen->disconnect();
}
```

```php
public function logRoute(Listen $listen,
                         AMQPChannel $channel, $route = '') {
    $this->_moveCurrentMessage($listen);
    $this->_considerFlush();
    $listen->ackRequest($channel);
}
```

This consumer is, in essence, doing bulk inserts to our MySQL call log table. The rows to insert arrive one message at a time, so we accumulate them in local Redis. When the right time comes, we send them to the database with a single bulk-insert query. The above code:

1. Moves the current message from the Redis server hash (indexed by Correlation ID) to a Local Redis hash with the same hash name, indexed by the same Correlation ID. We delete this hash item (by Correlation ID) from the Redis server.
2. Consider whether now is the time to flush the accumulated rows to the database.
3. Acknowledge the RabbitMQ message. RabbitMQ then

discards the message as we would expect.

This consumer *also* receives a Monitor message every five seconds. That message handler also calls `_considerFlush()` (see Listing 5). That way, even if call log traffic goes idle, we'll see 30 seconds have passed and do the bulk insert.

You'd think moving a message from one place to the other would be simple. It is. But, because we do a periodic sweep to look for "orphaned" messages, the message might have already have been moved. We'll see the sweep next, as we consider whether to flush rows to the database.

The code in Listing 6 says:

- If we have any records to write and either enough time has passed or we have enough records, go ahead and do the bulk insert to the database.
- But before we do the bulk insert, if it's time to do so, run the once-per-hour sweep to check for "orphaned" call log records.

We won't show the sweep code; it merely protects us from an edge case. Here is the bulk insert logic:

- The `insert()` call tells CakePHP 3 to set up the start of the bulk-insert statement.
- The `values()` call, once per row being inserted, tells CakePHP 3 to add that row.

Listing 7 shows the code which creates a Redis `HashKey` iterator to walk through the accumulated rows to insert. This process is the *only* process which adds to this Redis local hash, so when *this* process is iterating through the hash, it's completely safe. Whenever we do a bulk insert, we iterate through the *entire* hash. If a previous worker accumulated rows and then crashed, the next worker would pick up those rows as part of its insert. We take steps to ensure only one instance of this worker is ever running.

We delete the rows out of local Redis immediately after the bulk insert query. There is a small window allowing things to get out of sync, but this should ensure we insert each row exactly once, even if the query fails or the worker otherwise fails.

In other workers, where row integrity is more critical, we use unique keys to ensure the same row cannot be inserted twice.

Production Microservices

Insofar as coding is fun, microservices are fun to write. We have a lot of freedom because each new service can begin as a fresh project. We are not constrained (in theory) by either language or platform, so long as the language can connect to RabbitMQ.

As compared to standard webpage loads, microservices can be *fast*. If it's a daemon, there is no startup time for incoming requests. All of the object instantiations, resource connections, etc., take place at the beginning of the long-running service.

In fact, in our environment, I find a microservice can process a given function one to two *orders of magnitude* faster than can our web servers. The

Listing 5

```
protected function _moveCurrentMessage(Listen $listen) {
    $batsCorrelationId = $listen->batsMessage->correlationId();
    $this->_moveMessage($batsCorrelationId);
    $this->_frontClient->hdel(
       self::$_key, [$batsCorrelationId]
    );
}

protected function _moveMessage($batsCorrelationId) {
    if (!$this->_frontClient
              ->hexists(self::$_key, $batsCorrelationId)) {
       // The record has already been processed
       return;
    }
    $payload = $this->_frontClient
             ->hget(self::$_key, $batsCorrelationId);
    $this->_localClient
         ->hset(self::$_key, $batsCorrelationId, $payload);
}
```

Listing 6

```
protected function _considerFlush() {
    $flushTime = $this->_lastFlushTime + $this->_flushTime;
    $time = time();
    $pending = $this->_localClient->hlen(self::$_key);
    if ($pending && (($flushTime <= $time) ||
                     ($this->_flushCount <= $pending))
    ) {
       if (!$this->_haveSwept &&
           ($time > ($this->instanceStartTime + 1200))
       ) {
          // 20 minutes in, do one sweep for missed
          // call log messages
          $this->_sweepMessages();
          $this->_haveSwept = 1;
       }
       $this->_bulkInsert();
    }
}
```

microservice is running on a VM with minimal RAM, CPU, and disk. The web servers are real hardware stuffed with as much CPU and RAM as will fit in a rack.

Why the difference? The request, when run on the web server, has the startup cost every page load. It runs as part of the monolithic web application with a lot of unnecessary (for this particular request) overhead. It runs on CakePHP 2.x, which is known to be "heavy." It runs in an ecosystem with a *lot* of traffic, whereas the microservices VM is practically idle.

I strongly suspect that a lot of our web server processing could be offloaded as remote procedure calls via RabbitMQ. Let the microservice do one thing and do it well. However, in our environment, that's a bad idea.

For us, it's a matter of infrastructure. We have many years of experience creating a stable environment with our web servers. If a dozen web servers throw traffic at a single underpowered VM, that VM becomes a single point of failure. Bringing everything down with a dumb idea would be a remarkably *bad* idea. We'd need a way to scale our microservice horizontally. Our system administrator was kind enough to point out that I do *not* want to be that person. Better still, he pointed that out *before* I started typing any RPC code. I appreciate that!

My point is, you need to move carefully with microservices. Take advantage of your system administrators' experience. Communicate and educate. When you move to microservices you have, by definition, added a layer of complexity. Change brings discomfort.

Our BATS infrastructure is complex because we realized it's crucial to have complete visibility and control of the microservices from day one. We need to be able to see in order to learn.

Designing Distributed Messaging Systems

Different people have different learning styles. I tend to learn most quickly from books. Even so, the problem can be in finding the right book!

Given experience comes immediately after you needed it, I do my best to learn from *other* peoples' experiences. Having access to other peoples' experience makes *me* look good even though I'm just making stuff up as I go.

Here are the online resources and books I consumed cover-to-cover in learning to build a simple-as-possible distributed messaging system for InboxDollars[3]

Online Resources

1. The official RabbitMQ tutorials[4] RabbitMQ has enterprise-level support available, a mailing list, Stack Overflow watchers, and so on. These tutorials show working code for the various messaging patterns using RabbitMQ.

2. On GitHub, php-amqplib[5] This is the PHP-language wrapper we use in reaching RabbitMQ. The library documentation provides numerous advanced examples. At the very least, skim through the whole `README.md` page to see what capabilities are available.

3. Redis documentation[6]. Redis has good online documentation, and like RabbitMQ, enterprise-level support available. I looked for what I needed and wrote a couple of *learning tests* to verify things worked as I would expect in PHP code.

Listing 7

```
<?php

protected function _bulkInsert() {
    $rows = 0;
    $query = $this->_wssCallLog
                  ->query()->insert(self::$_insert);
    $keys     = [];
    $iterator = new HashKey($this->_localClient, self::$_key);
    foreach ($iterator as $item => $value) {
        $keys[] = $item;
        $record = json_decode($value, TRUE);
        if (is_array($record) && count($record)) {
            $row = [ /* ... */];
            $query->values($row);
            if (++$rows >= $this->_bulkLimit) {
                $query->execute();
                $query = $this->_wssCallLog
                              ->query()->insert(self::$_insert);
                $this->verbose("Inserted $rows call log rows");
                $rows = 0;
                $this->_localClient->hdel(self::$_key, $keys);
                $this->verbose('Local delete: ' . count($keys));
                $keys = [];
            }
        } else {
            $this->alert('Warning',
                         'Failed to decode call log record');
            $this->quiet(
                "Unable to decode call log record $item: $value"
            );
        }
    }
    if ($rows) {
        $query->execute();
        $this->verbose("Inserted $rows call log rows");
    }
    if (count($keys)) {
        $this->_localClient->hdel(self::$_key, $keys);
        $this->verbose('Local delete: ' . count($keys));
    }
    $this->_lastFlushTime = time();
}
```

3 *InboxDollars:* http://www.inboxdollars.com

4 *The official RabbitMQ tutorials:* http://www.rabbitmq.com/getstarted.html

5 *php-amqplib:* https://github.com/php-amqplib/php-amqplib

6 *Redis documentation:* http://redis.io

4. On GitHub, predis[7]. This is the PHP-language wrapper we use in reaching Redis. I found the examples useful, particularly how to correctly iterate through a Redis hash.

Books, E-Books, and Early Access Programs

1. *Building Microservices*[8] by Sam Newman. This book is the place to start. It's a well-balanced discussion of when and why you might consider microservices. Newman discusses breaking up monolithic applications (or not), deployment considerations, and various pitfalls.

2. *The Tao of Microservices*[9] by Richard Rodger. This book is part of the Manning Early Access Program (MEAP) with planned publication December 2016. The half of this book currently available is pure gold. I found the author's insights on "don't repeat yourself" and "technical debt" as applied to microservices to be immediately useful. Microservices require a different way of thinking, and that's why we all need this book.

3. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*[10] by Gregor Hohpe and Bobby Woolf. This 700-page monster has everything. It's old (published in 2003) and describes a different world. As a result, you'll need to do some filtering. I generally skim over the example code but read the discussions of the code. The examples use commercial "middleware" systems common 10-15 years ago. The book is about *Enterprise Integration Patterns* and that was great 10-15 years ago. Today it's the authoritative word on *Designing, Building, and Deploying Messaging Solutions*.

4. *RabbitMQ in Action: Distributed Messaging for Everyone*[11] by Alvaro Videla and Jason J.W. Williams. I enjoy Manning Publications' "In Action" series. When you buy the paperback copy, it comes with a coupon giving you access to the electronic version in all formats. Manning has great developer-centric infrastructure; each book has an author forum as well. I used this book to learn the "RabbitMQ way of doing things" including management and configuration.

5. *RabbitMQ: Patterns for Applications*[12] by Derick Bailey. Bailey put together a series of lessons sent out by email, and then assembled them into an eBook. Each lesson is useful and covers a specific topic. His writing is direct and to the point; for developers, by a developer. You can buy his book as part of a LeanPub bundle.

6. *RabbitMQ Layout: Basic Structure and Topology for Applications*[13] by Derick Bailey. This is where I learned how to lay out our RabbitMQ exchanges. Scroll down the LeanPub web page for a couple of book bundles.

7. *I'm British So I Know How to Queue: Long running RabbitMQ consumers with PHP*[14] by Stuart Grimshaw. This book is available as part of a LeanPub bundle (scroll down the LeanPub web page). This book is short, to the point, and nicely *on* point. If you're new to working with long-running PHP processes, this book is a great introduction. Being RabbitMQ-centric *and* PHP-centric, you can't lose from the experience.

8. *Mastering RabbitMQ*[15] by Emrah Ayanoglu, Yusuf Aytaş, Dotan Nahum. This book gives great insight into how RabbitMQ works under the cover. Most of the book is more than I currently need, but its explanation of RabbitMQ message packets, brokers, and so on are alone worth the price of the book.

9. *RabbitMQ in Depth*[16] by Gavin M. Roy. This book is part of the Manning Early Access Program (MEAP) with planned publication December 2016. I can't wait to read the rest as it comes out! This book is about designing real-world messaging architectures around RabbitMQ. All software involves tradeoffs. This book takes a deep dive into RabbitMQ's tradeoffs, helping you understand what you need for your situation.

10. *Redis Essentials*[17] by Maxwell Dayvson Da Silva and Hugo Lopes Tavares. I now use Kindle versions of books as a developer tool. *Redis Essentials* allowed me to quickly evaluate Redis capabilities and their suitability for our distributed messaging system being designed. This book gave me what I needed, which was how and whether to implement my specific use cases.

Ed Barnard began his career with CRAY-1 serial number 20, working with the operating system in assembly language. He's found that at some point code is code. The language and details don't matter. It's the craftsmanship that matters, and that craftsmanship comes from learning and teaching. He does PHP and MySQL for InboxDollars.com. @ewbarnard

7 *predis: https://github.com/nrk/predis*

8 *Building Microservices: http://phpa.me/building-microservices-book*

9 *The Tao of Microservices: https://manning.com/books/the-tao-of-microservices*

10 *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions: https://www.amazon.com/gp/product/0321200683*

11 *RabbitMQ in Action: Distributed Messaging for Everyone: https://www.amazon.com/gp/product/1935182978*

12 *RabbitMQ: Patterns for Applications: https://derickbailey.com/?p=699*

13 *RabbitMQ Layout: Basic Structure and Topology for Applications: https://leanpub.com/rabbitmq-structures-and-layout*

14 *I'm British So I Know How to Queue: Long running RabbitMQ consumers with PHP: https://leanpub.com/im_british_so_i_know_how_to_queue*

15 *Mastering RabbitMQ: http://phpa.me/mastering-rabbitmq*

16 *RabbitMQ in Depth: https://www.manning.com/books/rabbitmq-in-depth*

17 *Redis Essentials: https://www.amazon.com/gp/product/B00ZXFCFLO*

Mocking the File System with VfsStream

Gabriel Zerbib

Working with files is a common task in many PHP applications, whether for input data to be processed, generated results, or even as a logging tap. But the file system is a persistent resource, and creating unit tests that involve files requires some special care. In this article, you'll learn how to test against a mock file system the pieces of your application which use file operations, instead of involving your actual drive.

In an ideal world, every line of code exists for one very well known reason and is written with a clear purpose.

As developers, we quickly learned the safest way to ensure a piece of code does what it is expected to, is to have it pass through a unit test.

However, testing a program's interactions with files can be tricky, similarly to testing against a database resource. One needs to populate the test environment in a way that can be reproduced, with full control over both the data and metadata. Your file manipulations must be made in a sandbox (you don't want to modify critical files accidentally), and their outcome has to be measurable. You also need to perform clean-up tasks thoroughly after your test scenario has finished.

Fortunately, there are tools that can help, and we will discuss the **vfsStream** library in this article.

Base Principle

vfsStream utilizes a *stream wrapper* to expose a virtual file system to your program. Your unit tests component can manipulate files, folders, and permissions as if they were physically stored entities, whereas they are actually only in memory, much like in a RAM disk.

Using a virtual file system makes it easy for your unit tests to represent a machine-independent folder structure: no need to worry about the physical location of the project on the various developers' workstations. It also enables you to provision borderline case scenarios (for example, to simulate denied permission situations on critical files, when they would actually be fine on the developer's physical machine).

Stream Wrappers

PHP uses the concept of *stream wrappers*[1] to handle various protocols (either built-in or user-defined) as stream resources. For example, you can usually transparently read a file by its URL whether it is found in the local file system, or served by a remote web server (URL begins with `http://`) or by an FTP server (URL in `ftp://`). The scheme component of the address (`http`, `tcp`, and so on) tells PHP how to negotiate the access to the resource.

The default wrapper, when no scheme is specified, is the local file system.

You are probably already familiar with some non-trivial stream wrappers, in addition to our usual `http` friend: in particular the `php` scheme, as in `fopen('php://stderr', 'w')` (which writes to the Linux standard error stream of the PHP process). You may have worked with the `phar://` wrapper as well, to include a script inside a phar archive. And some of you were already aware the content of "folder/file.txt" inside "archive.zip" can be obtained by:

```
file_get_contents(
    'zip://archive.zip#folder/file.txt'
);
```

But we are not limited to the built-in schemes. PHP lets us register our own stream wrappers, designed to handle our custom schemes, through the function:

```
stream_wrapper_register(
    $scheme, $wrapperClass
)
```

The wrapper class you provide must implement the `StreamWrapper` interface[2], which defines all the low-level operations PHP carries out when it tries to reach a resource by your custom scheme. For example, the wrapper should supply a handler for the `stream_eof()` operation, which PHP consults to check if it has reached the end of the accessed stream. (For a terminal-like stream such as `php://stderr`, the end is never reached. It is really up to the wrapper to decide how to handle the stream.

> *Note: There is no actual* `streamWrapper` *base class or interface. The term only exists in the PHP manual: there is nothing to derive from, but your class must comply with the signatures indicated in the documentation.*

vsfStream[3] acts as a stream wrapper (namely `vfs://`), and implements some sort of in-memory file system, complete with (Linux-like) file permissions and other specifics that an application would expect from an actual persistent storage.

Getting Started

Classically, the installation is best achieved by adding a development dependency to your Composer project:

1 *stream wrappers:* http://php.net/en/wrappers

2 `StreamWrapper` *interface:* http://php.net/class.streamwrapper

3 *vsfStream:* https://github.com/mikey179/vfsStream

```
composer require --dev mikey179/vfsStream
```

> *A development dependency is a library which your application does not need at run-time, but rather, that you will use when developing or debugging. In our case, the mock file system library is only useful when writing and executing our unit tests, which means the dependency should not be deployed to production. In the* `composer.json` *file, the development dependencies are declared in the* `"require-dev"` *section.*

Before working with virtual files, you need to set up the virtual file system, by indicating its mounting root. vfsStream defaults to "root/," but I recommend simply `/` instead because it is closer to the common understanding of the top component of a hierarchical file-system-like structure. But, as we will see now, the root is only a logical naming, and it does not make any real difference in the way you write and use tests with vfsStream.

```
use org\bovigo\vfs\vfsStream;

$vfs = vfsStream::setup('/');
```

Then you are ready to populate your file system and to assert its layout and contents with the regular file-related functions, where all the file names are specified using the `vfs://` protocol scheme:

```
// Obtain a scheme-prefixed URL for the following virtual
// absolute path:
$url = vfsStream::url('/test.txt');
// $url is now: 'vfs:///test.txt'

// Use PHP functions to manipulate the file
file_put_contents($url, "Contents of the new file . \n");
```

Or, you may prefer the *fluent* methods provided by the vfsStream library to walk through the directory structure and create the nodes.

```
vfsStream::newFile('test.txt')
    ->at($vfs)->setContent("Contents of the new file.\n");
```

Both flavors create the file `'test.txt'` at the root of your virtual file system. Any piece of your code under test can reach this file by means of the usual PHP functions like `rename`, `unlink`, `fopen` (or most of them—see *Limitations* below). The only requirement is for you to specify the full address: `'vfs:///test.txt'`.

Recipes

We will now see how to use this tool in various common testing scenarios.

Directory Structure

When warming up the test environment, we often need to prepare a pre-existing structure of files and folders, because this is what the system under test expects to see.

This is done by the `create` helper.

