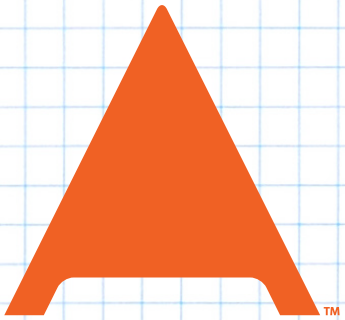
www.phparch.com

March 2017
VOLUME 16 - Issue 3



php[architect]

Back to the Drawing Board

ALSO INSIDE

Education Station: Validate the Complexity of Your Writing With TextStatistics

Community Corner: Three Tech Lessons I Learned Cleaning the Kitchen

Leveling Up: Exploring Object Immutability

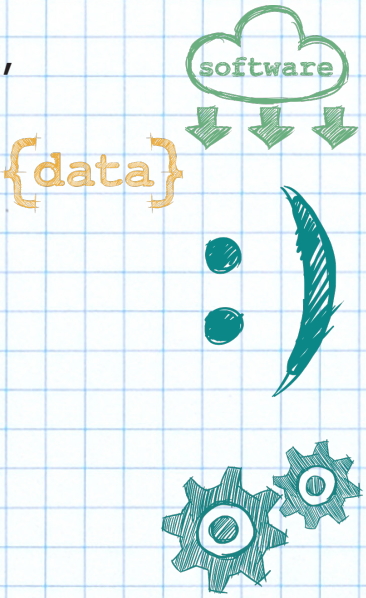
finally{}: Working Together

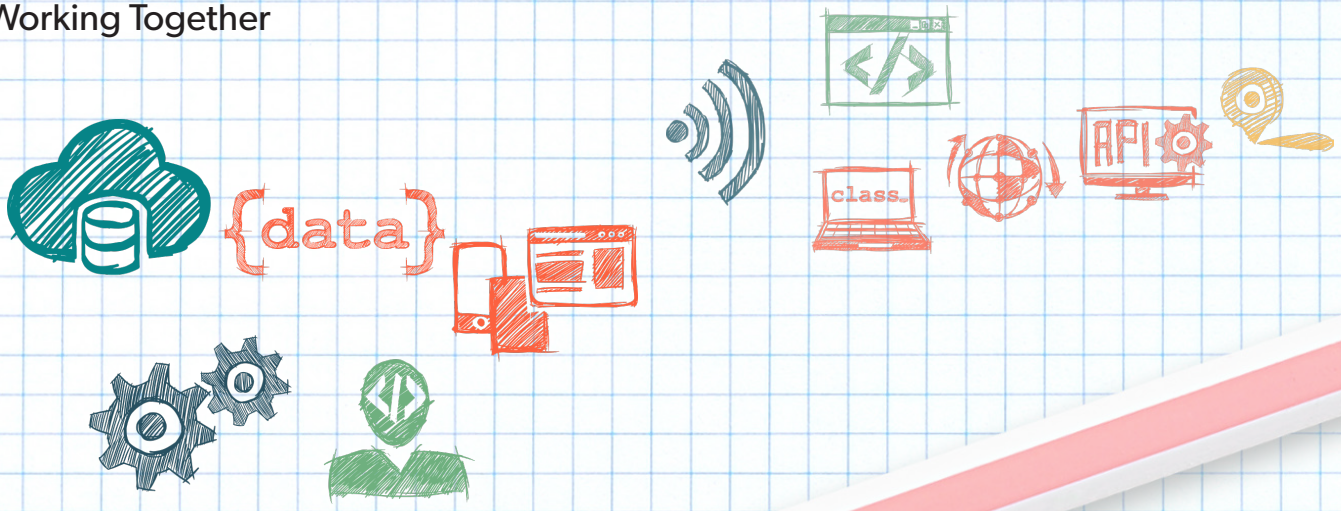
Learning to Code with Minecraft, Part Two

Deploying to Docker Swarm

How to Use SELinux (No, I Don't Mean Turn It Off)

Pursuing a Graduate Degree as Professional Development





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 · Zeamster · Open Sourcing Mental Illness · platform.sh · stickermule

tek.phparch.com

php[architect]

CONTENTS

MARCH 2017
Volume 16 - Issue 3

Back to the Drawing Board

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

Back to the Drawing Board

Last month we looked at some non-traditional uses of PHP in *Escape from the Server* and "serverless" infrastructures are all the rage these days. This month, the server is back ()with a vengeance?) with articles focused on server security, deployment with Docker, asynchronous PHP with Minecraft and more.

Early on in my career, I had the fortune to be pushed to learn some basic Linux administration so I could help set up new development sites for our projects. Before I knew it, I was in the on-call pager rotation and running Linux on my work desktop—this was before laptops were standard issue. The first thing I learned, is that being on-call is the worst. But in all seriousness, it really taught me to appreciate what goes into keeping Apache, MySQL, and the rest running and to consider the impact of the code I wrote on memory and CPU. These skills came in handy when I was part of the Digital Media team at D.C. United. We were a very small team in the basement of RFK and being able to manage our own servers was a huge benefit.

I also learned how to debug a "White Screen of Death" by finding and tracing through log files, a skill which seems very lacking on Stack Overflow and other forums. If you've been avoiding getting your hands into some DevOps tasks, don't put it off anymore!