```
vfsStream::create([
    'var' => [
        'log' => ['custom.log' => 'Some initial contents']
    ],
    'tmp' => []
]);
```

This code will create the `var` and `tmp` folders under the root of your VFS, and a plain file `custom.log` with some text contents under `var/log`. The argument to the `create` helper is a tree-shaped associative array, whose keys are node names (either folders or files). If the value for the key is another array, then the node is a folder. If the value is a string, then the node is a file whose contents is this string.

Of course, the directory tree thus created is not immutable, and you can always act on it through the PHP file system functions as in Listing 1 (which is the whole point of the library).

Listing 1

```
1. // Rename virtual 'var' folder
2. rename('vfs:///var', 'vfs:///srv');
3. // Equivalent to:
4.
// rename(vfsStream::url('/var'), vfsStream::url('/srv'));
5.
6. // Get a PHP iterator on the entries of virtual /tmp
7. $it = new DirectoryIterator(vfsStream::url('/tmp'));
8.
9. // Print the contents of this virtual file
10. readfile(vfsStream::url('/srv/log/custom.log'));
```

Permissions

The permission system in vfsStream does not pretend to implement a real life environment. Rather, it provides with tools to declare a particular node as accessible or forbidden to the current process (keep in mind that the library is intended for testing).

vfsStream defines a number of mock users and groups for your various scenarios. When creating a virtual file, the owner and group are those of the real process that runs your PHP script (see also `umask` for more details). But you can act on file permissions and ownership with `chmod`, `chown`, and `chgrp` to prove your code.

Therefore, these are the kind of checks you might want to do, see Listing 2.

Listing 2

```
1. $url = 'vfs:///var/log/custom.log';
2. echo intval( is_writable($url) ); // prints 1
3.
4. chmod($url, 0400);
5. echo intval( is_writable($url) ); // prints 0
6.
7. chown($url, vfsStream::OWNER_USER_1);
8. echo intval( is_readable($url) ); // prints 0
9.
10. readfile( $url );                // Issues a warning
```

Running Out of Disk Space

The case when your host system runs out of storage space is often ignored by developers because it seems unlikely enough, and mostly because we lack a test tool to help in this task. An attacker can try a denial-of-service attack by overwhelming a web application with uploaded files, and exploit the resulting faulty behavior. Preparing for this situation should not be left out anymore.

```
//Declare that the virtual storage has only 10 bytes left.
vfsStream::setQuota(10);

file_put_contents('vfs:///tmp/test.txt', 'abcdefghijkl');
// Raises: PHP Warning:  file_put_contents(): Only 0 of 12 bytes
//   written, possibly out of free disk space
```

Handling Large Files

In a similar mindset, loop performance and memory usage when parsing files can be tested by simulating the existence of large files of arbitrary size.

Suppose the class under test opens an input file for reading and an output file for writing, and transforms the data in between, by small chunks.

Proving your logic on a large input file is easily done, using the file factory:

```
$largeFile = vfsStream::newFile('large.txt')
    ->withContent(LargeFileContent::withMegabytes(100))
    ->at($vfs->getChild('tmp'));

echo filesize($largeFile->url()); // prints: 104857600
```

The library will smartly maintain a vacuum-based incomplete structure whose virtual bytes are seen as spaces (`0x20`) by the PHP functions. The actual memory is not filled up and you can still write custom data at arbitrary positions in the file (see Listing 3).

The vfsStream tool stores a fine representation of your simulated data: all those megabytes will still be seen as spaces by `fread` and `copy` etc., but your actual phrase is found there in its proper place.

The `readfile` instruction here, will dump a big lot of white spaces, with 'some real content' after the first 5,000 (take my word for it; Oscar won't let me print the console output here).

Listing 3

```
// open virtual file for modifying its content
$fp = fopen($largeFile->url(), 'r+');
// go to position 5K
fseek($fp, 5000);

// and put some actual phrase there, among all the spaces.
fwrite($fp, 'Some real content');
fclose($fp);

readfile($largeFile->url());
```

Listing 4

```
<?php

class JpegMover
{
    public function searchAndMove($directory, $moveTo) {

// Iterate over the children of specified directory
        $di = new FilesystemIterator($directory);
        foreach ($di as $pathname => $item) {
            // If it's a regular file, with extension 'jpg'
            // then move it to destination folder.
            if ($item->isFile() &&
                ($item->getExtension() == 'jpg')
            ) {
                rename($pathname, $moveTo);
            }
        }
    }
}
```

SPL with VfsStream

Browsing the file system to get a (potentially recursive) list of the items it contains below a specific point, is best done in PHP with the `FilesystemIterator` class (and `RecursiveDirectoryIterator`) from the Standard PHP Library (SPL). The vfsStream package is a perfect companion for testing an application using these classes.

Let's consider a program which searches a specific directory for JPG images (*.jpg file names) in order to move them elsewhere. The code is in Listing 4.

You could test this code using PHPUnit (see Listing 5), by checking that all the JPG files were moved (and only those).

The SPL file system classes, as well as the `rename` file operation function, are indifferent to whether the file system is physical or virtual. They just work as expected.

Build for Testing

You may have noticed, as you drove on a bridge, the metallic or concrete building blocks that compose the piers and girders often present large holes or grooves at intervals.

Construction engineering teaches us that some of these holes are meant to lighten the structure, by simply extracting matter in a way which does not affect the strength or function of the block. But most of them really, are designed solely for quality assurance and transportation purposes. It is easier to carry a heavy block of concrete, move it around, and position it accurately when you can pick it up with a backhoe by its holes and grooves.

The lesson to learn in programming is that it is not less important, when constructing our code for a functional purpose, to make it testable by design.

One good practice that is very dear to me is called Test-driven Development: as soon as a feature is specified, we write a set of corresponding unit tests first, before developing the actual feature. Of course, the tests fail in the beginning,

because the code to achieve it does not exist yet. Then, as the development proceeds, more and more test assertions become green.

Although this methodology is not directly related to our current topic, nor is it mandatory, I can only recommend it warmly in general, and in particular when it comes to working with files. Because, as seen earlier, not everything is possible to abstract the file-related testing. Your effective code must be aware of the possibility to pass through a unit test engine using a virtual file system (which means allowing for prefix-enabled file names).

In particular your code cannot use the magic constant `__DIR__` to locate a data file relative to the running script (because your running script will probably never reside inside the vfsStream virtual space). You should also avoid relying on absolute physical paths for your temp or cache location etc., under penalty of not being able to test your code thoroughly.

Whenever possible, tell your code where to find the "root" of the (potentially virtual) file system rather than making assumptions on its nature and location. Either use some sort of dependency injection container or another configuration mechanism.

Limitations

Although this tool has proven itself very useful to me in several projects, it is worth noting not all the file-related functions are compatible with scheme handlers. Some of them are only a thin encapsulation around their C counterpart, for which the PHP engine won't activate its layer of stream wrappers. There is a list of known issues[4] on the project's page, which may save you some precious time when using vfsStream.

For example, the following will not let you obtain a temporary filename in the virtual file system, even if you specify a URL in `vfs://` as the `$dir` parameter because the underlying, low-level C function is unaware of user-land handlers.

```
tempnam($dir, $prefix)
```

But there is a workaround. As mentioned above, you shall design your application to receive from "outside" the location of the temp folder (whether through configuration files, environment variables or dependency injection so as to provide different bootstrapping in production or unit tests) and you can generate a unique string by different methods (such as `uniqid` or `openssl_random_pseudo_bytes`).

Perhaps the most frustrating limitation in using vfsStream is that the `gz` family of PHP functions (`gzopen`, `gzwrite`, etc.) are low-level encapsulations of the zlib C library. Currently, the zlib library cannot be aware of your PHP-specific vfsStream memory structure. If your application needs to write *gz* data transparently to a file, while keeping low in CPU and not eating up the memory of your PHP script, the `gz` functions are here to help. But then you can't test using virtual files, and you will have to write tests using physical files. (Yes, you still need to write tests!). The workaround here, to be able to test with vfsStream, is to use the higher level *gz* functions (`gzencode`, `gzdecode` etc.) which work in the PHP domain, in memory (not directly on disk), and to read and write to disk after in-memory compression. However, this might not always be a suitable solution for your project.

Listing 5

```
<?php
class MoverTest extends PHPUnit_Framework_TestCase
{
    public function testAllTheFilesWereMoved() {
        // Prepare the virtual filesystem layout
        vfsStream::setup('/');

        // We create two jpg files and one txt file
        vfsStream::create([
            'images' => [
                'move1.jpg' => "",
                'move2.jpg' => "",
                'keep.txt' => "",
            ],
            'target' => []
        ]);

        $dir = vfsStream::url('/images');
        $dest = vfsStream::url('/target');

        $systemUnderTest = new JpegMover();
        $systemUnderTest->searchAndMove($dir, $dest);

        // Assertions
        $this->assertFalse(file_exists($dir . '/move1.jpg'));
        $this->assertFalse(file_exists($dir . '/move2.jpg'));
        $this->assertTrue(file_exists($dir . '/keep.txt'));
        $this->assertTrue(file_exists($dest . '/move1.jpg'));
        $this->assertTrue(file_exists($dest . '/move2.jpg'));
    }
}
```

Conclusion

Even though you may encounter specific situations where vfsStream can't help, it remains handy for most projects. Judging by the number of Packagist downloads and projects dependent on it the library is very stable and popular.

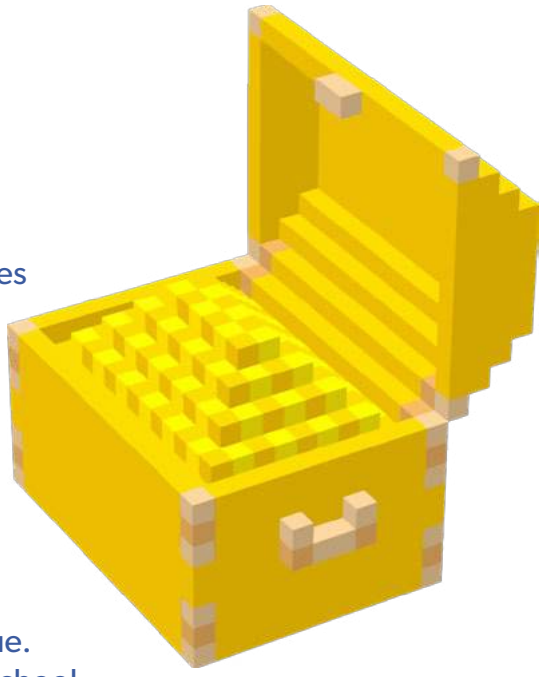
If you've been avoiding writing tests for file system related methods in your code, stop procrastinating! vfsStream is a capable tool for adding unit tests to verify file-system operations work as intended.

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. @zzgab

4 *known issues:* https://github.com/mikey179/vfsStream/wiki/Known-Issues

Learning to Code with Minecraft, Part One

Chris Pitt



I love Minecraft. Not because it's one of the biggest games of our time, or because it has created a community of creative expression. Not because it started as a hobby project, or because I've seen many people (including my son) fall in love with it.

I love Minecraft because, in all these things, it has become one of the best ways to teach people about code.

I want to show you just one of the ways in which this is true. We're going to look at how to build a cooperative code school, using Minecraft, PHP, and JavaScript.

The idea is one person will try to survive, in a Minecraft battle arena, while the other will try to solve code problems. The more keystrokes and/or errors the coder makes, the harder it will be for the other player to survive.

You can find this code on GitHub[1]. I've tested the JS in a recent version of Google Chrome[2], and the PHP code is 7.1 compatible. Minecraft Server[3] 1.11 is the latest version at the time of writing.

Setting up an Async Server

Let's begin with setting up a web server. Traditionally, PHP developers have been able to depend on Apache or NGINX to pass HTTP requests to our PHP scripts. These are excellent tools, designed for scale and extensibility. But they're also designed with a specific architecture in mind.

Apache and NGINX are designed to keep a PHP script running only as long as it takes to complete a single HTTP request.

Our code school requires a different architecture. We're going to communicate with a Minecraft server which requires a persistent connection.

We'll also use web sockets to communicate between the coding interface and the PHP server. If you've ever tried to use web sockets in PHP, you may already be familiar with the architectural limitations servers like Apache and NGINX impose on us.

Let's try something new. There are a few projects which will help us use a different architecture, my favorite being Aerys[4]:

```
composer require amphp/aerys
```

Aerys provides simple interfaces for non-blocking HTTP servers and web sockets. To use it, we'll need to set up a simple configuration script (this is from `config.php`):

```
$host = new Aerys\Host();
$host->expose("*", 1337);
```

This isn't a typical PHP script, in that it isn't run directly. We don't need to include the Composer autoloader[5] or start an event loop[6]. These things are done by the Aerys command line tool, and the configuration script is included during that process.

To start the Aerys command line tool, we need to run:

```
php vendor/bin/aerys -d -c config.php
```

This can be a pain to typed repeatedly, so I've added the following alias to my `.bashrc` file:

```
alias aerys="php ./vendor/bin/aerys -d -c $1"
```

Now I can launch an Aerys server, in debugging mode, with the command below. You should see something like Figure 1.

```
aerys config.php
```

1 *GitHub:* http://phpa.me/assertchris-code-minecraft
2 *Google Chrome:* https://www.google.com/chrome/
3 *Minecraft Server:* https://minecraft.net/en/download/server
4 *Aerys:* https://github.com/amphp/aerys
5 *Composer autoloader:* http://phpa.me/composer-autoloading
6 *event loop:* https://www.sitepoint.com/?p=114483

Image courtesy of BDcraft

Serving Static Files

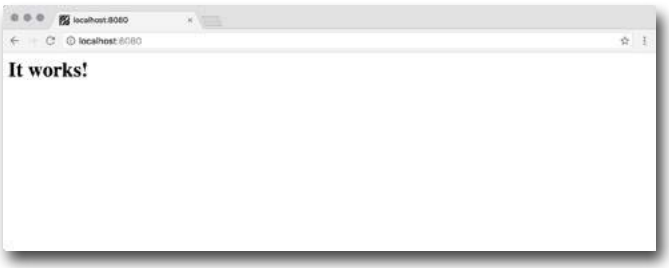
Before we get into Minecraft, there are a couple of things we need to add, to this non-blocking HTTP server. The first is that we need to be able to serve static files so that we can add the required HTML, CSS, and JS (from `config.php`):

```
// ...create host

$public = Aerys\root(__DIR__ . "/public");
$host->use($public);
```

`Aerys\root` is a wrapper around all the code required to serve a folder of static files. It uses the non-blocking file system library[7] to serve files from `public` with the correct mime-types.

Figure 1. Viewing Aerys in Chrome



Listing 1. from public/index.html

```
1. <!doctype html>
2. <html lang="en">
3.     <head>
4.         <meta charset="utf-8">
5.         <title>Learning To Code With Minecraft</title>
6.         <link rel="stylesheet" href="/css/app.css">
7.     </head>
8.     <body>
9.         Hello world
10.        <script src="/js/app.js"></script>
11.    </body>
12. </html>
```

Listing 2. from public/css/app.css

```
1. body {
2.     margin: 0;
3.     padding: 50px;
4.     cursor: default;
5.     font-family: helvetica, arial, sans-serif;
6.     font-size: 14px;
7.     color: #333;
8. }
```

Once we've made the `public` folder, and placed a `.txt` file inside it, we should then be able to access that file via `http://localhost:8080/hello.txt`. Now we can start placing HTML, CSS, and JS files inside `public`, and we'll be able to view them in a browser.

7 *non-blocking file system library:* https://github.com/amphp/file

Let's do that now, by creating the first bits of our coding interface (see Listing 1)

`css/app.css` has just a few styles, to make this page look a little better, see Listing 2.

Enabling Web Sockets

AJAX[8] is a common form of communication, between browser and server. With infrequent requests, it's easy to use. Once we turn up the dial (i.e. multiple messages per second) it starts to become slow. This is because browser and server are constantly repeating a handshake[9]. We can avoid this cost by opening a persistent connection to the server, in the form of web sockets.

This is tricky using traditional PHP architectures, but Aerys opens the door to web sockets! There's an interface we can implement, which tells Aerys about our web socket endpoint, see Listing 3.

This code isn't all that interesting. The `onStart` and `onStop` methods are run as the `Host` starts and stops this `Websocket`. As a browser attempts to connect to the socket, `onHandshake` is run. It checks to see that the origin of the page connecting to the web socket is the same as the server we set up. That limits web socket connections to pages on the same website.

If the origin matches, `onOpen` gets the IP address of the client. We store this so that we can address the client as part of the game. When the client sends a message through the web socket, `onData` receives it. When the client closes the connection, `onClose` lets us clean up any lingering data.

Memory management is a huge part of persistent applications like this one. If we aren't rigorously managing memory usage, memory leaks can bring our application down.

Let's add a bit of code to echo web socket messages back to the clients that send them. This will allow us to test browser web socket connections. This is from `src/Socket.php`:

```
public function onData(int $client, Message $message) {
    $body = yield $message;
    $this->endpoint->send($client, $body);
}
```

This `yield` keyword may look a bit strange, but it's just a way that Amp and PHP `5.6` allow for asynchronous code to look like synchronous code.

We could spend ages learning the finer details of `yield`, but that's not the point of this article. I talk about them in more detail, at Modding Minecraft with PHP[10]

8 *AJAX:* http://phpa.me/AJAX-programming

9 *repeating a handshake:* https://en.wikipedia.org/wiki/Handshaking

10 *Modding Minecraft with PHP:* https://www.sitepoint.com/?p=141561

Listing 3. from src/Socket.php

```
<?php
use Aerys\Request;
use Aerys\Response;
use Aerys\Websocket;
use Aerys\Websocket\Endpoint;
use Aerys\Websocket\Message;

class Socket implements Websocket
{
    private $endpoint;
    private $connections;

    public function onStart(Endpoint $endpoint) {
        $this->endpoint    = $endpoint;
        $this->connections = [];
    }

    public function onHandshake(Request $request,
                                Response $response) {
        $origin = $request->getHeader("origin");

        if ($origin !== "http://localhost:8080") {
            $response->setStatus(403);
            $response->end("<h1>origin not allowed</h1>");

            return NULL;
        }

        $info = $request->getConnectionInfo();

        return $info["client_addr"];
    }

    public function onOpen(int $client, $data) {
        $this->connections[$client] = $data;
    }

    public function onData(int $client, Message $message) {
        // ...do something when we get a message
    }

    public function onClose(int $client, int $code,
                            string $reason) {
        unset($this->connections[$client]);
    }

    public function onStop() {
        // ...nothing to do here
    }
}
```

Listing 4. from public/js/app.js

```
var socket = new WebSocket("ws://localhost:8080/ws")

socket.addEventListener("open", function(e) {
    socket.send("hello from the browser")
})

socket.addEventListener("message", function(e) {
    console.log("message from server: " + e.data)
})
```

Now, let's connect to this web socket from the browser (see Listing 4):

Before we can run this, we need to add `Socket` to our application in `config.php`:

```
$socket = new Socket();

$router = new Aerys\Router();
$router->route("GET", "/ws", Aerys\websocket($socket));
$host->use($router);
```

…and Composer's autoloader:

```
"autoload": {
    "classmap": [
        "src"
    ]
}
```

One last Composer command should do it:

```
composer dump-autoload
```

Now, we should be able to communicate through the web socket. Confirm via the console in your browser's developer tools, see Figure 2.

Interacting with Minecraft

To make our game work, we need to be able to connect to a Minecraft server and execute commands. This is so we can place blocks, teleport players, and spawn monsters.

Head over to the Minecraft website[11] and download the stand-alone server. Run it for the first time:

```
java -Xmx1024M -Xms1024M -jar server.jar --nogui
```

The first time you run the server, it'll stop with a message to accept the terms. We do that by editing `eula.txt`. The server also created a few configuration files. Add the following lines to `server.properties`:

```
enable-query=true
enable-rcon=true
query.port=25565
rcon.port=25575
rcon.password=password
```

These settings allow us to connect to the server, and execute commands as though we were admin server players, walking around in the world. You may have noticed the word `rcon` repeated a few times. It's the name of the protocol[12] by which we will send these commands to the server.

The PHP RCON code is a bit gnarly, so I've gone ahead and created an abstraction[13] around it, which we'll use to execute commands:

```
composer require theory/builder
```

We can make a swift connection to the server, mirroring the settings we added to `server.properties` from `config.php`:

```
$builder = new Theory\Builder\Client(
    "127.0.0.1", 25575, "password"
);

$socket = new Socket($builder);
```

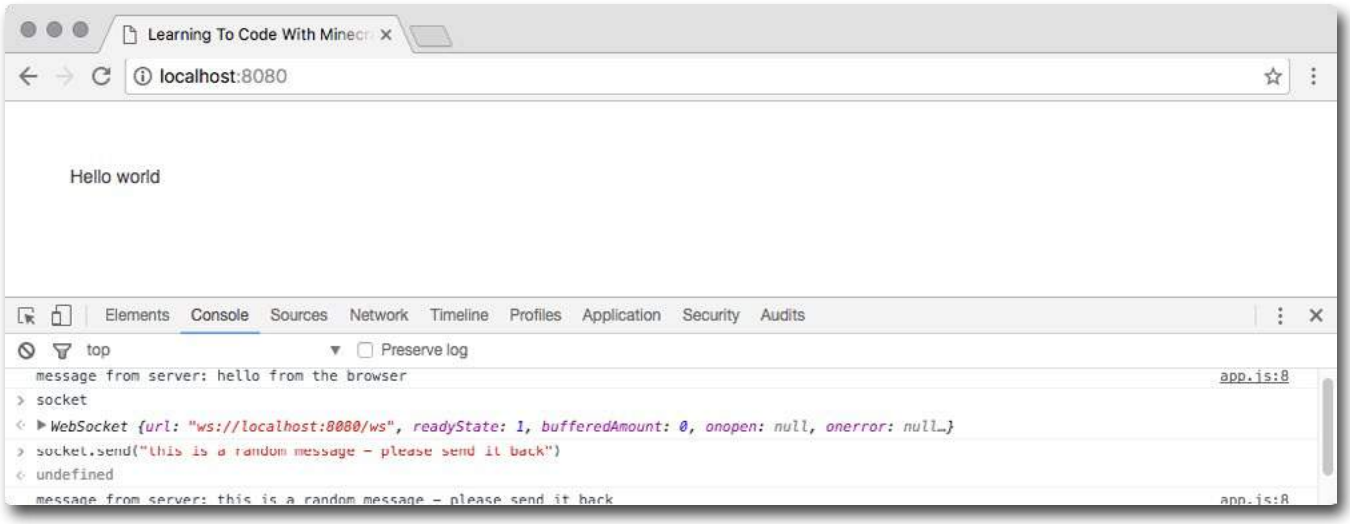
Let's store this builder in `$socket`. This is from `src/Socket.php`:

```
private $builder;

public function __construct(Client $builder) {
    $this->builder = $builder;
}
```

Now, we can listen for players entering and leaving the

Figure 2. Testing the echo server



11 *Minecraft website:* https://minecraft.net/en-us/download/server

12 *the protocol:* http://phpa.me/remote-admin

13 *abstraction:* https://github.com/theorymc/builder

Listing 5. from src/Socket.php

```
1. private $logTime = 0;
2. private $logLines = 0;
3.
4. public function onStart(Endpoint $endpoint) {
5.    // ...set properties
6.
7.    Amp\repeat(function () {
8.       $newLines = yield Amp\resolve($this->getNewLines());
9.       // ...do something with the new log lines
10.   }, 1000);
11. }
12.
13. private function getNewLines() {
14.    $path = __DIR__ . "/../server/logs/latest.log";
15.
16.    $time = yield Amp\File\mtime($path);
17.
18.    if ($this->logTime !== $time) {
19.       $body = yield Amp\File\get($path);
20.       $allLines = explode(PHP_EOL, $body);
21.       $newLines = array_slice($allLines, $this->logLines);
22.       $this->logLines = count($allLines);
23.       return array_filter($newLines);
24.    }
25.
26.    return [];
27. }
```

Listing 6. from src/Socket.php

```
1. private $waitingCoordinates = "1098 16 45";
2.
3. public function onStart(Endpoint $endpoint) {
4.    $this->endpoint    = $endpoint;
5.    $this->connections = [];
6.
7.    Amp\repeat(function () {
8.       $newLines = yield Amp\resolve($this->getNewLines());
9.
10.      foreach ($newLines as $line) {
11.         preg_match("/(\\S+) joined the game/",
12.                    $line, $matches);
13.
14.         if (count($matches) === 2) {
15.            $this->initiatePlayer($matches[1]);
16.         }
17.      }
18.   }, 1000);
19. }
20.
21. private function initiatePlayer($player) {
22.    $this->builder->exec(
23.       "/tp {$player} {$this->waitingCoordinates}"
24.    );
25.
26.    $this->builder->exec("/gamemode a {$player}");
27. }
```

server, and respond to them accordingly. For instance, when they join we want to teleport them to a central waiting area. To achieve these kinds of things, we're going to have to inspect the server logs:

A lot is going on here, so let's look at each step:

1. We use `Amp\repeat` to call another method, every second.
2. The results of that `getNewLines` method are wrapped in `Amp\resolve`, so that the `Generator` it returns is wrapped up as a promise and resolved.
3. Inside `getNewLines` we check the modification time of the latest log file. If it has changed, we get all the times.
4. Then, we slice this down to only the new lines (depending on the number of lines already recorded in the `logLines` property).
5. We set the `logLines` property to the new total number of lines and filter out any blank lines. If the file hasn't been modified, we return an empty array.

Teleporting Players

What do we do with these log lines? To begin with, we can search for the pattern that indicates a new player has joined, and use it to begin a game:

Now, each time we get new messages, we check them to see if they match the pattern of new players joining the server. If so, we teleport them to a set of predefined coordinates and set their game mode to "adventure." This ensures they can't break out of the waiting room, and we can start to build arenas for them.

The techniques we're exploring apply to any un-modded, stand-alone Minecraft server once RCON has been enabled. They can work on any map, but as we'll see, they are particularly suited to "super flat" type maps, as these require the least amount of work to clear before placing arenas and teleporting players.

I've set `waitingCoordinates` *to coordinates that make sense, for my "super flat" world. Your coordinates are bound to be different, and you can discover your current player coordinates by pressing* `F3`*. I suggest you build a glass box and teleport new players inside it while they wait their turn to play.*

Starting Games

The game we're building is cooperative. Now that we can tell when new Minecraft players join the world, we need to put them in a queue system. We need to tell them how to get their friends to join the party.

There are a number of ways we can achieve these things, but we're going to go for the simplest approach. We're going to have a limited number of concurrent active sessions, and everyone else will just wait around until active players survive or perish.

The easiest way to build such a queue is to store player

Listing 7. from src/Socket.php

```
private $players = [];

public function __construct(Client $builder) {
    $this->builder = $builder;
}

private function initiatePlayer(string $player) {
    $this->players[$player] = $player;

    $this->builder->exec("/gamemode a {$player}");

    $this->builder->exec(
        "/tp {$player} {$this->waitingCoordinates}"
    );
}

public function onData(int $client, Message $message) {
    $raw = yield $message;

    $parsed = json_decode($raw, TRUE);

    if ($parsed["type"] === "players") {
        $this->endpoint->send($client, json_encode([
            "type" => "players",
            "data" => $this->players,
        ]));
    }
}
```

names in an array, and associate `Game` objects to them, which we'll use to link their coding partners (Listing 7).

Players are now being added to the queue and the list of online players, during `initiatePlayer`. We've also changed how messages are parsed, to allow for more complex types of messages. This requires a change to the browser code, see Listing 8.

We've added a new web socket command: `players`. When the browser sends this to the server, the server matches the type and returns a list of online players.

We can take this a step further and display these players in a list. That way, coders can pick their friends names to start a game. We need to keep refreshing the list until the coder has picked their buddy's name (to start a game), as shown in Listing 9.

Listing 8. from public/js/app.js

```
socket.addEventListener("open", function () {
    socket.send(JSON.stringify({
        "type": "players"
    }))
})

socket.addEventListener("message", function (e) {
    var parsed = JSON.parse(e.data)

    if (parsed.type === "players") {
        console.log("players:", parsed.data)
    }
})
```

Listing 9. from public/js/app.js

```
var players = []
var $players = document.querySelector(".players")

socket.addEventListener("open", function () {
    setInterval(function() {
        socket.send(JSON.stringify({
            "type": "players"
        }))
    }, 1000)
})

socket.addEventListener("message", function (e) {
    var parsed = JSON.parse(e.data)

    if (parsed.type === "players") {
        players = parsed.data
        $players.innerHTML = ""

        for (var player in players) {
            var $player = document.createElement("li");

            $player.innerHTML =
                "<a href='#' data-player='" +
                    + player + "'>" + player + "</a>"

            $players.appendChild($player)
        }
    }
})
```

As we receive new players from the server, we clear the `.players` list. Then we add a new list item with a link to each player's name. We also need to associate coders with the players' names they click on, see Listing 10.

This technique is called event delegation. Instead of adding event listeners directly to all of the links we create, we add a single event listener to the `body` element. When events bubble up to it, we check whether they match a selector. In this case, the player name links.

If so, we prevent the links' default actions, and send a message to the socket server, telling it to associate this browser

Listing 10. from public/js/app.js

```
var $body = document.querySelector("body")

$body.addEventListener("click", function(e) {
    if (e.target.matches(".players a")) {
        e.preventDefault()

        socket.send(JSON.stringify({
            "type": "join",
            "data": e.target.getAttribute("data-player")
        }))
    }
})
```

Listing 11. from src/Socket.php

```
public function onData(int $client, Message $message) {
   // ...other stuff

   if ($parsed["type"] === "join") {
      $player = $parsed["player"];

      $this->builder->exec(
         "/w {$player} Your friend has joined"
      );

      $game = new Game($player, $client);
      $this->games[$player] = $game;
   }
}

public function onClose(int $client, int $code,
                        string $reason) {
   unset($this->connections[$client]);

   foreach ($this->games as $player => $game) {
      if ($game->hasCoder($client)) {
         unset($this->games[$player]);

         $this->builder->exec(
            "/w {$player} Your friend has left"
         );
      }
   }
}
```

session with the named player. It avoids memory leaks associated with existing event listeners on elements removed from the DOM, and missing listeners on new elements added through AJAX and web sockets.

> **Event delegation is one of those tricks that will help you avoid an entire category of problems you never knew you had.**

We need to link the two together, and let the player know their friend has joined (see Listing 11).

When a coder clicks on a player's name, they send a message to the server. This message leads to a new `Game`. We'll use it to hold the state for each game, later on. For now, it just acts as a link, as shown in Listing 12.

Listing 12. from src/Game.php

```
<?php

class Game
{
   public $player;
   public $coder;

   public function __construct(string $player, int $coder) {
      $this->player = $player;
      $this->coder  = $coder;
   }

   public function hasPlayer(string $player) {
      return $this->player === $player;
   }

   public function hasCoder(int $coder) {
      return $this->coder === $coder;
   }
}
```

When the browser tab is closed (or reloaded), the web socket connection is closed. This means the `onClose` method is called. We use this notification to remove the active game and notify the player that their friend has left.

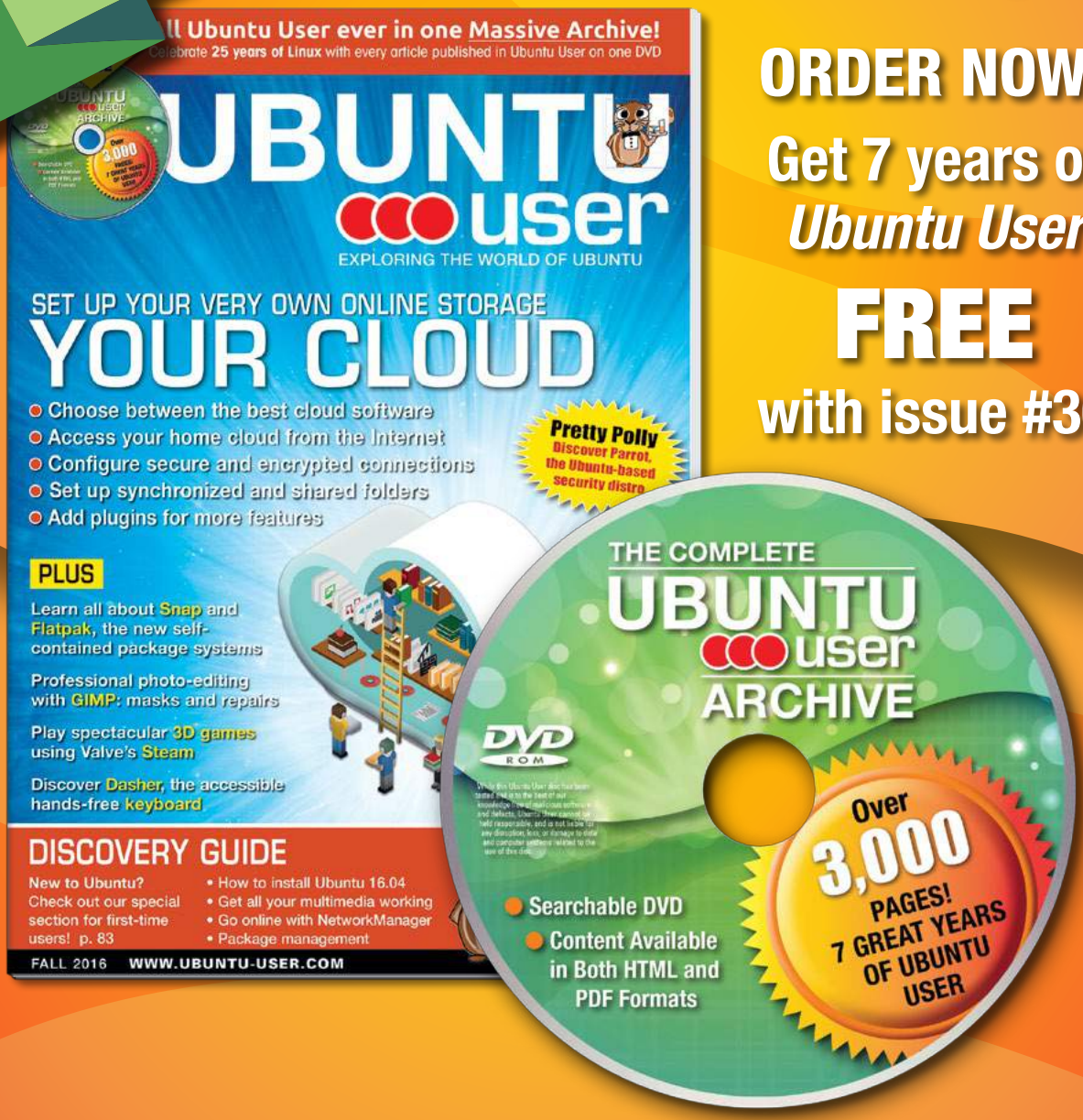
That's It for Now...

In this article, we've seen the beginnings of a much larger game. We've put together a small, but interesting asynchronous PHP HTTP and web socket server. We've even connected it to a JS front-end so players can connect with their coder friends.

Next time we'll get into the mechanics of each game, the coding challenges that pop up, and the awards and achievements we'll present to each team.

Christopher Pitt is a developer and writer, working at https://io.co.za. He usually works on application architecture, though sometimes builds compilers or robots. @assertchris

Celebrating 25 Years of Linux!



ORDER NOW!

Get 7 years of *Ubuntu User*

FREE

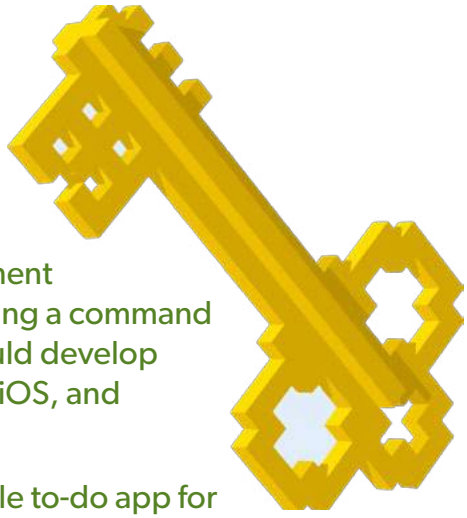
with issue #30

***Ubuntu User* is the only magazine for the Ubuntu Linux Community!**

BEST VALUE: Become a subscriber and save 35% off the cover price!
The archive DVD will be included with the Fall Issue, so you must act now!

Order Now! Shop.linuxnewmedia.com

Creating a Cross-Platform App With Apache Cordova



Ahmed Khan

Apache Cordova is very popular mobile application development framework that offers several APIs for creating applications using a command line interface with HTML, CSS, and JS. Using Cordova, you could develop applications for popular mobile platforms including Android, iOS, and Windows Phone.

In this tutorial, I will demonstrate how you could create a simple to-do app for Android using Cordova. I will use MaterializeCSS for the design and JS to interact with the server for CRUD operations of the app. For server side services, I will use PHP for CRUD operations and MySQL to save the data. For the REST API, I will use Slim Framework.

Creating Server Side Service

Create the Database

The first step is the creation of databases and tables for the users and to-do data items. For this, log in to the MySQL manager you are using (I prefer phpMyAdmin), create a new database and name it `CordovaApp`. Create a new table with the following schema:

```
CREATE TABLE `todo` (
  `id` int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `todo` varchar(255) NOT NULL,
  `category` varchar(255) NOT NULL,
  `description` varchar(255) NOT NULL,
  `create_date` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

This table will handle the to-do data items.