First, Chris Pitt has *Learning to Code with Minecraft, Part Two*. In this installment, he updates the async PHP server, adds to the game, and dabbles with a preprocessor to automate the writing of some boilerplate code. In *Deploying to Docker Swarm*, Chris Tankersley looks at the evolution of deployment solutions for Docker containers. He'll show you how to use Docker Swarm to take your container into production. Next, Chuck Reeves explains *How to Use SELinux (No, I Don't Mean Turn It Off)*. SELinux provides a mandatory access control layer which gives you more fine-grained control over how services can access your server resources.

Jack Polifka shares his advice on *Pursuing a Graduate Degree as Professional Development*. If you're self-taught or considering going back for a degree, he'll explain the skills you'll learn and how they're applicable throughout your career. Conversely, you're never really done getting better as a software developer. Cal Evans shares *Three Tech Lessons I Learned Cleaning the Kitchen* in *Community Corner*.

This month's *Education Station* by Matthew Setter is on how to *Validate the Complexity of Your Writing With TextStatistics*. There's a surprising variety of algorithms to assess how readable an article or tutorial might be. If you want to improve as a writer, these metric offer valuable feedback. David Stockton continues exploring how to build better objects in *Leveling Up: Exploring Object Immutability*. Unexpected changes in application state can cause bugs which are hard to find. See how immutable objects give you more control over where these changes happen. Eli White concludes this issue with a timely call to action based on what he observed while traveling in *finally{}: Working Together*.

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.

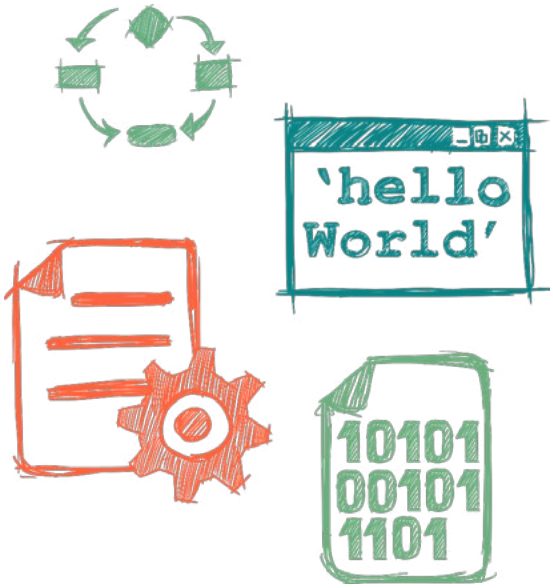
- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/March2017_code

Learning to Code with Minecraft, Part Two

Chris Pitt



Last time, we started to build a multiplayer code school. We began by creating an async PHP server, complete with WebSockets. We connected to an unmodified Minecraft server and allowed the Minecraft player to join teams with a code learner. In this article, we're going to upgrade to the latest version of Aerys. We're also going to build more of the game, so that people can learn to code!

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] `1.11.2` is the latest version at the time of writing. I was having some segmentation fault issues with PHP 7.1.2, but switching to 7.0 bypassed them.

Upgrading to Amp 2

When I started writing the last part, the latest version of Aerys (`^0.4.3`). Since then, there's been a lot of progress in the Asyncrhonous Interoperability Group[4], and follow-on effects for Amp and Aerys libraries. I'd like us to upgrade our codebase to use this new code, as shown in Listing 1.

This means we need to change some of our server and WebSocket code, beginning with how we check for new log lines (see Listing 2).

This is exactly the same code we had before, but it's wrapped in a generator function. Amp 2 introduces the `Coroutine` class as a way of converting generator functions to promise-like constructs so they can be consumed when they're put next to the `yield` keyword. In turn, we need to execute this code differently, refer to Listing 3.

Where before we were using `Amp\repeat`, we now have to use `Loop::repeat`. This functionality moved out into the `AsyncInterop\Loop` class.

By the time you read this, Amp 2 may already have been tagged. Try that version instead of `dev-amp_v2` *as it'll be more stable!*

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 *Asyncrhonous Interoperability Group: https://github.com/async-interop*

Listing 1. composer.json

```
{
   "require": {
      "amphp/loop": "dev-master",
      "amphp/aerys": "dev-amp_v2",
      "theory/builder": "^0.0.1"
   },
   "autoload": {
      "classmap": [
         "src"
      ]
   },
   "minimum-stability": "dev",
   "prefer-stable": true
}
```

Listing 2. from src/Socket.php

```
private function getNewLines() {
   $generator = function () {
      $time = yield Amp\File\mtime($this->path);

      if ($this->logTime !== $time) {
         $body = yield Amp\File\get($this->path);

         $allLines = explode(PHP_EOL, $body);

         $newLines = array_slice(
            $allLines, $this->logLines
         );

         $this->logLines = count($allLines);
         return array_filter($newLines);
      }
      return [];
   };

   return new Amp\Coroutine($generator());
}
```

These coroutines implement a promise interface, to provide a consistent and interoperable experience. Amp 2 has essentially moved the magic out of the `Amp\*` methods, and into dedicated structures. It's an approach I very much like, but it does introduce a lot of new boilerplate code.