Next, write the PHP code connecting to the database and handling CRUD operations.

Connect to the Database

Several classes will connect to the database, perform CRUD operations for the users and the to-do data items.

Let's start with Connection class. Create a new folder in the localhost's root directory and name it `cordovaapp`. Next, create a new folder and name it `TodoRepository`. Inside `TodoRepository` create a new folder called `db` and create a new file inside it named `config.php`. In this file, shown in Listing 1, we will define the database configuration settings. You can also configure if Slim should show any errors, which is useful on a development site but should be off on a production server.

Inside `TodoRepository` create a new folder `Api`, inside `Api` create a new folder `Db`, and then create a new file in it named `Connection.php`. Paste the code from Listing 2 inside.

This class will be used for connecting to the database. Where `TodoRepository\Api\db\` is our subnamespace.

> *To test the API while you're developing the to-do application, you can use PHP's built-in web server by using the command below from the API application directory.*
>
> `php -S localhost:8080`

CRUD Class for To-Do Data Items

Create a new file in the`Api` folder and name it `Todo.php`. Paste the code shown in Listing 3 in it.

Where `TodoRepository\Api` is the main namespace of our project, the `Validation` function checks the string which is being sent to `add` or `updateTodo` is not empty. If it's empty, it will throw a *new Exception*[1]. The function `ErrorHandling($e)` handles those errors by returning the required message.

1 *new Exception:* http://php.net/class.exception

Listing 1. config.php

```
<?php
$config = [
    'displayErrorDetails' => true,

    'db' => [
        'servername' =>'localhost',
        'username' => 'YOUR_USER',
        'password' => 'YOUR_SECRET',
        'dbname' => 'cordovapp',
    ],
];
```

Listing 2. Connection.php

```php
<?php
namespace TodoRepository\Api\Db;

class Connection
{
    protected $conn = NULL;

    public function __construct($value) {
        $this->conn = new \mysqli(
            $value['servername'],
            $value['username'],
            $value['password'],
            $value['dbname']
        );

        if ($this->conn->connect_error) {
            throw new \Exception(
                    $this->conn->connect_error);
        }
    }

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

    public function close() {
        $this->conn->close();
    }
}
```

This class will be used for handling the data items of the to-do app. The next step is the creation of a simple API so I could interact with the code.

Suppose you have already installed Composer, if not follow the *guide to install Composer*[2]. Now create a new file in the `cordovaapp` folder and name it `composer.json` and paste the following code inside.

```
{
   "autoload": {
      "psr-4": {
        "TodoRepository\\": "TodoRepository"
      }
   }
}
```

In the above code, we have defined our namespace using *PSR-4*[3] and its location so Composer adds its location in autoload. To learn more about autoloading read the *following article*[4] or watch this *video on YouTube*[5]. In your command line run the following command:

```
composer dump-autoload
```

2 *guide to install Composer:* http://phpa.me/composer-install-guide

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

4 *following article:* http://phpa.me/composer-usage

5 *video on YouTube:* https://youtu.be/VGSerlMoIrY

Listing 3. Todo.php

```php
<?php
namespace TodoRepository\Api;

class Todo
{
    protected $db = NULL;

    public function __construct(Db\Connection $db) {
        $this->db = $db;
    }

    public function createTodo($todo, $cat, $desc) {
        try {
            $con = $this->db->conn();
            $this->validation($todo, $cat, $desc);
            $stmt = $con->prepare(
                "INSERT INTO todo(todo, category, description)
                VALUES (?,?,?)"
            );
            $stmt->bind_param("sss", $todo, $cat, $desc);
            $result = $stmt->execute();
            if (!$result) {
                $sql = $con->error;
                throw new \Exception($sql);
            }
            $this->db->close();
            return TRUE;
        } catch (\Exception $e) {
            return $this->errorHandling($e);
        }
    }

    public function getAllTodo() {
        $result = $this->db->conn()
            ->query("SELECT * FROM todo");

        if (!$result) {
            $error = $this->db->error;
            throw new \Exception($error);
        }
        $this->db->close();
        return $result;
    }

```

See this month's code archive for the full listing

Once the command is finished, you will notice that a new folder vendor has been added in your `TodoRepository` folder. Now whenever we include `vendor\autoload.php` our `TodoRepository` classes will be automatically loaded upon request. Now, let's create our REST API.

Create REST API Using Slim Framework

In this tutorial, I will use Slim Framework to create the REST API. Make sure you understand *installing and creating APIs using Slim Framework*[6] before proceeding any further.

The first step is the installation of Slim Framework using

6 *installing and creating APIs using Slim Framework:* http://phpa.me/slim-rest-api

Composer in the same folder.

```
composer require "slim/slim:\^3.0"
```

The second step is the actual creation of the API.

For this article, I'll assume you're using Apache for your web server. Once you have created a host entry for your site, create a `.htaccess` file to define default index file. Paste the following code in `.htaccess` file

```
RewriteEngine On
RewriteCond %{Request_Filename} !-F
RewriteCond %{Request_Filename} !-d
RewriteRule ^ index.php [QSA,L]
```

Listing 4. index.php

```
<?php
require 'vendor/autoload.php';
require 'config.php';

$app = new Slim\App(['settings' => $config]);

// dependencies
$container = $app->getContainer();
$container['db'] = function ($c) {
    $settings = $c['settings']['db'];
    $db = new TodoRepository\Api\Db\Connection($settings);
    return $db;
};

// routes
$app->options('/{routes:.+}',
    function ($request, $response, $args) {
        return $response;
    }
);

$app->add(function ($req, $res, $next) {
    $response = $next($req, $res);
    return $response
        ->withHeader('Access-Control-Allow-Origin', '*')
        ->withHeader('Access-Control-Allow-Headers',
            'X-Requested-With, Content-Type, '
            'Accept, Origin, Authorization')
        ->withHeader('Access-Control-Allow-Methods',
            'GET, POST, PUT, DELETE, OPTIONS');
});

$app->get('/todo/', function ($request, $response)
use ($container) {
    $todo   = new \TodoRepository\Api\Todo($container['db']);
    $data   = NULL;
    $result = $todo->getAllTodo();
    while ($row = $result->fetch_assoc()) {
        $data[] = $row;
    }

    $result = $data;
    $response->withStatus(200)->write(json_encode($result));
});
```

See this month's code archive for the full listing

Create a new file `index.php` and paste the code shown in Listing 4. Let's take a moment to understand a part of the routing code, see Listing 5.

The above snippet provides access to the API from the app and makes sure there are no *CORS errors*[7]. After that, `GET`, `PUT`, `POST`, `DELETE` headers request to create the to-do CRUD operations from the app. Let us discuss what these headers do:

- `GET` is used to tell the server the request is to read and collect the response.
- `POST` is used to tell the server the request is for saving the data.
- `PUT` is used to tell the server the request is for updating the data.
- `DELETE` is used to tell the server the request is for deleting the data.

We have used two HTTP responses `200` and `422` here. Where status `200` tells the request it's "OK" it means it is processed without any error. Status `422` is for `validation` and means your request is not processed because of errors in data.

Creating the To-Do App Using Cordova

Apache Cordova provides a set of APIs enabling you to develop an application using HTML, CSS, and JS. Cordova has a command line interface that allows you to develop applications for Android, iOS, Windows Phone, and other mobile OS. Read their *official documentatio*[8] to learn more about Cordova.

Before proceeding further, make sure *Cordova is installed*[9] and performing without any hitches on the local machine.

I will now create an Android app using Cordova.

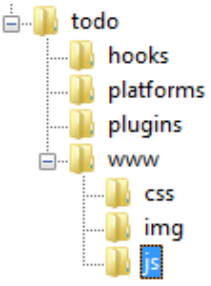
Create a Cordova Project

Create a new directory in your root folder and name it `todoapp`. In this `todoapp` directory create a new project using the Cordova command:

```
cordova create todo
```

Once the project has been created, you will see the directory structure shown in Figure 1.

Figure 1. Project structure



Creating UI for the ToDo App

The `www` folder holds all the data that will be converted into the CROSS application. This app will allow the user to create, update, delete, and read their to-do data.

Open the `index.html` file and

7 *CORS errors:* http://phpa.me/CORS-errors

8 *official documentatio:* http://phpa.me/cordova-overview

9 *Cordova is installed:* http://phpa.me/cordova-cli-install

remove all the code from it. Paste in the code in Listing 6.

Let me explain the above code in some detail.

In the head section, I called MaterializeCss CDN and Jquery. I also added `custom.js` from which the application will interact with JS. Next, I created the navigation section in the page header.

```
<nav>
  <div class="nav-wrapper">
    <a href="index.html"
       class="brand-logo">
          ToDo Lists</a>
  </div>
</nav>
```

Then, we create a list tag with the ID `todo`. The to-do item list will be added dynamically to this tag.

```
<ul class="collection with-header"
    id="todo">
</ul>
```

Next, I created a **float button** which opens a form to add a to-do item. The event

```
onclick="$('\#modal2')
        .openModal();
```

opens `#modal2`.

```
<div class="fixed-action-btn"
     style="bottom: 45px; right: 24px;">
    <a class="btn-floating btn-large red"
      onclick="$('#modal2').openModal();">

<i class="large material-icons">add</i>
    </a>
</div>
```

Next, I created three modals (see Listing 7). In `#modal1`, will display the to-do list description. `#modal2` will contain a form to add a new ToDo item. `#modal3` contains a form to update the to-do items. None of these are displayed initially but appear when the user presses the appropriate button.

The buttons in the modals contain `onclick` events which trigger functions I will create using JS.

Test this code in the browser. You should see something similar to Figure 2.

This is the default view without any to-do items on the list.

Next, let's create the JS functions

Listing 5. routing

```
// routes
$app->options('/{routes:.+}',
   function ($request, $response, $args) {
      return $response;
   }
);

$app->add(function ($req, $res, $next) {
   $response = $next($req, $res);
   return $response
      ->withHeader('Access-Control-Allow-Origin', '*')
      ->withHeader('Access-Control-Allow-Headers',
         'X-Requested-With, Content-Type, Accept, Origin, Authorization')
      ->withHeader('Access-Control-Allow-Methods',
         'GET, POST, PUT, DELETE, OPTIONS');
});
```

Listing 6. index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0"/>
  <!--Import Google Icon Font-->
  <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
        rel="stylesheet">
  <!-- Compiled and minified CSS -->
  <link rel="stylesheet"
        href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.7/css/materialize.min.css">
  <!--Import jQuery before materialize.js-->
  <script type="text/javascript"
          src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
  <!--Let browser know website is optimized for mobile-->
  <script type="text/javascript" src="js/custom.js"></script>

</head>
<body>
<nav>
  <div class="nav-wrapper">
    <a href="index.html" class="brand-logo">ToDo Lists</a>
  </div>
</nav>
```

See this month's code archive for the full listing

Listing 7. Modals

```
<!-- Modal Structure -->
<div id="modal1" class="modal">
  <div class="modal-content">
    <div class="row">
      <div class="col s12 m8">
        <div class="card blue-grey darken-1">
          <div class="card-content white-text">
            <span class="card-title" id="title"></span>
            <div class="card-panel teal">
              <span class="white-text" id="desc"></span>
            </div>
          </div>
          <div class="card-action">
            <div class="chip" id="cat"></div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

See this month's code archive for the full listing

which will get, view, add, update, edit, and delete the to-do list.

JS Functions for To-Do List Operations

There is already a `custom.js` file mentioned in the `head` section of the code.head. Go to the `js` folder and create a new file with the name `custom.js`. I will introduce the functions step-by-step.

First, we need to define the function to retrieve all the ToDo items from the server and bind it to the `<ul>`. This function will also insert view, edit, and delete buttons. In `custom.js`, paste the code from Listing 8.

Since the project is being created on a localhost, I can access the API from the app using the localhost's IP address. For this, make sure the local server is up and running, then change `<ip>` to the IP address of the localhost. You might notice I am appending the view, delete, and edit functions on `onclick`.

Listing 9 shows the code for the function to add a new to-do list item.

This function receives values from the validated form (thus, no empty values) and then send the values to the API to add

Listing 8. init

```
$(document).ready(function () {
    init();
    $('.modal-trigger').leanModal();
});

function init() {
    $.ajax({
        // send select to url.
        type: 'get',
        url: 'http://127.0.0.1/todo/todo/',
        dataType: 'json', // expected returned data format.
        statusCode: {
            200: function (response) {
                $.each(response, function (index, value) {

                    $("#todo").append(
                        "<li class='collection-item'><div>" +
                        value.todo +
                        " <a class='secondary-content' onclick='view(" +
                        value.id +
                        ")'><i class='material-icons right'>send</i></a><a class='secondary-content' onclick='deletes(" +
                        value.id +
                        ")'><i class='material-icons right'>delete</i></a><a class='secondary-content' onclick='edits(" +
                        value.id +
                        ")'><i class='material-icons right'>edit</i></a></div></li>");
                });
            },
            422: function (response) {
                Materialize.toast('Error Getting Data', 4000);
            },
            400: function (response) {
                Materialize.toast('URL Not Found', 4000);
            }
        }
    });
}
```

Listing 9. addtodo()

```
function addtodo() {

    todo = $("#todo2").val();
    desc = $("#desc2").val();
    cat = $("#cat2").val();
    if (todo == "" || desc == "" || cat == "") {

       Materialize.toast('All Fields are Required', 4000);
    }
    else {
       url = 'http://172.17.20.95/todo/todo/';
       $.ajax({
          // Post select to url.
          type: 'post',
          url: url,
          data: {
             todo: this.todo,
             desc: this.desc,
             cat: this.cat
          },
          statusCode: {
             200: function (response) {
                Materialize.toast('Todo Inserted', 4000);
                $('#modal2').closeModal();
                $("#todo").load(location.href + " #todo");
                init();
             },
             422: function (response) {
                Materialize.toast(
                   response['responseText'], 4000);
             },
             400: function (response) {
                Materialize.toast('URL Not Found', 4000);
             }
          }
       });
    }
}
```

Listing 10. edits()

```
function edits(id) {
    url = 'http://172.17.20.95/todo/todo/' + id + "/";
    $.ajax({
       // Post select to url.
       type: 'get',
       url: url,
       dataType: 'json', // expected returned data format.
       statusCode: {
          200: function (response) {
             modeldata = response;
             $.each(response, function (index, value) {
                $("#todo3").val(value.todo);
                $("#cat3").val(value.category);
                $("#desc3").val(value.description);

             });
             $("#upbt").remove();
             $("#bt").append(
                "<a class='waves-effect waves-light btn' \
 id='upbt' onclick='updatetodo(" +
                id + ")'>Update</a>");
             $('#modal3').openModal();
          },
          422: function (response) {
             Materialize.toast(response['responseText'],
                4000);
          },
          400: function (response) {
             Materialize.toast('URL Not Found', 4000);
          }
       },
    });
}
```

the new to-do item.

Next is the function which will edit the to-do items, see Listing 10. Before creating this, I will create the function which will get the to-do item which needs to be edited and display it in the form.

Listing 11 defines the function to update the ToDo list.

The above function gets the updated data from the form and updates the database accordingly.

The function that gets a single to-do item and appends the model and view to it is in Listing 12.

Listing 13 has the function which deletes a single to-do item and appends the model and view to it.

Make sure all these functions are in the `custom.js` file. It is now time to test the app in the browser before converting it to an Android app.

Launching the App

Launch your browser and open the app. Start by adding a new to-do item by clicking the float button. You'll see a modal with our **Add Todo** form (Figure 3)

Fill out the form and click **Add**. The new item will be added to the app and it can be visually confirmed as in Figure 3.

Click the **play** button to view it as in Figure 4.

Exit by clicking anywhere in the app. Now, click the **edit** button to test it. An edit modal will open as in Figure 5.

Once the item has been updated, you will see the most recent version. While you are in the app, test the **delete** button as well. Once you have tested and verified all the functions are working properly, it is time to build the Android app.

Call up the command line and navigate to the Cordova app folder. Use the following line to add the Android platform.

```
cordova platform add android
```

Follow the following link to know what other *platforms Cordova supports*[10]. Once the platform is added, run the following command to build the `.apk` file for Android devices.

```
cordova build android
```

10 *platforms Cordova supports:* http://phpa.me/cordova-platforms-guide

Listing 11. updatetodo()

```
function updatetodo(id) {

    todo = $("#todo3").val();
    desc = $("#desc3").val();
    cat = $("#cat3").val();
    if (todo == "" || desc == "" || cat == "") {
       Materialize.toast("You Can't Update Emtpy Fields",
          4000);
    } else {
       url = 'http://172.17.20.95/todo/todo/' + id + "/";
       $.ajax({
          // Post select to url.
          type: 'put',
          url: url,
          data: {
             todo: this.todo,
             desc: this.desc,
             cat: this.cat
          },
          statusCode: {
             200: function (response) {
                Materialize.toast('Updated Successfully',
                   4000);
                $('#modal3').closeModal();
                $("#todo").load(location.href + " #todo");
                init();
             },
             422: function (response) {
                Materialize.toast(
                   response['responseText'], 4000);
             },
             400: function (response) {
                Materialize.toast('URL Not Found', 4000);
             }
          },
       });
    }
}
```

Listing 12. view()

```
function view(id) {
    url = 'http://172.17.20.95/todo/todo/' + id + "/";
    $.ajax({
       // Post select to url.
       type: 'get',
       url: url,
       dataType: 'json', // expected returned data format.
       statusCode: {
          200: function (response) {
             $.each(response, function (index, value) {
                $("#title").html(value.todo);
                $("#cat").html(value.category);
                $("#desc").html(value.description);

             });
             $('#modal1').openModal();
          },
          422: function (response) {
             Materialize.toast('Error Getting Data', 4000);
          },
          400: function (response) {
             Materialize.toast('URL Not Found', 4000);
          }
       }
    });
}
```

Listing 13. deletes()

```
function deletes(id) {
    url = 'http://172.17.20.95/todo/todo/' + id + '/';
    $.ajax({
       // Post select to url.
       type: 'delete',
       url: url,
       statusCode: {
          200: function (response) {
             $("#todo").load(location.href + " #todo");
             init();
             Materialize.toast('Deleted', 4000);
          },
          422: function (response) {
             Materialize.toast('Error Getting Data', 4000);
          },
          400: function (response) {
             Materialize.toast('URL Not Found', 4000);
          }
       }
    });
}
```

Figure 2.

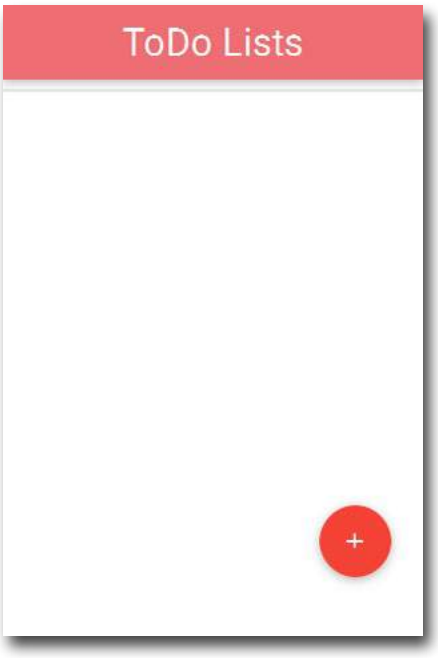


Figure 3. Add Modal

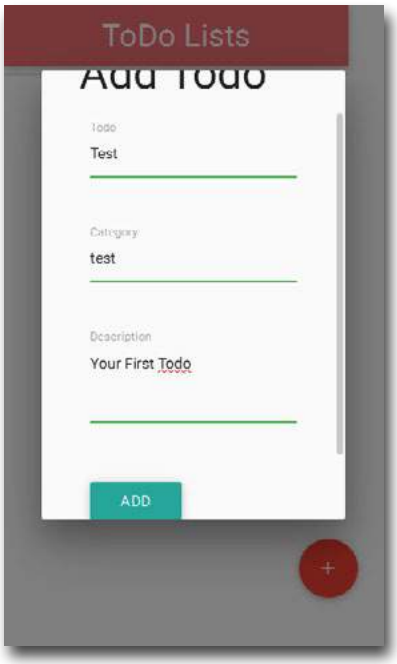


Figure 4. New Todo listed

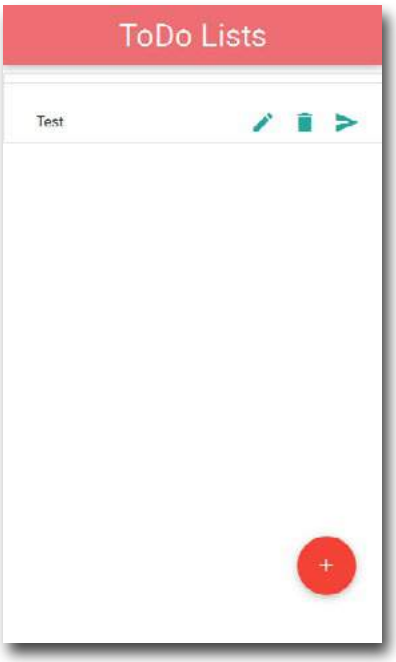


Figure 5. Detail modal

Figure 6. Edit modal

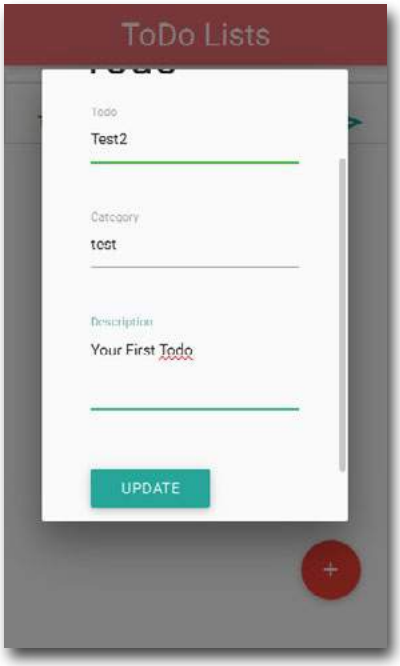


Figure 7. Cordova output

```
[ahmed92]:todo$ cordova platform add android
Adding android project...
Creating Cordova project for the Android platform:
        Path: ../../../../master/applications/cordova/public_html/todo/platforms
/android
        Package: io.cordova.hellocordova
        Name: HelloCordova
        Activity: MainActivity
        Android target: android-23
Android project created with cordova-android@5.2.2
```

Once the build is complete, upload the **.apk** file to an Android device and test all the functionality of the app. If you don't have an Android device handy, or you need to test on multiple devices, you can use an emulator for testing. You can follow this tutorial to *setup an Android emulator*[11] on your PC or you can use *Genymotion*[12] prebuilt emulators. Once the emulator is added you just need to run the following command and your newly created application will run on your emulator:

```
cordova run android
```

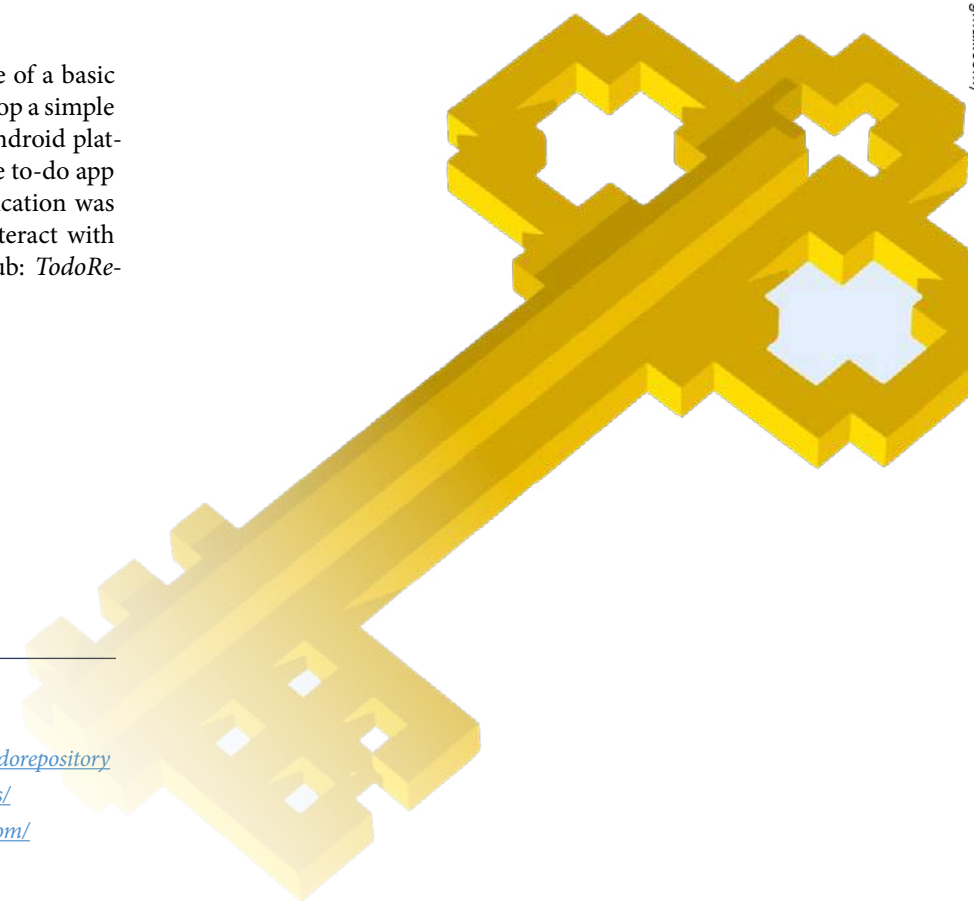
Conclusion

I hope you understood the design and the code of a basic to-do app. In this tutorial, I discussed how to develop a simple cross-platform to-do app for both the web and Android platforms using Cordova. I created a REST API for the to-do app using Slim Framework. The frontend of the application was created in MaterializeCSS and JS was used to interact with the API. The complete code can found on Github: *TodoRepository*[13].

Further Resources:

- *Official Cordova Docs*[14]
- *Find different Cordova packages*[15]
- *Cordova tutorial on TutorialsPoint*[16]
- *Cordova Advanced Topics*[17]
- *Using Cordova with Ionic*[18]
- *Publishing your Cordova Application*[19]

Ahmed Khan is the PHP Community Manager at Cloudways, a hosting company that specializes in optimized PHP hosting services. He writes about PHP, MySQL and covers different tips and tricks related to PHP. He is currently active on Cloudways and other different blogs. When he is not writing about PHP, he likes watching The Flash, Game Of Thrones and is a die-hard fan of DC Comics. You can follow him on @ahmed0627 or connect with him on Facebook.



11 *setup an Android emulator:* http://wp.me/p16cDS-3c

12 *Genymotion:* https://www.genymotion.com

13 *TodoRepository:* https://github.com/ahmedkhan847/todorepository

14 *Official Cordova Docs:* https://cordova.apache.org/docs/

15 *Find different Cordova packages:* https://www.npmjs.com/

16 *Cordova tutorial on TutorialsPoint:* http://phpa.me/tutorials-point-cordova

17 *Cordova Advanced Topics:* http://phpa.me/cordova-advanced-topics

18 *Using Cordova with Ionic:* https://ionicframework.com

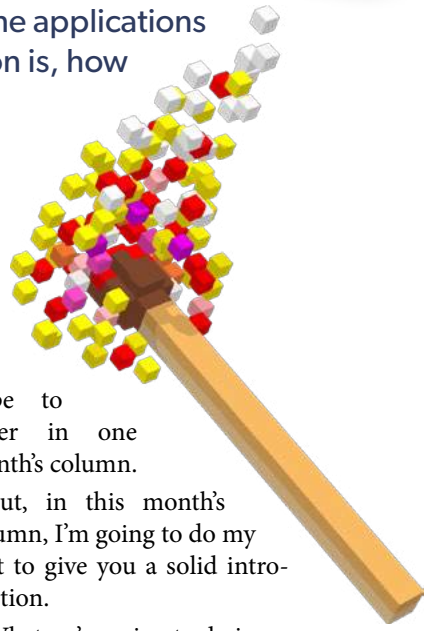
19 *Publishing your Cordova Application:* http://www.9bitstudios.com/?p=1118

Instrument Your Apps and Make Them Fly—With Tideways!

Matthew Setter

If there's one thing you want to do properly if you're serious about the applications you're developing, it's ensuring they perform optimally. The question is, how do you do it?

You can't do it by running it on your laptop and going with your gut. You have to instrument the application, using tools designed to measure application performance; tools which can profile and trace, digging deep into how your application works.



Now, you may be an old hand at developing applications in PHP. You may even have been around long enough to warrant the title of veteran. If so, you've likely already used some of the existing tools, such as New Relic[1] or Xdebug's profiler[2]. However—speaking frankly here for a moment—New Relic focuses on a broad range of languages, not just PHP. And profiling with Xdebug isn't as easy as it, *perhaps*, could be.

So, what about taking a new tool for a spin, one written just for PHP? How about trying one that knows PHP's strengths, weaknesses, and quirks; one which is easy to install and a breeze to use? Sound good? Then learn the fundamentals of Tideways.

What Is Tideways?

While the name isn't too indicative, to quote the Tideways documentation[3]:

> *Tideways saves you time by taking the guesswork out of your app's backend performance. Gain detailed insights, spot performance bottlenecks, and get real-time error detection alerts.*

Tideways provides instrumentation in three areas; they are:

1. **Monitoring and alerting**: including the response time, memory consumption and transaction name, along with PHP version, extensions, and the request URL
2. **Profiling**: following the same format as XHProf[4], it includes SQL queries, garbage collection information, controller names, and template names
3. **Error and exception tracking**: including the type of exception or error, stack traces, and an anonymized version of PHP exception and error messages

Technically speaking, Tideways is a combination of a daemon, a PHP extension, and a PHP wrapper script. Once enabled and configured, as we'll see shortly, the PHP extension sends instrumentation data for an application to the Tideways daemon. The daemon, in turn, sends the collected information to your account on the Tideways server.

If you're looking for a profiler for your PHP applications that has an excellent interface and is relatively easy to install and configure, this is a package you don't want to miss.

Tideways has a significant amount of functionality, a level of which I can't hope to cover in one month's column.

But, in this month's column, I'm going to do my best to give you a solid introduction.

What we're going to do is:

- Create an account.
- Create and boot a Docker container setup to run a PHP application and collect instrumentation data.
- Walk through some of the core features on offer, through the eyes of a side project which I've been working on for some time.

It's my intention that by the time we're done, you'll be able to install it in any project of your own, and be able to start instrumenting and improving its performance.

Getting Started

Like almost anything with computers, the first thing we need to do is create an account. However, if you already have a GitHub account, you can skip doing so and instead login using your GitHub credentials. So, navigate to the login page[5] and click **Login with Github**. After you've done so, you'll land on the

1 *New Relic:* https://newrelic.com

2 *Xdebug's profiler:* https://xdebug.org/docs/profiler

3 *Tideways documentation:* https://tideways.io

4 *XHProf:* https://php.net/book.xhprof

5 *the login page:* https://app.tideways.io/login

Tideways dashboard, which you can see in Figure 1.

By default, you'll see three demo applications which contain demo data. There will be one for *Shopware*, *Symfony2*, and *WordPress*. These give you an overview of what Tideways offers and show off its integration with some of PHP's largest and most successful frameworks and applications.

However, I want to give you a more hands-on feel and show you how to integrate it with a custom application which you're creating.

First, though, in the Tideways dashboard, we need to create an application. To do so, near the top right-hand corner, click the link **Create Application**. This will present a new page, where you can enter the name for the application, as in Figure 2. What this does is internally create a container to store the instrumentation data for one, specific application.

You'll now be back on your dashboard, where you'll see the new application in the application's list. If you click on its name, you'll see the app's dashboard, where there will be no information yet stored for it.

Install Tideways

We next need to install and configure the daemon, PHP extension, and PHP script. There are several setup options[6] covering all the major operating systems, as well as options for AWS Elastic Beanstalk and Heroku.

To provide an installation process as universal as possible, I'm going to step you through creating a Docker setup based on Ubuntu. That way, no matter what operating system you're using, you can follow along. I'm not going to cover the setup in too much depth, as this isn't a Docker tutorial. If you'd like to learn more about Docker, check out *How To Build a Local Development Environment Using Docker*[7]. I'm only going to cover the parts relevant to Tideways and PHP.

However, you can download the setup, from my repository for this month's column from GitHub if you want to skip ahead. Assuming you're following along manually, however, in the root directory of your project, you need to create the following directory structure:

```
docker/
  nginx/
    Dockerfile
    default.conf
  php/
    Dockerfile
    tideways.ini
  tideways/
    Dockerfile
    run.sh
```

Then, in the root directory of our project, create a new file called `docker-compose.yml`. In there, add the code in Listing 1. This will set up the basis of a configuration of three Docker containers. There will be one container for NGINX, one for PHP with the Tideways extension, and one containing the Tideways daemon.

Figure 1. Dashboard

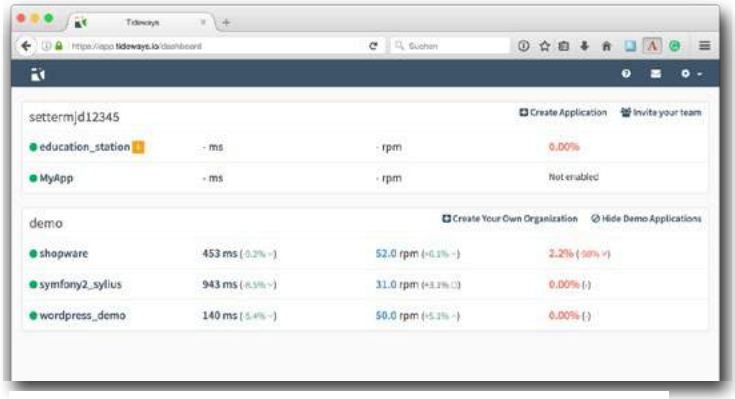
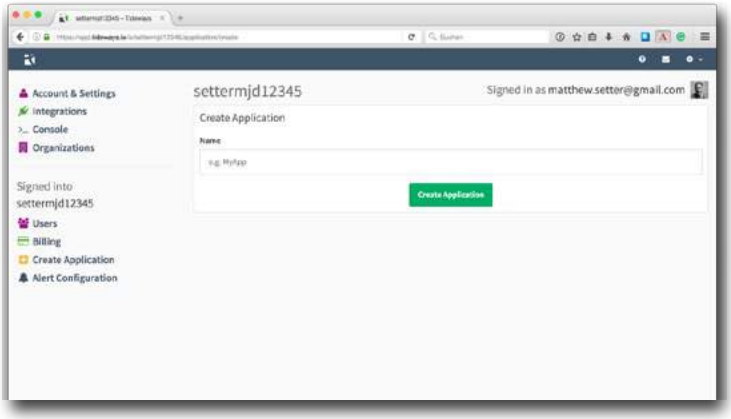


Figure 2. Create Application



Listing 1.

```
1. version: '2'
2.
3. services:
4.   nginx:
5.     image: nginx:latest
6.     ports:
7.       - 8080:80
8.     volumes:
9.       - ./docker/nginx/default.conf:/etc/nginx/conf.d/default.conf
10.    volumes_from:
11.      - php
12.  php:
13.    build: ./docker/php/
14.    expose:
15.      - 9000
16.    volumes:
17.      - .:/var/www/html
18.
19.  tideways:
20.    build: ./docker/tideways
```

6 *setup options:* http://phpa.me/tideways-install

7 *How To Build a Local Development Environment Using Docker:* http://phpa.me/master-zf-docker

In `docker/nginx/Dockerfile`, add the code in Listing 2. What this does is indicate the Docker image the container is based on. In `docker/php/Dockerfile`, add the code in Listing 3. What we're doing here is indicating the Docker image to build the container on. Then, we're adding the `qafoo-profiler` Apt repository to the existing APT repositories and installing the `tideways-php` and `tideways-daemon` packages.