Preprocessors

Faced with a lot of boilerplate code, we could dip our toes in the pool of compiler macros. Compiled languages have used these for ages, transforming patterned code into a preferred syntax or configuration.

Let's look at the structure of these async methods:

```
private function doAsyncThing() {
   $generator = function() {
      // code that yields values
   };

   return new Amp\Coroutine($generator());
}
```

That's what a class method looks like, but the pattern extends to anonymous functions as well (see Listing 4).

It seems we could shorten this recognizable pattern to something expressive and succinct as in Listing 5.

There are a few things we could add, which would also make working with bound variables easier. We could capture and repeat defined variables, so the generator function has access to `doAsyncThing` parameters. In the case of anonymous functions, we could also capture the surrounding scope so we wouldn't need a bunch of `use ($var1, ...)` code.

That's exactly what I set out to do, with `preprocess.io`. It's a collection of Composer installable preprocessor macros. I won't go into much more detail about how they work, but preprocess/pre-async[5] does exactly what I've just described.

> *I get that preprocessors aren't everyone's cup of tea. There is definitely some way to go before they are as easy to code-suggest or syntax highlight. But they work well at the moment, and I've greatly enjoyed trying them in my side-projects. Just last week, I refactored a semi-popular library to use class accessor preprocessor macros[6]. I was able to shave about 350 lines of code from the library, preserving all the functionality and introducing PHP 7 type hinting.*

5 *preprocess/pre-async:* https://github.com/preprocess/pre-async

6 *class accessor preprocessor macros:* http://github.com/asyncphp/doorman/pull/20

Listing 3. from src/Socket.php

```
1. public function onStart(Endpoint $endpoint) {
2.    $this->endpoint = $endpoint;
3.
4.    Loop::repeat(
5.       500, Amp\wrap(function () {
6.          $newLines = yield $this->getNewLines();
7.
8.          foreach ($newLines as $line) {
9.             preg_match(
10.               "/(\\S+) joined the game/", $line, $matches
11.            );
12.
13.            if (count($matches) === 2) {
14.               $this->initiatePlayer($matches[1]);
15.            }
16.         }
17.      })
18.   );
19. }
```

Listing 4

```
1. $mapper = function($path) {
2.    $generator = function() {
3.       yield asyncFileGetContents($path);
4.    };
5.
6.    return new Amp\Coroutine($generator());
7. };
8.
9. $paths = [...];
10.
11. return mapWithCoroutines($paths, $mapper);
```

Listing 5

```
1. // when you see...
2.
3. private async function doAsyncThing() {
4.    // ...
5. }
6.
7. $mapper = async function () {
8.    // ...
9. };
10.
11. // then replace it with...
12.
13. private function doAsyncThing() {
14.    // capture parameter values
15.
16.    $generator = function () {
17.       // expand parameter values
18.       // ...
19.    };
20.
21.    return new Amp\Coroutine($generator());
22. }
23.
24. // capture external values
25. $mapper = function () {
26.    // capture parameter values
27.
28.    $generator = function () {
29.       // expand external values
30.       // expand parameter values
31.       // ...
32.    };
33. };
```

Listing 6. from src/Socket.php

```
$newLines = yield $this->getNewLines();
foreach ($newLines as $line) {
    preg_match("/(\\S+) joined the game/", $line, $matches);

    if (count($matches) === 2) {
        yield $this->playerJoined($matches[1]);
    }

    preg_match("/(\\S+) left the game/", $line, $matches);

    if (count($matches) === 2) {
        yield $this->playerLeft($matches[1]);
    }
}
```

Listing 7. from src/Socket.php

```
private function playerJoined(string $player) {
    $generator = function () use ($player) {
        $this->players[$player] = $player;

        yield $this->broadcast([
            "type" => "player-joined",
            "data" => $player,
        ]);

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

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

    return new Amp\Coroutine($generator());
}

private function broadcast($payload) {
    $generator = function () use ($payload) {
        $payload = json_encode($payload);

        yield $this->endpoint->broadcast($payload);
    };

    return new Amp\Coroutine($generator());
}

private function playerLeft(string $player) {
    $generator = function () use ($player) {
        unset($this->players[$player]);

        yield $this->broadcast([
            "type" => "player-left",
            "data" => $player,
        ]);

        foreach ($this->games as $i => $game) {
            if ($game->hasPlayer($player)) {
                unset($this->games[$i]);
            }
        }
    };

    return new Amp\Coroutine($generator());
}
```

Cleaning up the Socket Code

Last time we started some basic WebSocket interaction. We queried for a list of connected Minecraft players and made a way to join them in a game. We still need that functionality, but we should clean it up, and extend it to show when new players enter and leave, as in Listing 6.

We begin by renaming the `initiatePlayer` method, to match a pattern of listening for players joining and leaving. Then, we need to broadcast these activities (see Listing 7).

These methods are mirror images of each other, one for moving the player to the waiting area and creating a game; the other for removing the game once the player leaves the server. I thought it would make things slightly clearer if I abstracted the broadcasting behavior, which includes its own JSON encoding.

Next up, we need to refactor the `onData` method in Listing 8.