Next, we're installing a few other binaries and PHP extensions which my particular application needs. Feel free to skip or modify these to suit the needs of your code. The final three lines copy the Tideways `.so` extension and the Tideways PHP wrapper script to the PHP extensions directory. These files need to either be there or symlinked so that Tideways will work.

Finally, we're copying the Tideways PHP configuration directives to PHP's configuration directory. You can find these in Listing 4. If you've done any PHP configuration, then this will be familiar to you. It first enables the extension, then sets the API key, UDP and TCP connection URIs.

At this point, you may be wondering where you get your API key from. If so, in the top right-hand corner of the dashboard, you'll see the cog icon. Click that, and then click the first menu which appears, entitled: **Application Settings**. Then, about halfway down the **Application Settings** page, you'll see a section titled **API Key**. There you'll find your API key. Copy and paste it into the configuration.

Lastly, copy the code in Listing 5 into `./docker/tideways/Dockerfile`. Similar to the PHP container configuration, this one installs the Tideways daemon and calls a Bash script to boot the daemon when the container starts. Note the second last directive, `RUN`. I didn't see this in the official setup documentation[8] and encountered a permission denied error anytime I tried to boot the container when it wasn't present. So make it's included!

Finally, copy the code from Listing 6 into `./docker/tideways.run.sh`. This boots the daemon on the standard Tideways ports. If you have any trouble with your setup, double check that you've completed all the steps, and then read through the troubleshooting guide[9] for their suggestions.

Listing 2

```
1. FROM nginx:latest
2.
3. COPY ./default.conf /etc/nginx/conf.d/default.conf
```

Listing 3

```
FROM php:7.0-fpm
 2.
 3. RUN echo 'deb http://s3-eu-west-1.amazonaws.com/qafoo- \
 4.     profiler/packages debian main' > /etc/apt/ \
 5.     sources.list.d/tideways.list && curl -sS \
 6.     'https://s3-eu-west-1.amazonaws.com/qafoo-profiler/ \
 7.     packages/EEB5E8F4.gpg' | apt-key add - && \
 8.     apt-get update && \
 9.     DEBIAN_FRONTEND=noninteractive apt-get -yq install     \
10.        tideways-php tideways-daemon  && \
11.     apt-get autoremove --assume-yes && \
12.     apt-get clean && \
13.     rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
14.
15. RUN docker-php-ext-install pdo_mysql \
16.     && docker-php-ext-install json
17.
18. RUN apt-get update \
19.     && apt-get --yes --force-yes install git \
20.     && apt-get --yes --force-yes install zip \
21.     && apt-get --yes --force-yes install unzip
22.
23. RUN cp /usr/lib/tideways/Tideways.php /usr/local/lib/php/  \
24. extensions/no-debug-non-zts-20151012
25. RUN cp /usr/lib/tideways/tideways-php-7.0.so /usr/local/   \
26. lib/php/extensions/no-debug-non-zts-20151012
27.
28. COPY tideways.ini /usr/local/etc/php/conf.d/40-tideways.ini
```

Listing 4

```
1. extension=/usr/lib/tideways/tideways-php-7.0.so
2. tideways.api_key=
3. tideways.udp_connection=tcp://tideways:8135
4. tideways.connection=tcp://tideways:9135
```

Listing 5

```
 1. FROM phusion/baseimage:0.9.18
 2.
 3. RUN echo 'deb http://s3-eu-west-1.amazonaws.com/qafoo-
profiler/packages debian main' > /etc/apt/sources.list.d/
tideways.list && \
 4.     curl -sS 'https://s3-eu-west-1.amazonaws.com/qafoo-
profiler/packages/EEB5E8F4.gpg' | sudo apt-key add - && \
 5.     apt-get update && \
 6.     DEBIAN_FRONTEND=noninteractive apt-get -yq install
tideways-daemon && \
 7.     apt-get autoremove --assume-yes && \
 8.     apt-get clean && \
 9.     rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
10.
11. ADD run.sh ./run.sh
12.
13. RUN ["chmod", "+x", "./run.sh"]
14.
15. CMD ["./run.sh"]
```

8 *the official setup documentation:* https://tideways.io/profiler/docs/setup/installation

9 *the troubleshooting guide:* https://tideways.io/profiler/docs/setup/troubleshooting

Listing 6

```
1. #!/bin/bash
2.
3. tideways-daemon --hostname=tideways-daemon --address=0.0.0.0:9135 --udp=0.0.0.0:8135 >> /dev/stdout 2>/dev/stderr
4.
5. /sbin/my_init
```

Boot the Container

Now that we've created the Docker container configuration, it's time to boot it up. From the terminal, in the root of your project, run the command: `docker-compose up -d --build`. This will read `docker-compose.yml` and determine how to build the container setup.

If this is your first time booting a Docker container, it may take a few minutes to complete, depending on your network speed. If not, you should be ready to go in under a few minutes. Either way, when the container is ready, you will be able to open your web browser to `http://localhost:8080` and see your running application.

Profile an Application

It's time to learn about the information Tideways collects and collates[10], as well as get an idea of how the data is presented in the dashboard.

As we've only just integrated Tideways with our application, naturally, there's not going to be any information available.

We could repeatedly click through the application. But, that would take an inordinate amount of time to register a meaningful amount of data. Given that, we're going to speed things up by using ApacheBench[11].

If you're not familiar with it, ApacheBench is a command line utility for measuring the performance of web-based applications. There are other tools too, but `ab` is easy to use to get a baseline comparison. It's able to simulate traffic to an endpoint in your application. As my application only has one route, the default route, that's what I'll simulate traffic for.

To do so, I'll run the command:

```
ab -n 1000 -c 20 http://localhost:8080/
```

What this does is simulate a situation where twenty concurrent users send a total of 1,000 requests. Depending on the capacity of your development machine, that might take a while and put a bit of load on it. But after a few minutes, it'll be finished, and we can view some more meaningful data in the account dashboard.

Assuming that you've done that already, after reloading the dashboard, you'll see it's now better populated, as in Figure 3. The dashboard is composed of three sections:

- **Response Times**: which reveal application speed
- **Callgraph Traces**: transactions made against the application
- **Transactions:** transactions grouped by context

Collectively, these three sections aggregate:

- Fast and slow types of requests
- Frequently and rarely visited types of requests
- Error rates, total requests, and memory usage

If you mouse over the Response Times graph, you'll see details about the requests made to the application. This includes:

- The time of the request
- The number of requests
- The error rate
- The number of traces

You can see an example of this in Figure 4. If you click on **default** under Transactions, you'll see the Response Times graph again, and under it a Response Time Distribution graph.

By default, all the information is grouped into one transaction, called **default**. This is analogous to the global namespace in PHP if you will.

Get Context-Specific Information

Let's start getting more specific and configuring the information which we collect to be more context-sensitive. To do that, we're going to need to add some code to our application. As I'm working with Zend Expressive, I'm going to add it to

10 *Tideways collects and collates:* http://phpa.me/tideways-data

11 *ApacheBench:* http://phpa.me/apacheBench

Figure 3.

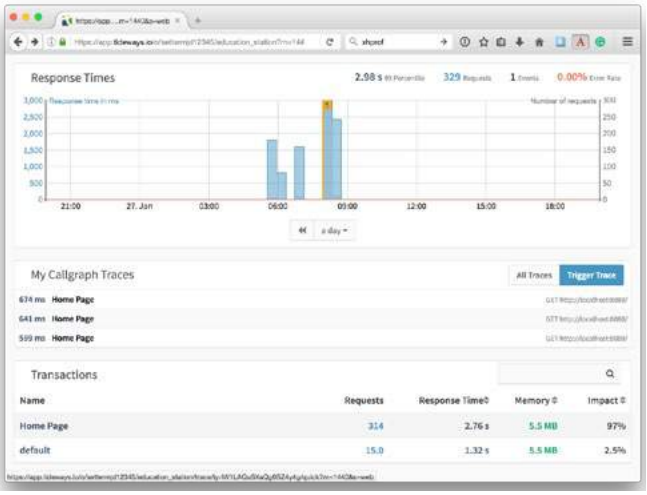
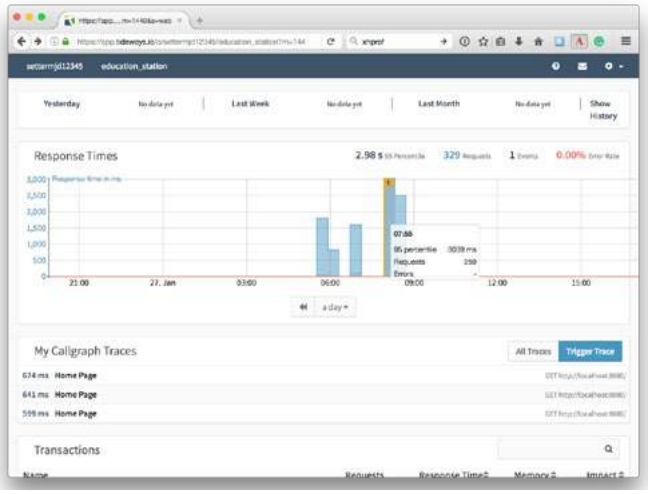


Figure 4.



the `__invoke` method in one of the `PageAction` classes. You can see the code below.

```
if (class_exists('Tideways\Profiler')) {
    \Tideways\Profiler::setTransactionName(__CLASS__);
}
```

This registers a transaction, with the name of the current class, if `Tideways\Profiler` exists. Naturally, this will only exist if the extension has been installed and enabled. In this case, I've only set the name of the current class, as there's only one function in that class. However, if you have a more sophisticated application than mine, feel free to use something more specific.

If we now either make a few more requests or run ApacheBench again, then we'll have collected data which makes use of that new transaction. You'll be able to see it under both **My Callgraph Traces** and Transactions. As we only have a transaction created for one part of the application, this example likely isn't that meaningful.

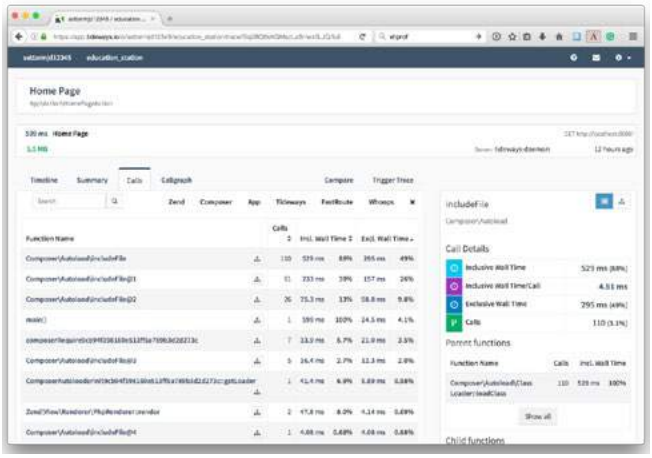
But, if you have a much larger application or, using an MVC-centric assumption, have a controller with a range of actions, you can now begin to collect information in very accurate ways. Instead of lumping all of the collected data under one banner, default, you can now collect it per action, per controller, and so on.

Once instrumented, you can now begin looking through the collected data based on the transaction. Now let's see how we can make the way the information is collected that much more meaningful.

Make the Dashboard Data More Human-Readable

How about, instead of using the name of a class, or a combination of class and function, or controller and action, we give it a more human-readable name? Tideways supports that, and I think it's an excellent feature. Let's label our previous class name as instead **Home Page**.

Figure 5.



To do so, again under Application Settings, click the button **Configure Transactions** in the Transactions section. There you'll see all the registered transactions. On the far right-hand side of the transaction's row, click the pencil link. A modal dialog will appear, and you can set a human-readable name to replace the one generated in code. After clicking **Save**, if you click the name of your application, near the top-left of the page, you'll see the transaction is now represented with the new name in the dashboard.

Now let's look at another component of the Tideways interface, a trace. A trace stores comprehensive information about a given request to your application. It has a timeline, which includes information such as the time, memory consumption, duration and compile overhead. It stores the class calls made to fulfill the request, which you can see in Figure 5. It stores a call-graph so that you can see the calls visually, quite similar to Xdebug. And it allows you to compare one trace to another.

Create Alerts

I'd like to finish up with one last aspect: creating alerts. Depending on the nature, size, depth, and complexity of our application, we may need to know how it's performing at different times, and in different ways.

To satisfy this, Tideways allows for creating alerts. Alerts can be set to trigger based on a condition, such as response time or error rate. The trigger condition can be set, based on a range of applicable values. And if it triggers, a given response can be taken. This can include sending an email, sending a message to an IRC channel, HipChat or Slack group, as well as to OpsGenie and Flowdock.

Let's step through sending an email, based on an error rate. The first thing we would need to do is create an action. From the main dashboard[12], we could click the cog icon, and then click **Organizations** in the drop-down menu. Then, under our organization name, we'd click **Alert Configuration** to

12 *the main dashboard:* https://app.tideways.io/dashboard

see a list of existing alerts, and to be able to create a new one.

You'll see one alert is already created, called **Home Page Action Notification**. This is an email, which is sent to the default email address. You can see a test example in Figure 6.

As that's already created, we need only create a trigger for it. So, go back to the dashboard for your application, click the cog, and then click **Application Settings**. From there, click **Configure Alerts**. Now, click on **Create Alert Condition**. In the popup window which appears, click **Error/Exception Rate**. Set *has error rate greater than* to 20, *will go back to normal if error rate gets down to* to 0, and leave *for at least* to 30 minutes. Then, click **Create**.

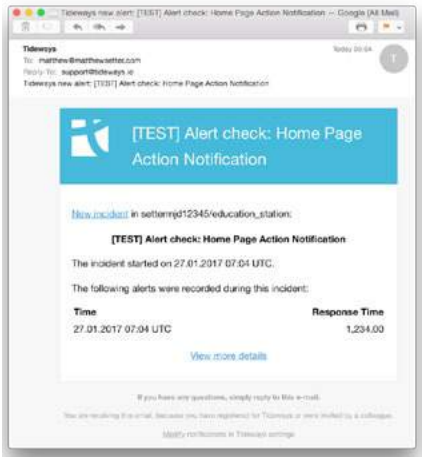
In Conclusion

And that, unfortunately, brings us to the end of the column. There's far, far more to Tideways than I've had the opportunity to cover in this month's column. You can instrument CLI scripts, integrate with PHP's major frameworks, configure user and team access control, group transactions together, influence the amount of tracing data collected, and much more. I hope I've given you a good taste as to what's on offer and that you'll try it out.

In the short time, I've had available to use it, I've genuinely come to appreciate it for the excellent tool it is. After from a few initial issues getting it setup, I've quickly come to appreciate the depth of the information which it can provide straight out of the box.

Yes, you *do* need to do a bit of configuration to get yourself up and running. But that's no different to any other application or service. Tideways is an excellent service. If you've not considered adding instrumentation to your applications, I strongly encourage you to do so now—with Tideways. Like so many things, it will take some getting use to. But when you appreciate the benefits, I'm confident you'll be hooked and never look back.

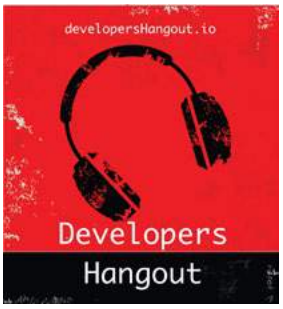
Figure 6.



Matthew Setter is an independent software developer, specializing in creating test-driven applications, and a technical writer http://www.matthewsetter.com/services/. He's also editor of Master Zend Framework, which is dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com/welcome-from-phparch.

Welcome to php[architect]'s new MarketPlace! MarketPlace ads are an affordable way to reach PHP programmers and influencers. Spread the word about a project you're working on, a position that's opening up at your company, or a product which helps developers get stuff done—let us help you get the word out! Get your ad in front of dedicated developers for as low as $33 USD a month.

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



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

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

Kara Ferguson
Editorial

Refactoring your words, while you refactor your code.

@KaraFerguson
karaferguson.net

technology : running : programming
rungeekradio.com

The Frederick Web Technology Group
meetup.com/FredWebTech

Learn to Say No

Cal Evans

Recently, I was approached by a friend who wanted me to deliver a keynote address at their upcoming conference. I had to tell them no.

I am going to pause here and let that sink in. Many of you know me. You know how much I love the sound of my own voice. You know how much I love doling out wisdom while sitting in my Little Rascal, even if I don't always follow that advice myself. Many of you know first hand exactly how big my ego is. So the notion of me saying no to a speaking gig, especially a keynote slot, will surprise you as much as it did me when I said it. Even though I was replying to the person via email, in my mind, I heard myself telling the person "Thank you for the invite, but I am going to have to say no."

Why did I say no? I've accepted other talks already this year. By the time this is in print, I will have already delivered "Life Badges" at SunshinePHP 2017. I've been accepted to speak at WordCamp Miami, and while I cannot name the conference, I may or may not be speaking in New York City this summer. So it's not like I'm turning down everything. Why this one? Why for only the third time in my speaking career did I turn someone down?

Simple. In 2016 I spoke at ten events. In 2015, that number was about the same. In 2014, I attended or spoke at 25 events. Over the past few years, I have learned the lesson that those who came before me tried to teach me and failed. That lesson? It's not as fun as it looks.

Oh sure, the conferences are awesome. I love every one of them. I make new friends and reconnect with old friends; it's a gas. But invariably, when asked, "How was it?" upon my return, I have to think back and realize that for three to five days, I saw the inside of an airplane, taxi, and hotel. As I get older (Shut up Joe Ferguson, you'll be old one day too!) it starts to wear on a body and a mind.

Right now, I've been off the road

since ZendCon 2016, and I am feeling refreshed and reinvigorated. I've started a new book, I've built a new API, and I am experimenting with new technologies like an Amazon Echo Dot. (Shut up, Phil Jackson. Just because I have one doesn't mean the NSA isn't listening)

I am also riding my bike four days a week. By every metric physical or mental, I feel better than I have in a long time.

I love seeing my friends at conferences. I will miss not seeing people every month. After ten years of doing this, I thought I would go stir crazy. Still, here I am happy and content. All because I have learned my career won't come to a crashing halt if I tell people no.

To my friend I had to say no to, I sincerely apologize. I know you've probably already filled the slot because there are a lot of great speakers in our community. Still, I absolutely hate letting people down, and a little part of me feels I have let you down. For that, I am sorry.

To everyone else, forget pleasing everyone. Ignore FOMO (Fear Of Missing Out); it is the enemy. For your sake, whether it's projects at work or side hustles, learn to say no.

Learn to walk away.

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.

Past Events

January

PHPBenelux Conference 2017
January 27–28, Antwerp, Belgium
https://conference.phpbenelux.eu/2017/

Upcoming Events

February

SunshinePHP 2017
February 2–4, Miami, Florida
http://sunshinephp.com

PHP UK Conference 2017
February 16–17, London, U.K.
http://phpconference.co.uk

March

ConFoo Montreal 2017
March 8–10, Montreal, Canada
https://confoo.ca/en/yul2017/

WordCamp London
March 17–19, London, U.K.
https://2017.london.wordcamp.org

Midwest PHP 2017
Bloomington, Minnesota,
https://2017.midwestphp.org

SymfonyLive Paris 2017
March 31–31, Paris, France
http://paris2017.live.symfony.com

April

PHP Yorkshire
April 8, York, U.K.
https://www.phpyorkshire.co.uk

Lone Star PHP 2017
April 20–22, Addison, TX
http://lonestarphp.com

DrupalCon Baltimore
April 24–28, Baltimore, MD
https://events.drupal.org/baltimore2017

May

phpDay 2017
May 12–13, Verona, Italy
http://2017.phpday.it

PHPKonf Istanbul
May 20, Istanbul, Turkey
http://phpkonf.org

PHP Tour 2017 Nantes
May 18–19, Nantes, France
http://event.afup.org

php[tek] 2017
May 24–26, Atlanta, Georgia
https://tek.phparch.com

PHPSerbia Conference 2017
May 27–28, Belgrade, Serbia
http://conf2017.phpsrbija.rs

International PHP Conference 2017
May 29–June 2, Berlin, Germany
https://phpconference.com

June

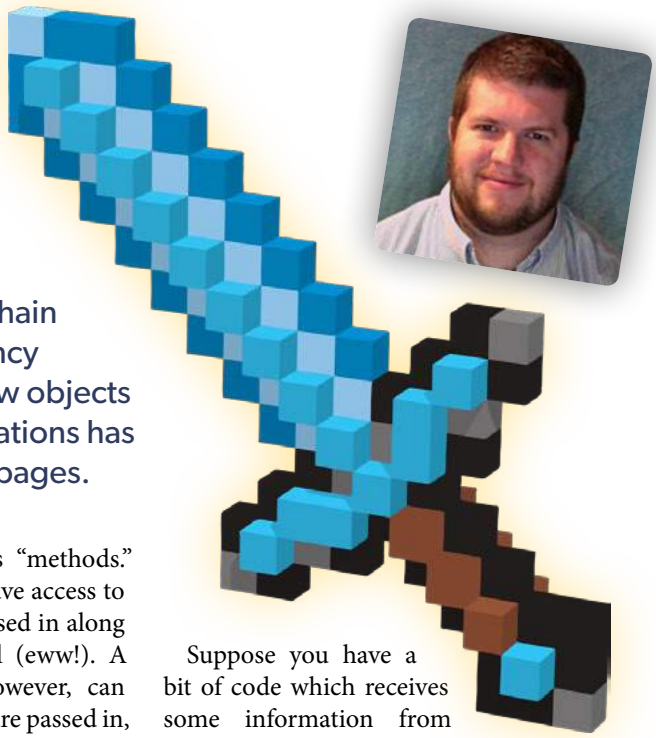
CakeFest 2017
June 9–10, New York, USA
https://cakefest.org

PHP South Coast 2017
June 9–10, Portsmouth, UK
https://2017.phpsouthcoast.co.uk

Dutch PHP Conference
June 29–July 1, Amsterdam, The Netherlands
https://www.phpconference.nl

Building Better Objects

David Stockton



Last month, we started talking about objects. Joseph Maxwell's article discussed several common design patterns for objects, including factory, strategy, and chain of command. Additionally, he talked about dependency inversion. This month we'll continue talking about how objects should be built and why using objects to build applications has advantages over the old style of building procedural pages.

Why Objects?

Classes and objects help us organize code. In old-style PHP, the code is typically organized in folders and subdirectories. Sometimes, when the same bits of code are found over and over, these may be extracted out into a function. When these functions are found to be repeated in file after file, inevitably, someone decides these functions should all be available in the same place; then `functions.inc.php`, `funcs.php`, or some similarly named collection of files is created. It's then `required` all over.

The problem is there's usually not a lot in common between these functions. Some may be used on nearly every page, while others are used in only a handful. And yet, without any sort of discrimination or limitations, all these functions need to be loaded to use any of them.

Objects are a combination of data and behavior. In an object, we call the "data" fields properties or attributes. We tend to call the behaviors "methods." Typically, a function will have access to the parameters that are passed in along with the occasional global (eww!). A method on an object, however, can work with parameters that are passed in, globals (eww!), as well as any properties on the object itself.

In fact, most methods will utilize properties on the object more than anything else. I think I'd go so far as to say if there's a public method in a class that doesn't use `$this` then perhaps the object is more of a function bucket than a well-designed object. A common counter-example I've seen, however, are factory classes. The ones I tend to see look like this:

But Really, Why Objects?

I've not really talked about why we should use objects—what the benefit is. Since objects combine behavior and data, it allows certain affordances we don't have when we're dealing with PHP primitives.

Listing 1

```
1. class SomeSortOfThingFactory implements FactoryInterface
2. {
3.     public function createService(
4.         ContainerInterface $container
5.     ) {
6.         $dep1 = $container->get(Dependency1::class);
7.         $dep2 = $container->get(Dependency2::class);
8.
9.         return new SomeSortOfThing($dep1, $dep2);
10.    }
11. }
```

Suppose you have a bit of code which receives some information from the query string or posted form fields and does things with them. Perhaps it looks up some information in a database, or it performs some calculations on them. With normal use of `$_GET` or `$_POST` to retrieve these values, we're working with an array. If you're trying to avoid PHP spewing notices, you have to first check for the existence of a key in those arrays in order to use the value that may or may not be there.

```
$filter = isset($_GET['filter']) ?
$_GET['filter'] : 'default';
```

With PHP 7, we now have the null coalescing operator which can simplify the statement above:

```
$filter = $_GET['filter'] ?? 'default';
```

If we don't do any of these and just try to use `$_GET['filter']` without checking, we'll get a notice from PHP:

```
$filter = $_GET['filter'];

Notice: Undefined index: filter in /path/
to/file.php on line 42
```

With object-oriented design, we can build objects which don't have to check that a field exists in an object, or that it's an instance of a certain type. We can be assured by the time we're working with the object, everything about it will be as

expected and as we wanted.

This means instead of dealing with `$_GET` or `$_POST`, we could work with a request object:

```
$filter = $request->get('filter',
'default');
```

I haven't shown you any code to make the request work, but it's not too hard to build.

Objects are About Messages

The power of objects comes from being able to compose them together and allowing them to communicate to allow for more complex and powerful behavior. Methods, specifically public methods, are the definition of the messages you can send to an object. If you think about it, this makes sense. While the methods define the functionality and behavior, the caller of the method is the one telling the object to do something.

The messages are the methods along with any parameters we provide. Let's take a look at what I mean.

```
class Greeter
{
    public function sayHello() {
        return 'Hello, World!';
    }
}
```

In this case, our Greeter class has a single public method, and thus, a single message that can be sent to it.

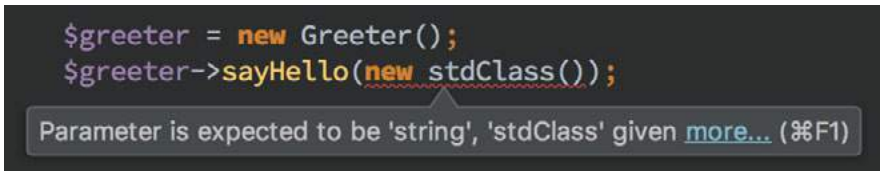
```
$greeter = new Greeter();
$greeting = $greeter->sayHello();
```

Suppose we update the class so `sayHello` takes a parameter:

```
public function sayHello($who = 'World')
{
    return "Hello, $who!";
}
```

With this update, the previous example still works, but the `sayHello` method can now say hello to many more. In fact, any parameter for `$who` that is, or can be rendered as a string will work. This means we could pass in names, numbers, or even other objects that define a `__toString()` method.

Figure 1. IDE Error



What happens if we passed in an array, a database connection, or an object that didn't have `__toString()`? We'd end up with an error.

```
$greeting = $greeter->sayHello(
    new stdClass()
);
```

For this we get the following error:

```
Catchable fatal error: Object of class
stdClass could not be
converted to string
```

As of PHP 7, it's now possible to type hint almost everything. With PHP 7.1, this includes type hints on the return types of functions and methods. If you can, and it does take some practice and discipline, I recommend using these any place you can. Type hints eliminate a lot of boilerplate checks on your parameters and ensure your code is smaller, simpler to understand, easier to test, and more robust. It also will allow your IDE to present issues and look for bits of code likely to be bugs before you even run your code.

```
public function sayHello(string $who =
'World'): string {
    return "Hello, $who!";
}
```

Now, if we were to try to call `sayHello` with the object as before, we get a slightly different error:

```
Fatal error: Uncaught TypeError: Argument
1 passed to Greeter::sayHello() must be
of the type string, object given
```

This will be followed by a stack trace indicating how we got to a place where we made a method call using an object when the code expected a string.

In PhpStorm, without even running the code (but with strict types turned on), we'd see something like this:

So, anyway, back to messages. In the examples above, we have an external user sending a message to an object. Messages can also be sent from:

1. one method to another method,
2. from a method to itself (recursion),
3. from an object to its parent or child,
4. to objects passed in as parameters,
5. or to an object's dependencies.

It is these relationships where the real power comes in.

Delegation

In the last article, and probably in a few before it, I said, "If you have to use 'and' to describe what your class does, it's doing too much." This is essentially a restatement of what's commonly known as the Single Responsibility Principle[1]. It states every class or module should have a responsibility for a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class. It was introduced by Robert C. Martin in *Principles of Object Oriented Design*[2]. Software applications are quite complex. There are many "moving parts" and aspects which need to be handled, checked, and controlled. So how, then, can we ensure a class has only one responsibility? This is where delegation comes in.

It is often said there are two hard problems in computer science—naming things and cache invalidation. The joke continues by including "off by one errors." Part of the problem with

1 *Single Responsibility Principle:* https://phpa.me/wikipedia-SRP

2 *Principles of Object Oriented Design:* http://phpa.me/unclebob-ood

determining what the purpose or responsibility of a class falls into the hard problem of "naming things." Coming up with descriptive and accurate names for classes is hard. Determining where the line delineating where the responsibility of one class ends and the next begins is hard. Sometimes, it may be better to build a class that does too much and then refactor out different responsibilities to different classes later.

This leads back to delegation. This means as much as we can, our classes should delegate work to another class.

Imagine you have a controller class with the responsibility of showing a user their preferences. Let's look at what needs to happen to do this.

- Determine if user is authenticated.
- Retrieve ID of user to look up.
- Connect to the database.
- Issue the appropriate query to look up the preferences record.
- Determine if a record was found.
- If a record was found, pull out the relevant fields for display.
- Display the record.

There's a lot going on and if we do all of this in the controller, there's no way to describe the operation without using "and." We need to delegate.

If we can allow another class (or set of classes) to determine if the user is authenticated or not, then that's one job off our (controller's) plate. However, we need the result before we can move forward, so whatever determines authentication will need to pass in the user object to our controller. I'll forego breaking down the authentication at this point as I'm trying to concentrate on the controller and its directly related objects.

Each object has a purpose. They come together and help solve a problem.

Assuming the authenticated user is made available to our controller we get the user ID from it. Instead of providing the entire user object, it could instead provide the user id if that's all we needed. Next up, we have to connect to the database. This sounds like a job for something else entirely. In fact, some other object that doesn't even necessarily know about the database at all could be used to retrieve a user.

We could have a sort of user service which itself encapsulates objects which encapsulate the database connection, the query, any sort of object hydration, etc. All we need in our controller at this point is that whatever does all this work ought to give back the user preference record. Once the controller has that record, it can delegate to some sort of view or view model object to render the actual HTML.

After this, most of the responsibility is delegated. The only things left are to ask a user service for the record and shuttle that data into a view or view model.

Now we have this:

1. Determine if user is authenticated (**delegated**)
2. Retrieve ID of user to look up. (**potentially delegated, probably provided**)
3. Connect to the database (**delegated**)
4. Issue the appropriate query to look up the preferences record. (**delegated, controlled by controller**)
5. Determine if a record was found (**possibly delegated**)
6. If a record was found, pull out the relevant fields for display.
7. Display the record. (**delegated**)

There are only one of those tasks left which has not been delegated. It's certainly possible there could be a class that has the job of converting from a user object into a view model object. In that case, we could delegate that responsibility as well.

This means delegation, our final action method could look like the following:

```
public function fetchById($id) {
    $preferences = $this->userPreferences->fetchById($id);
    return new ViewModel(['preferences' => $preferences]);
}
```

There's not a lot to it. If user authentication is handled elsewhere, then we can assume we won't get into this method if we're not authenticated. If the ID is provided (via the parameter), then we don't need to worry about that responsibility either. Presumably, the SQL connection and query are handled by the `$this->userPreferences` object, or delegated on to other objects from it. The `userPreferences` dependency would be injected into the constructor when creating the controller.

The `userPreferences` service would be responsible for making a determination about what to do if there was no record found. It could throw an exception, or potentially return an object following the null object pattern. The controller retrieves this object, passing it through to a `ViewModel` which is handled outside the `fetchById` action.

Each object has a purpose. They come together and help solve a problem. We've laid a foundation for code reuse by building an object with the sole responsibility of fetching user preferences. Fetching the preferences may be done via delegation to a group of other objects, but *how* it happens is not a concern of our controller. Also, if we needed to build a utility that could look up user preferences and display them as a command line utility, we can reuse the `userPreferences` object. That object doesn't know anything about how the data it returns will be used; it just knows how to retrieve a record when it is given an ID.

Depending on how that object itself were structured, it may be possible to seamlessly switch it or its dependencies out with something else that could retrieve a record from a CSV file, or an API or anything else. The rest of the system

wouldn't need to change. Caching layers could be introduced with minimal changes.

Encapsulation

Encapsulation is the ability for us to treat some aspects of what an object does as a black box. It is information and functionality hiding. It's counter-intuitive, but knowing less about what the code is doing is empowering. What I mean by this is: the less one class has to know about how another class works, the better. As a developer using classes you didn't build, the less you have to know to use them, the better. You only need to know what arguments to pass to a class' methods and what they will return.

In the controller example earlier, the only thing our controller needs to know about the user preferences object is that it has a method (or accepts a message) asking for user preferences given an integer identifier. It will return a user preferences object. Note in the description, nothing was said about a database. In fact, the controller doesn't need to know anything about a database to use that service.

```
interface UserPreference
{
    public function fetchById(int $id): UserPreferenceModel
}
```

Essentially, anything that implements the interface above could be swapped out with something else which also implements it, and our controller would not care. It could be a test double or mock object, or with a different internal implementation which retrieves user preferences from files, APIs, NoSQL databases or even a hardcoded list right in the code.

As another example of why encapsulation is important is in the use of everyday objects. If you drive, you're aware that roughly speaking, the gas pedal makes the vehicle go faster, the brake slows it down, and the steering wheel is used to aim the car in the desired direction. The car hides a lot of what it does and how it works from us, and we are better for it.

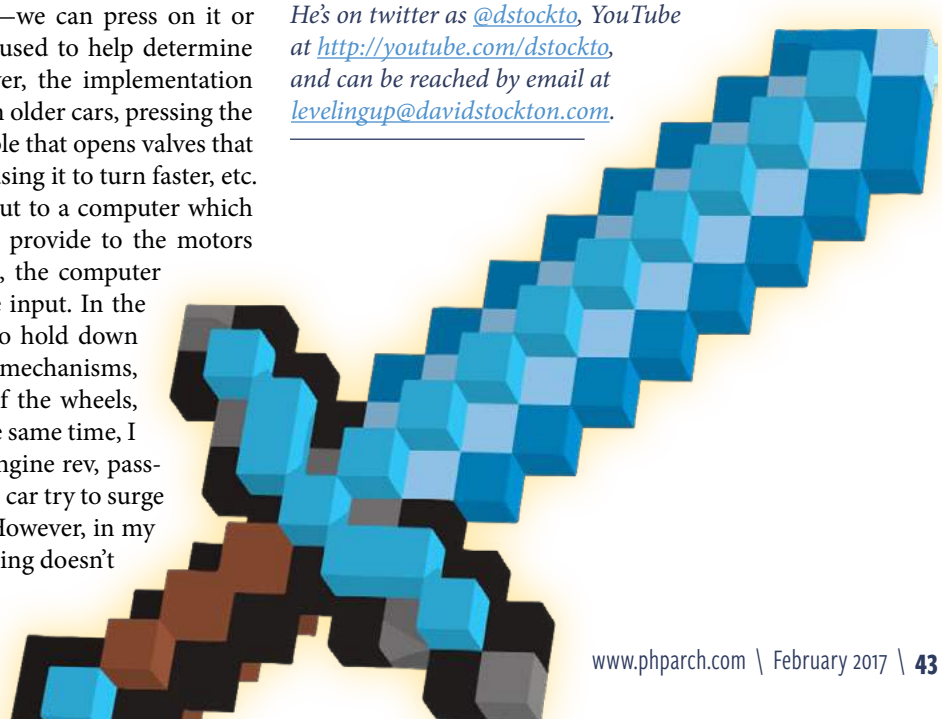
The gas pedal interface is simple—we can press on it or release pressure on it. This input is used to help determine how fast the car will move. However, the implementation differs from one vehicle to another. In older cars, pressing the pedal moves a lever which pulls a cable that opens valves that release more fuel into the engine, causing it to turn faster, etc. In some cars, the pedal provides input to a computer which determines how much electricity to provide to the motors turning the wheels. In certain cases, the computer may decide to completely ignore the input. In the car I drive daily, I have the ability to hold down the brake which through a series of mechanisms, holds physical pads against a part of the wheels, preventing them from turning. At the same time, I can press the gas which makes the engine rev, passing power to the wheels. I can feel the car try to surge and hear the engine pitch increase. However, in my wife's car, a hybrid, doing the same thing doesn't result in a surge or any increase in engine pitch. In most cases, the engine doesn't even turn on at all. The computer decides that if both inputs are pressed, the gas pedal will be ignored.