I've also abstracted the send behavior, as it's something we repeatedly do. The website can request the initial list of players and coders can join players in new games. Each of these sends messages back through the WebSocket, but the latter

Listing 8. from src/Socket.php

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

    if ($parsed["type"] === "get-players") {
        yield $this->send(
            $client, [
            "type" => "get-players",
            "data" => array_values($this->players),
        ]);
    }

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

        yield $this->send(
            $client, ["type" => "joined"]
        );

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

        array_push($this->games, new Game($player, $client));
    }
}

private function send($client, $payload) {
    $generator = function () use ($payload, $client) {
        $payload = json_encode($payload);

        yield $this->endpoint->send($payload, $client);
    };

    return new Amp\Coroutine($generator());
}
```

Listing 9. from public/js/app.js

```
class App extends React.Component {
    render() {
        return React.createElement(
            "div", {"className": "message"}, "hello world"
        )
    }
}

ReactDOM.render(
    React.createElement(App),
    document.querySelector(".app")
)
```

Listing 10. from public/js/app.js

```
class Game extends React.Component {
    render() {
        return React.createElement(
            "div",
            {"className": "game"},
            "game"
        )
    }
}

class PlayerList extends React.Component {
    render() {
        return React.createElement(
            "div",
            {"className": "player-list"},
            "player-list"
        )
    }
}

class App extends React.Component {
    constructor(...params) {
        super(...params)

        this.state = {
            "player": null,
        }
     }

    render() {
        let child = null

        if (this.state.player !== null) {
            child = React.createElement(Game)
        } else {
            child = React.createElement(PlayerList)
        }

        return React.createElement(
            "div", {"className": "message"}, child
        )
    }
}
```

also notifies the player.

Reacting to Changes

This leads to an interesting place. Previously, we were requesting a list of players periodically, and rendering that list into HTML elements using jQuery. The trouble is, that's difficult to manage once we start receiving messages about single players joining or leaving.

This definitely seems like a case for using something like ReactJS[7]. For the sake of time, I won't go too much into detail about ReactJS, but it's something that's going to save us some time here.

To begin, we should add the react libraries to the document in `public/index.html`:

```
<body>
    <div class="app"></div>
    <script src="https://unpkg.com/react@15/
        dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15/
        dist/react-dom.js"></script>
    <script src="/js/app.js"></script>
</body>
```

We've replaced the previous "hello world" message and the ordered "players" list. Instead we're going to build those things using ReactJS components.

> *I'd like to avoid setting up a build-chain, as they're generally brittle and will lead to more questions than answers. Instead, we'll use the ES6 (non-JSX) version of ReactJS code.*

Commenting out all the previous JS code, we can begin the new ReactJS interface, see Listing 9.

Create components are either ES6 classes or ordinary functions. In class form, the `render` method is required to return another ReactJS component or array of them.

In function form (sometimes referred to as "stateless functions"), the function itself needs to return another ReactJS component or an array of them.

The `React.createElement` method's parameters should be:

1. The HTML element tag type or the name of a ReactJS component class.
2. An object of properties, similar to the attributes HTML elements can have.
3. A string, react component (created with `React.createElement`), or an array of these.

With this code, we've replaced our "hello world" message. Let's make the list of players now. You can imagine of the `render` method only being executed once: returning the DOM elements which should appear given *any* application state. So adding a list would look like Listing 10.

7 *ReactJS: https://facebook.github.io/react*

We can create a constructor function which, after calling the parent class's constructor with `super(...params)`, sets the `App` class's initial state. This state is used, in `render`, to work out whether a player has been selected. If so, the `Game` component class is rendered, else the `PlayerList` component class will be shown.

Next, let's create a new socket connection, and pass it to the `PlayerList` and `Game` component classes as in Listing 11.

It's the same WebSocket connection code we had before, but now the connection happens just before the `App` component class is added to the DOM. Next, in Listing 12, we need to attach listeners to the socket, from within the `PlayerList` component class.

Just before the `PlayerList` is added to the DOM, we attach an open event listener to the socket. This will be invoked as soon as the WebSocket can start receiving messages.

Then, we attach another event listener to be invoked when new messages are received. We also send out a "get-players" message, to get the initial list of connected Minecraft players.

Though not strictly required, it's a good idea to clean up things like event listeners in `componentWillUnmount`*. If you muddy up the environment in* `componentWillMount` *you should clean it up in* `componentWillUnmount`*.*

Listing 11. from public/js/app.js

```
class App extends React.Component {
   constructor(...params) {
      super(...params)
      this.state = {
         "player": null,
         "socket": null,
      }
   }
   componentWillMount()
   {
      this.state.socket = new WebSocket(
         "ws://127.0.0.1:8080/ws"
      )
   }
   render() {
      let child = null
      if (this.state.player !== null) {
         child = React.createElement(
            Game,
            {"socket": this.state.socket}
         )
      } else {
         child = React.createElement(
            PlayerList,
            {"socket": this.state.socket}
         )
      }
      return React.createElement(
         "div", {"className": "message"}, child
      )
   }
}
```

In `onMessage`, we set the state for the initial list as well as individual players entering and leaving the game. The `render` changes to allow us to map over the connected players and build individual ReactJS objects for each (see Listing 13).

The `Player` component class merely renders a link, containing the text player name. When it's clicked, we send the player's name to an `onJoin` property. So, we need to pass that down the tree as in Listing 14.

This takes care of the process for seeing a list of Minecraft players, and for joining a game with one of them. I decided it would be better to avoid any of the components other than `App` knowing about the WebSocket. Then, `Game` and `PlayerList` can act as function components for now. I also added a bit of code to reset the interface if the Minecraft player left. The final code looks like Listing 15.

Listing 12. from `public/js/app.js`

```
class PlayerList extends React.Component {
    constructor(...params) {
        super(...params)

         this.state = {
            "players": [],
         }

          this.onMessage = this.onMessage.bind(this)
    }

    componentWillMount() {
        this.props.socket.addEventListener(
            "open", () => {
                this.props.socket.addEventListener(
                    "message", this.onMessage
                )

                this.props.socket.send(JSON.stringify({
                    "type": "get-players",
                }))
            }
        )
    }

    componentWillUnmount() {
        this.props.socket.removeEventListener(
            "message", this.onMessage
        )
    }

    onMessage(e) {
        let parsed = JSON.parse(e.data)

        console.log(parsed)

        if (parsed.type === "get-players") {
            this.setState({
                "players": parsed.data
            })
        }

        if (parsed.type === "player-joined") {
            this.setState({
                "players": this.state.players
                    .filter(
                        player => player !== parsed.data
                    )
                    .concat([parsed.data])
            })
        }

        if (parsed.type === "player-left") {
            this.setState({
                "players": this.state.players
                    .filter(
                        player => player !== parsed.data
                    )
            })
        }
    }

    render() {
        return React.createElement(
            "div", {"className": "player-list"},
            this.state.players.map(player => {
                return React.createElement(
                    Player,
                    {"name": player, "key": player}
                )
            })
        )
    }
}
```

Listing 13. from `public/js/app.js`

```
class Player extends React.Component {
    constructor(...params) {
        super(...params)

        this.onClick = this.onClick.bind(this)
    }

    onClick(e) {
        e.preventDefault()

        this.props.onJoin(this.props.name)
    }

    render() {
        return React.createElement(
            "a",
            {
                "onClick": this.onClick,
                "className": "player",
                "href": "#",
            },
            this.props.name
        )
    }
}
```

Listing 14. from `public/js/app.js`

```
class PlayerList extends React.Component {
    render() {
        return React.createElement(
            "div", {"className": "player-list"},
            this.state.players.map(player => {
                return React.createElement(
                    Player, {
                        "key": player,
                        "name": player,
                        "onJoin": this.props.onJoin,
                    }
                )
            })
        )
    }

    // ...
}

class App extends React.Component {
    constructor(...params) {
        super(...params)

        this.state = {
            "player": null,
            "socket": null,
        }

        this.onJoin = this.onJoin.bind(this)
    }

    onJoin(name) {
        this.setState({
            "player": name,
        })
    }

    render() {
        let child = null

        if (this.state.player !== null) {
            child = React.createElement(
                Game,
                {"socket": this.state.socket}
            )
        } else {
            child = React.createElement(
                PlayerList,
                {
                    "socket": this.state.socket,
                    "onJoin": this.onPlayerClick,
                }
            )
        }

        // ...
    }

    // ...
}
```

Listing 15. from `public/js/app.js`

```
function Game() {
    return React.createElement("h1", null, "Game")
}

class Player extends React.Component {
    constructor(...params) {
        super(...params)
        this.onClick = this.onClick.bind(this)
    }

    onClick(e) {
        e.preventDefault()
        this.props.onJoin(this.props.name)
    }

    render() {
        return React.createElement(
            "a",
            {
                "onClick": this.onClick,
                "className": "player",
                "href": "#",
            },
            this.props.name
        )
    }
}