The steering wheel is another interface hiding a lot of complexity. In older vehicles, turning the wheel moved gears, racks and pinions, and other sorts of mechanical systems which eventually cause the wheels to turn in the desired direction. These systems got more complex as power steering was added, and now, with the latest models of Tesla vehicles, the wheels can turn without any input on the steering wheel at all. The computer, along with the various vision and sensor inputs can make the determination the car needs to move in one direction or the other. In the not too distant future, the interface for the car—one that has not changed much at all since its invention—may undergo a pretty radical change. Instead of the interface being gas, brakes, and steering, the interface may be voice input where the user of the vehicle simply speaks their destination, and the car determines everything that needs to happen to deliver the user to the destination safely and quickly.

Conclusion

By delegating work to other objects, we can help to ensure our objects only have one responsibility. By hiding data and functionality, we make it easier to be able to swap out certain pieces of our application without the need to rewrite large portions of it. The concepts of delegation and encapsulation are powerful in OOP and we'll be utilizing them later. In the coming months, we'll continue to explore objects, how they work, how they fit together, how they can communicate, and how you can use them to make better software. 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.



PHP 7—a Step Closer to a More Secure PHP

Chris Cornutt

As I write this another milestone in the PHP language has been reached. As of January 19th, 2017 the last version of PHP in the 5.x series (5.6) has finally rolled out of active support and is in "security only" support. If you're not familiar with the timeline of the language's release structure, let me give you a brief overview. Once a version goes stable—a major version—it then goes into active support. During this time smaller things are fixed, and features that were added are tweaked. Bugs are inevitably found and corrected, and minor versions are released to deploy those changes to the rest of the world. This goes on for a while (usually around a year) then it hits te stage PHP 5.6 is in now.

When it rolls into the "security fix only" state only the things that come out of reported or discovered security issues are updated, and minor versions keep on rolling out until it finally hits the end of life. PHP 5.6 hasn't quite hit the end of life yet, but with the move to "security only" it's getting closer all the time.

Why am I talking so much about PHP 5.6 in an article clearly titled to cover PHP's next major version? I wanted to give you a little perspective on the state of things right now and encourage you that, if you haven't already, the upgrade to PHP 7 is practically a must now. There are all sorts of goodies that come with PHP 7's initial release including:

- a massive performance boost
- improvements to type handling and enforcement
- new error types (and catchable fatal errors)
- new operators

…just to name a few. I'm not here to talk about all of the new handy things in PHP 7, though. As always, we're talking security here, so I wanted to take a little time to introduce you to some of the security related improvements which have come along with PHP 7. Security fixes are rolling all the time in the latest versions of PHP 7 (especially with 7.1 out now) but there are some major features I'd like to touch on so you'll know what to watch out for and what you can integrate when you're running on a PHP 7-powered platform.

Type Declarations

I know what you're saying—"how are type declarations security related?" Well, the answer is rooted in basic security principles including that simple systems are better than complex. If you've worked with a security system of any largish size you know unless good care is taken, things can get out of control quickly. Pieces are added to the authentication flow here, one-off authorization methods are added over there. There may even be whole sections of your application using completely different controls than others.

In all of this mess, it may be tricky to figure out exactly what's going on in a particular place in the code. While having a good threat model can help with the basic outline of these systems, there are still going to be times you have to dig in and get your fingers into the code and debug for those odd, hard-to-find problems. Make use of this new PHP 7 feature, the type declarations and return type declarations, to help keep your code simpler and enforce values being passed around are exactly what you want.

Imagine a scenario like this: you have a login system that is technically the merging of two larger pieces of work. Each had a slightly different way of working with the user and so they each came up with their own "user" class. On one you can access a set of private values through a magic `__get` method and in the other, you're forced to call a "getter" method to grab the

property value. Now, imagine combining these two systems into one, taking pieces from one and somehow merging them into a complete whole. You'll start passing around "user" objects between methods but here's the rub: how do you know for sure which kind of user you're working with?

Sure, you could create yet another "user" type that tries to merge the interfaces of the other two but what if the logic one user instance needs is slightly different than the other? If you're not sure what kind of user you're working with you can't effectively know which methods you'd need to call to get the information you need. This calls for added complexity to be introduced into the system and makes the result even more of a maintenance nightmare down the road. Do yourself a favor and keep it simple. It may end up being more work, but you can write code like this and know 100 percent that you're working with the right information:

```
function login(\App\Model\User $user,
               string $password) : bool {
    return password_verify($password, $user->password);
}
```

It's not specifically a security feature, but it does help keep code clean, clear, and more concise. Simple code is clear code, and simple security is always better than complex.

Random Numbers & Bytes

One of the other major improvements that came along with PHP 7 is the addition of two new functions backed by entirely new code for generating random values. Previously, PHP had functions like `randRandom Numbers and Bytes`, `mt_rand`, or even `uniqid` to create "random" values in your code. Unfortunately, the code behind the scenes of this functionality was flawed. It tried its best to come up with its own method creating values which seemed random but were, in truth, only "mostly random." Because of the method they used to calculate the random values they spit out, the results could sometimes be predicted if the attacker was given enough prior information. The key to randomness is the uncertainty in the result, making it much harder and much "stronger" (in cryptography terms).

PHP 7 changed things up and added in two new functions to help you replace the previous random generation methods: `random_string` and `random_bytes`. What sets these two functions apart from the previous methods is the source it pulls the data from. While previous PHP-only methods tried to be random internally, these new methods use external sources. They make use of well-tested, standardized randomization sources that Unix and Windows users have been enjoying for a long time.

For those on Unix-based systems, there are two special kinds of "files" that come built into the operating system whose only job is to provide random data. When you access `/dev/random` or `/dev/urandom` input is pulled from a wide range of sources (including "environmental noise" from device drivers) and condensed down to spit consumable data back out to the waiting user. The origins of these tools come from way back when Unix was first getting started. Developers realized they needed a good source of randomness based off of the data from an entropy pool, populated from the "noise" that's been gathered. The result is a highly random (though not 100 percent random) value. This value is shared back to the user as bytes of data they can do with as they please. This kind of randomization is ideal for use in cryptographic settings and in what these new functions in PHP use it for—returning random bytes of data and using it as a seed to create random integers.

random_bytes

Let's start with the easiest one to understand: `random_bytes`. True to its name is just returns the contents of the random data requested from whatever randomization source is relevant to the OS.

> *For Windows users, the randomization source is always* `CryptGenRandom()`. *For Unix-based systems, there are a few different options it could pull from. They are, in order of how they would be tried:* `getrandom(2)`, `syscall`, *or* `/dev/urandom`. *If no valid source can be found an* `exception` *is thrown.*

This data is then returned to PHP and, based on whatever `$length` value you've provided, will return that many bytes of information back to your script. This data can then be used to create things like keys for use in cryptographic operations. If you'd like to translate this value back into some sort of string (it's just bytes of data, remember), you could use something like `bin2hex` and reduce the result back down to the standard ASCII character set.

random_int

The second method that's included in this CSPRNG update is the `random_int` function. Much like its sibling, the `random_int` function pulls data from the best randomization source it can find. It then takes the data and massages it into an integer value based on the minimum and maximum values you provide. You can return an integer up to the `PHP_INT_MAX` value (currently somewhere in the 9 quintillion range) or as small as the `PHP_INT_MIN` value (negative 9 quintillion, or so). This is perfect for when you just need a number but don't want to have to rely on some of the pre-PHP 7 functionality to get it.

While there's not a "random string" function included in the set, it's pretty trivial to get a usable string back from the output of a `random_bytes` call with the help of `bin2hex`.

Unserializing Safely

For a while, there was a flaw in how PHP unserialized the serialized strings it was provided. In fact, there was even a major security issue defined for it (several actually). The main issue was in how the `unserialize` function handled the

unserializing of objects in the provided string. The key to the vulnerability was that behind the scenes PHP was recreating the object, methods and all, as a data structure and following along with the standard object creation rules. This includes some of the special functions which come along with PHP's object handling including the magic methods `__wakeup` and `__destruct`.

Why is this a problem? Well, remember when I said PHP takes the serialized string it's given and goes through the steps to re-create the object? This includes calling those magic methods too. Let's look at an example to see how this could be used for evil. Let's assume someone decided taking in a serialized string on the URL was a good idea (It's not; don't do it!) and then they could just magically use the object on the other side. First, we'll come up with our custom class:

```
class Foo
{
   public function __wakeup() {
      echo 'BOO!';
   }
}
```

This is a simple example, but it works to make our point. If you make an instance of this class and serialize it, you'll end up with a string like:

```
O:3:"Foo":0:{}
```

Looks pretty benign, right? Well, what you don't see is what could happen when that value is unserialized. When PHP pulls the string in and runs the process to unserialize, it says, "Oh look, I know what a `Foo` object is—let's go ahead and make one." When it does so, the class' `__wakeup` function is executed and the string "BOO!" is returned out to the waiting user. It's a bit contrived, but it gives you an idea as to how an attacker, if they knew enough about your application, could execute parts of your codebase or arbitrary PHP code without your script calling it directly. This is what's known as a Remote Code Execution (RCE) and can be a very dangerous vulnerability.

So, how can we fix this and what does PHP 7 have to offer that would magically make this problem go away? PHP 7 added a second parameter to the `serialize` function allowing you to control what classes can be instantiated from the call (or maybe even none at all). Here's an example which would block the process from executing our `Foo` example up above:

```
// Allowing only specific classes through
unserialize($inputString,
   ['allowed_classes' => 'MyTestClass']
)
```

Or, if you know there's no way your `unserialize` call would need to create objects, you can completely disable it:

```
unserialize($inputString,
   ['allowed_classes' => false]
);
```

This effectively stops the RCE from being possible and closes up one more hole and hardens your application that much more.

Looking Forward

Finally, I want to end this month with a look forward to what's coming up in future versions of the PHP 7.x series as it pertains to security needs. One of the most major changes coming down the line is the deprecation of the `mcrypt` functionality. This has been one of the standard, go-to cryptographic libraries in the PHP language for as long as I can remember. However, the project it is based on—the mcrypt libraries themselves are now unmaintained and haven't had a major update in a very long time. As such, PHP moving forward will be pushing OpenSSL's cryptographic functionality into its place. PHP 7.1.x has recently been pushed out the door and includes the mcrypt deprecation message and, eventually, will be removed entirely (possibly in 7.2 but later is a bit more likely).

There are also a few RFCs out that are attempting to solve the "secure unserialization" problem in a different way than just adding a second parameter. Other PHP RFCs out on http://wiki.php.net/rfc also trying to make it into future versions including increasing the security of sessions by default and more secure settings in the default `php.ini`. Some of these won't ever see their way into the PHP codebase, but it's good to see the ideas out there and being discussed.

> *PHP's community wouldn't be where it is without the developers who use it daily and contribute their ideas to the project.*

PHP's community wouldn't be where it is without the developers who use it daily and contribute their ideas to the project. Don't hesitate to start up a discussion if there's a feature—security or otherwise—you think could improve the security stance of the language.

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

January Happenings

PHP Releases

PHP 7.1.1: http://php.net/archive/2017.php#id2017-01-19-1
PHP 7.0.15: http://php.net/archive/2017.php#id2017-01-19-2
PHP 5.6.30: http://php.net/archive/2017.php#id2017-01-19-3

News

Laravel News: Deprecations for PHP 7.2

In this new post to the Laravel News site they list out some of the deprecations coming in PHP 7.2 based on this RFC. Most of the things in the list are functionality that you just don't see much in recent PHP development. It could cause problems for older codebases but for those already on PHp 7 it shouldn't take much to make the necessary changes. http://phpdeveloper.org/news/24863

ThePHP.cc: Refactoring to PHP 7

On thePHP.cc blog today there's a new post sharing some helpful hints related to refactoring your application to PHP 7 written up by a friend of the group, Tim Bezhashvyly. He makes the recommendation of a bold first step: enabling the strict typing on every file in your application to enforce the typing of all values. Next he recommends running your current test suite to see where the failures are. http://phpdeveloper.org/news/24862

Matthieu Napoli: Using Anonymous Classes to Write Simpler Tests

In a recent post to his site Matthieu Napoli shows you how to use the recently added anonymous classes functionality to help make your unit tests simpler. Anonymous classes allow for the on-demand creation of class instances without the need for the predefined class being required. http://phpdeveloper.org/news/24837

Exakat Blog: PHP Likes Sorting Too Much

The Exakat blog has a recent post talking about how PHP likes sorting too much, that is, even in some places you're not using specific sort functions, PHP does it for you anyway. He covers the functionality for each of those previously mentioned functions and what kind of sorting they're performing: `array_unique`, `glob`, and `scandir`. http://phpdeveloper.org/news/24818

Symfony Blog: Symfony 2016 Year in Review

The Symfony blog has posted their wrap up of the activities around the project and its ecosystem in 2016. They cover releases made, events/conferences and updates on both components and documentation changes. http://phpdeveloper.org/news/24802

QaFoo Blog: Getting Rid of static

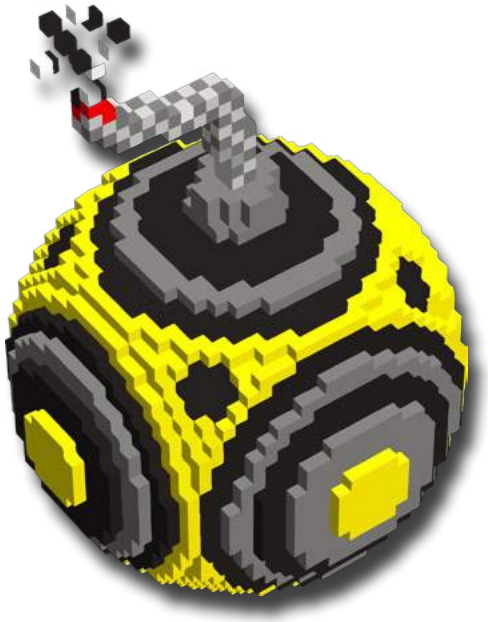
On the QaFoo blog today Kore Nordmann has posted a suggestion that could make your unit testing life simpler: get rid of statics (variables, methods, etc). They offer three things you can do to help refactor away from using static methods: Replaceable Singletons, Service Locator, Dependency Injection. http://phpdeveloper.org/news/24801

Derick Rethans: Good Bye PHP 5

On his site Derick Rethans has posted an announcement about a major change in the Xdebug project (a widely used PHP debugger) he leads, saying goodby to PHP 5. He shares some of the responses to the change (via Tweets) from the community ranging from full support to outcry over the change. He points out that the current version of Xdebug (2.5) will continue to operate on PHP 5 systems. http://phpdeveloper.org/news/24795

Aidan Woods: Secure Headers for PHP

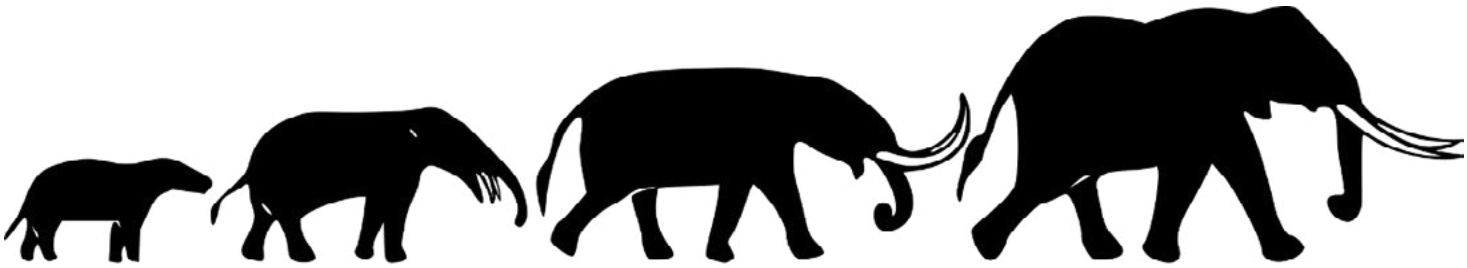
In a recent post to his site Aidan Woods shares information (and code) related to the use of secure headers in PHP applications. He's even created a package to help make it easier to drop them into a new or existing project without too much trouble. He starts by covering why he created the library and what it can help you with including making things like a CSP policy easier to maintain. http://phpdeveloper.org/news/24786



The Value of Moving Forward

Eli White

I often write about how it's important for us not to alienate the newer programmers. To not immediately put down the coding methods and styles that are older (and more well known). However, today I want to talk about the value that exists in at the same time, pushing things forward in technology.



Specifically, I want to refer to the recently passed proposal for PHP 7.2, to deprecate a number of functions that have existed, in some cases, for a very long time. These functions would begin to throw deprecated warnings in 7.2 and are aimed to be completely removed in 8.0 (whenever it is released).

For reference, you can find the full list of functions and a discussion on the topic by reading the RFC with the proposed deprecations[1]

It includes deprecating functions such as `__autoload`, `each()`, `create_function()`, and more. But I'm not going to go into an exhaustive list of these items because that's not the point of this discussion. It's not that things are being deprecated (nor perhaps that you may have feelings about that). But it's about the general situation: the need to move forward at times.

PHP is constantly evolving. We continually add new features to the language, we constantly strive to improve how PHP works, and expand what the language can do. But that does come with a cost. That cost is that the complexity level of PHP increases as well. With every new operator we add, with each new library, PHP gets a bit more convoluted.

> *We keep moving forward, opening new doors, and doing new things, because we're curious and curiosity keeps leading us down new paths.*
>
> *— Walt Disney*

It is important for us, therefore, to take times like this, to be willing to move forward. To be prepared to remove some old cruft that people are not using (much) or which promotes poor—or insecure—habits (`register globals` I'm looking at you). We should clean up the language as we go learning what works and what doesn't. While yes, it does mean that some old code might start throwing deprecation warnings, eventually leading to full on errors in PHP 8.0 appearing.

We are making sure that the language remains accessible to the beginner, especially if removing things which can be highly confusing to someone just starting out. For example, explaining what the difference is between `__autoload` and `spl_autoload` or code that uses `create_function()` versus code that creates lambda functions.

While at times it may be painful to lose those parts of our past, those keywords, and functions that some of us have used for over a decade. It is an important process of allowing the language to move forward, and continue to be the fantastic tool we all know it to be.

Eli White is a Conference Chair for php[architect] and Vice President of One for All Events, LLC. He probably need to deprecate some areas in his own brain as well. @EliW

1 *RFC with the proposed deprecations: https://wiki.php.net/rfc/deprecations_php_7_2*

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