function PlayerList(props) {
    return React.createElement(
        "div",
```

See this month's code archive for the full listing.

Building the Arena

It's time to start building the battle arena. It's a good idea to try and restrict things to a fixed amount of concurrent games, but we've also built the server and client to be able to handle any number of concurrent games. Before we start the games off, let's define a set number of starting positions in Listing 16.

These constants are defined on the `Game` class shown in Listing 17.

We've created nine spaces for new arenas to be built. Let's also define `clear` and `build` methods, the former to clear away old arenas, and the latter to build them up again in Listing 18.

We can execute the `fill` command in much the same way as we teleport the player around, or whisper to them. It takes `x`, `y`, and `z` start and end coordinates. So, repeating this, we can build a box and fill the bottom of it with lava. Then, we can create a platform above that lava, and teleport the player there.

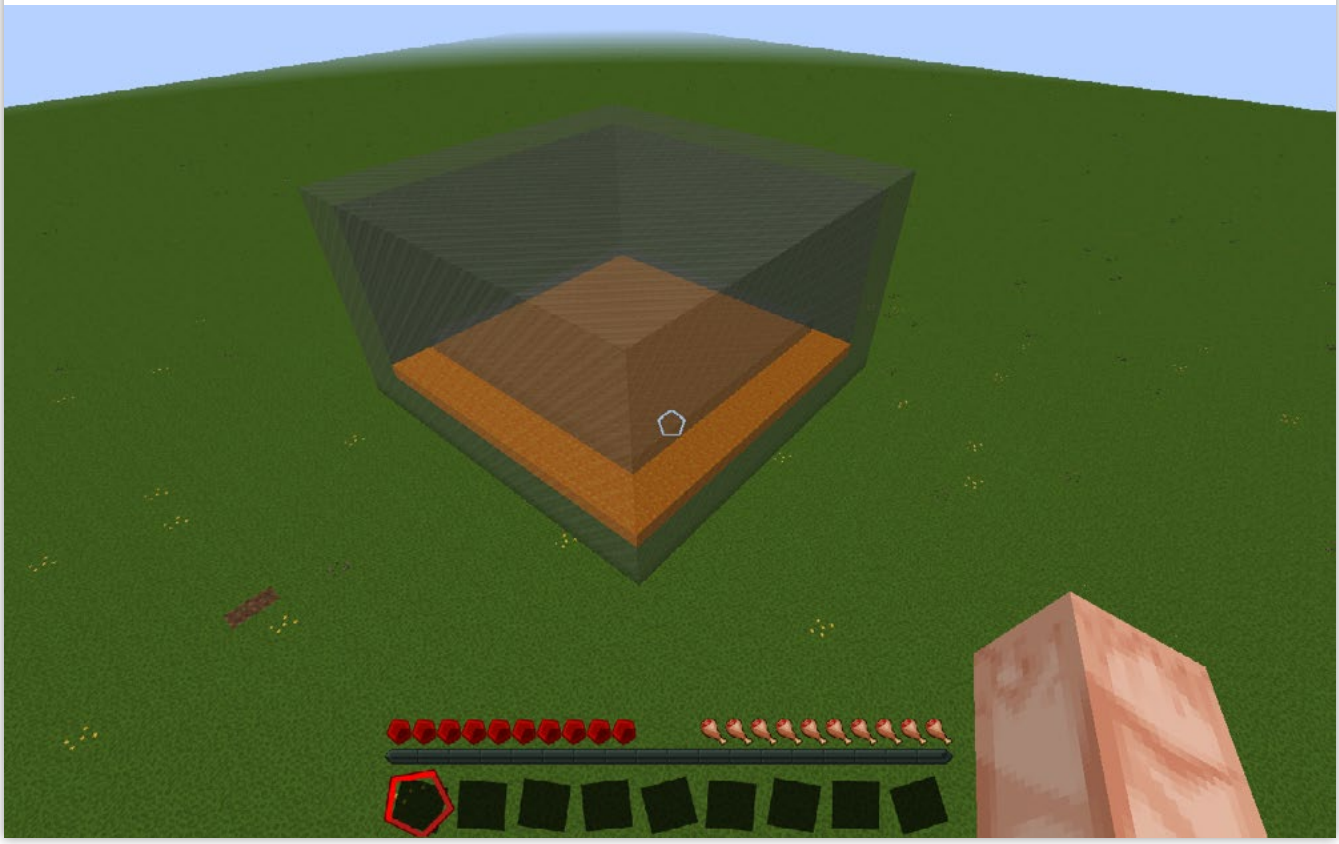
Deciding which position to create a new arena in, is a tricky task (see Listing 19).

We begin by clearing each potential arena position, at the script starts. This clears out any mess left by a previous server crash. Next, we create a repeating function. It's ok for the callback to be blocking because there's no asynchronous code we need to run in it. It would be ideal if the builder library were asynchronous, or if we wrapped it in `amphp/parallel`.

If there are fewer than nine running games (i.e. nine teams busy playing) and there are still games queued, we pop one off the back of the array and start to build it. Not before clearing some space, though.

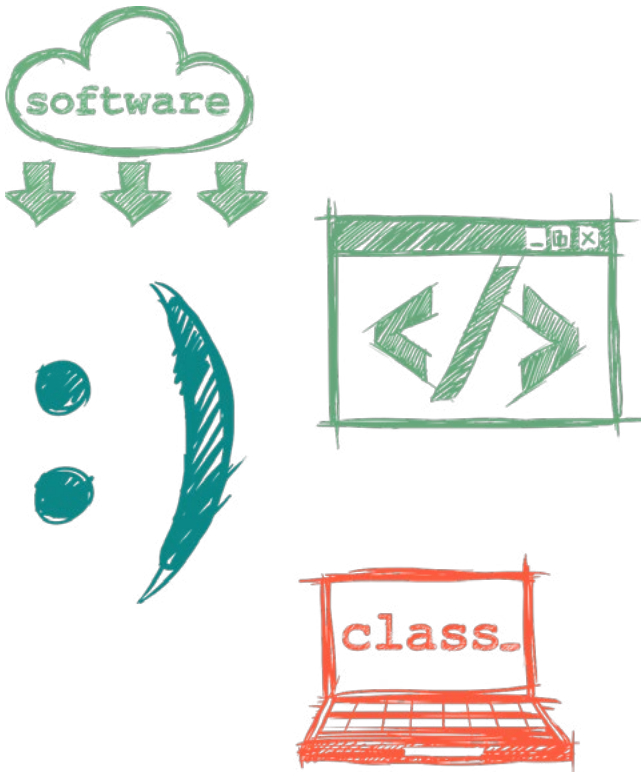
Finally, we add it to the array of running games, and back on to the start of the socket's games array. This combination of pop/shift ensures games are processed in a first-in, first-out order.

Watching as arenas are built...



Listing 16. from `src/Socket.php`

```
1. public function __construct(Client $builder) {
2.     $this->builder = $builder;
3.
4.     Loop::execute(
5.         Amp\wrap(
6.             function () {
7.                 yield Amp\File\put($this->path, "");
8.             }
9.         )
10.    );
11.
12.    $this->setPositions();
13. }
14.
15. private function setPositions() {
16.    $parts = explode(" ", $this->waitingCoordinates);
17.
18.    $x = $parts[0] - 11;
19.    $y = 4;
20.    $z = $parts[2] - 11;
21.
22.    $this->positions = [
23.        Game::POSITION_NORTH_WEST => [$x - 22 - 3, $y, $z - 22 - 3],
24.        Game::POSITION_NORTH => [$x - 22 - 3, $y, $z],
25.        Game::POSITION_NORTH_EAST => [$x - 22 - 3, $y, $z + 22 + 3],
26.        Game::POSITION_WEST => [$x, $y, $z - 22 - 3],
27.        Game::POSITION_ORIGIN => [$x, $y, $z],
28.        Game::POSITION_EAST => [$x, $y, $z + 22 + 3],
29.        Game::POSITION_SOUTH_WEST => [$x + 22 + 3, $y, $z - 22 - 3],
30.        Game::POSITION_SOUTH => [$x + 22 + 3, $y, $z],
31.        Game::POSITION_SOUTH_EAST => [$x + 22 + 3, $y, $z + 22 + 3],
32.    ];
33. }
```



Listing 17. from `src/Game.php`

```
1. class Game
2. {
3.     const POSITION_NONE = 0;
4.     const POSITION_NORTH_WEST = 1;
5.     const POSITION_NORTH = 2;
6.     const POSITION_NORTH_EAST = 3;
7.     const POSITION_WEST = 4;
8.     const POSITION_ORIGIN = 5;
9.     const POSITION_EAST = 6;
10.    const POSITION_SOUTH_WEST = 7;
11.    const POSITION_SOUTH = 8;
12.    const POSITION_SOUTH_EAST = 9;
13.
14.    // ...
15. }
```

Listing 18. from `src/Game.php`

```
1. public function build(Client $builder, int $x, int $y,
2.                       int $z, $then = NULL) {
3.     $builder->exec("fill ... stained_glass 7");
4.     $builder->exec("fill ... stained_glass 7");
5.     $builder->exec("fill ... stained_glass 7");
6.
7.     // ...
8.
9.     $builder->exec("fill ... lava");
10.    $builder->exec("fill ... stained_glass 7");
11.
12.    if (is_callable($then)) {
13.        $then();
14.    }
15. }
16.
17. public function clear(Client $builder, int $x, int $y,
18.                       int $z, $then = NULL) {
19.    for ($i = 0; $i < 20; $i++) {
20.        $builder->exec(
21.            sprintf(
22.                "fill %s %s %s %s %s %s air",
23.                $x, $y + $i, $z, $x + 22, $y + $i, $z + 22
24.            )
25.        );
26.    }
27.
28.    if (is_callable($then)) {
29.        $then();
30.    }
31. }
```

Where to Go from Here?

In these two articles, we've looked at how to do everything from creating a ReactJS/socket interface to building a Minecraft world. Where you take it from here is only limited by your imagination.

You could, for instance, provide a list of code questions, sending them to the web client. The coder would have to answer them or put the Minecraft player in more peril. That's exactly what I did the first time I built this game.

If you'd like to see how I asked and answered those questions, check out a previous (albeit messy) implementation, at theorymc/coder[8]. You could try for something more straightforward, like a general knowledge kind of game.

Hopefully, you have a deeper understanding of what PHP applications are capable of, and what creative things we can start to build when we're able to step outside our comfort zones. I trust you have had the chance to learn a bit more about asynchronous PHP, WebSockets, and ReactJS interfaces.

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

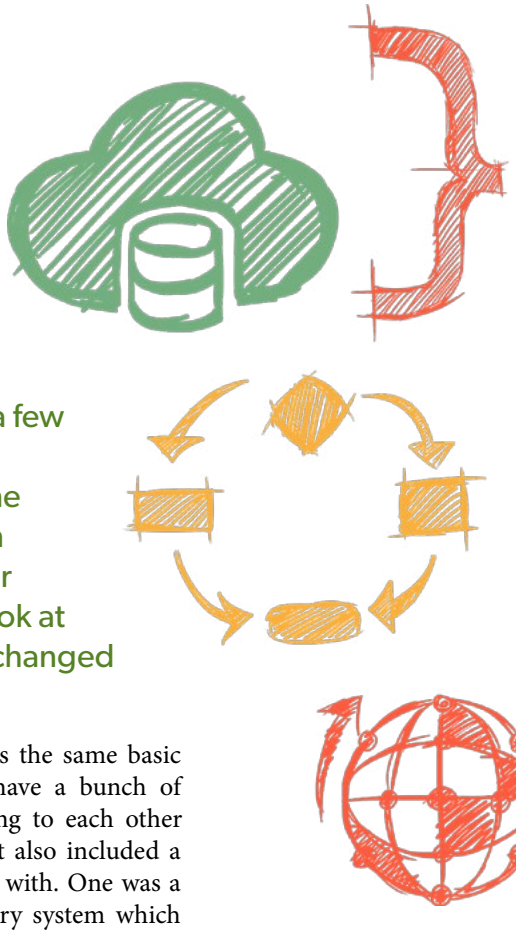
8 *theorymc/coder:* https://github.com/theorymc/coder

Listing 19. from `config.php`

```
<?php
foreach ($socket->positions as $position) {
    $x = $position[0];
    $y = $position[1];
    $z = $position[2];

    for ($i = 0; $i < 15; $i++) {
        $builder->exec(sprintf(
            "fill %s %s %s %s %s air",
            $x, $y + $i, $z, $x + 22, $y + $i, $z + 22
        ));
    }
}

Loop::repeat(1000, function () use ($builder, $socket) {
    static $running;

    if (is_null($running)) {
        $running = [];
    }

    $running = array_filter(
        $running,
        function ($game) use ($socket) {
            return in_array($game, $socket->games);
        }
    );

    if (count($running) < 9 && count($socket->games) > 0)
{
        $game = array_pop($socket->games);

        if ($game->position !== Game::POSITION_NONE) {
            return;
        }

        $available = [
            Game::POSITION_NORTH_WEST => TRUE,
            Game::POSITION_NORTH => TRUE,
            Game::POSITION_NORTH_EAST => TRUE,
            Game::POSITION_WEST => TRUE,
            Game::POSITION_ORIGIN => TRUE,
            Game::POSITION_EAST => TRUE,
            Game::POSITION_SOUTH_WEST => TRUE,
            Game::POSITION_SOUTH => TRUE,
            Game::POSITION_SOUTH_EAST => TRUE,
        ];

        foreach ($running as $next) {
            unset($available[$next->position]);
        }

        $game->position = array_shift($available);
        $position = $socket->positions[$game->position];

        $x = $position[0];
        $y = $position[1];
        $z = $position[2];

        $game->clear(
            $builder, $x, $y, $z,
            function () use ($x, $y, $z, $game, $builder,
                             $position) {
                $game->build(
                    $builder, $x, $y, $z,
                    function () use ($x, $y, $z, $game,
                                     $builder, $position) {

                        $builder->exec(
                            sprintf(
                                "/tp {$game->player} %s %s %s",
                                $x + 11, $y + 5, $z + 11
                            )
                        );
                    }
                );
            }
        );

        array_push($running, $game);
        array_unshift($socket->games, $game);
    }
});
```

Deploying to Docker Swarm

Chris Tankersley



Docker has come a long way in the last few years. It's now possible to set up Docker as easily as other development tools on any operating system most people use, and (barring a few weird OS-specific gotchas) you can comfortably work in Docker from just about any machine. Some people, like myself, do not even install Vagrant on their machines anymore and opt for a pure Docker development environment. In this article, we'll look at how deploying your Dockerized application has changed and become easier.

At some point in time, you will need to move code from your machine into a production environment. While you can comfortably build your application on Docker and eventually move it into a non-containerized production environment, Docker is, at its heart, a deployment tool. It wants very much to remove the environment as a factor when deploying applications.

Docker Swarm 1.0 was released in November of 2015, alongside Docker Engine 1.9. While it had been in beta for a while, Docker 1.9 added support for multi-host networking and better volume support. This meant it was easier for containers to talk across different systems, and volumes could use distributed plugins like Flocker[1] or NFS to share volumes between all of the nodes in a cluster. These were big steps forward, but the tooling around this was still very primitive. Because of this, we saw the rise of tools like Rancher[2] and Kubernetes[3].

Docker Engine 1.12 released with a new subsystem called "Swarm Mode." It replaced the old standalone Swarm project and was released as part of Docker Engine. It was the same basic idea, which was to have a bunch of Docker systems talking to each other over a network, but it also included a few new ideas to play with. One was a better service discovery system which used a basic DNS server built into Docker's networking layer. A second important part was the ability to define services, which could be scaled up and down. The third, and probably the most powerful, was a new routing system which enabled clusters to route traffic to containers regardless of which node a client hit, or which node a container existed on.

Not all was great. Docker Engine 1.12 was still missing an easy-to-use tool to manage all of this. You could easily set up a cluster, but all of the scaling up and down, configuring, and wiring of the containers was manual. This was miles ahead of Docker Swarm, but it was still a tough barrier to entry. Swarm mode also did not work well with Docker Compose, which was the orchestration tool provided by Docker many people use. This meant configuring your application in two different formats.

Docker Engine 1.13 finally delivered on many of the toolings promised for Swarm mode 1.12. Swarm mode in Docker Engine 1.13 includes:

- Multi-host networking and routing
- Multi-host volume support
- Service discovery across nodes
- Integration with Docker Compose and Docker Machine
- Compatible workflows with Docker Compose

All of this adds up to the ability to start a project, build a development environment around Docker, and deploy it to production with minimal changes. The rest of this article will help fill in that last part as we take a sample application and deploy it to a Docker Swarm cluster.

The Setup

This article assumes you understand how to use Docker in-and-of-itself, and have been using Docker Compose to develop your application. If not, I would recommend looking over the

1 *Flocker:* http://phpa.me/flocker-intro

2 *Rancher:* http://rancher.com

3 *Kubernetes:* https://kubernetes.io

Sponsors

php women

OSMI

VPSSERVER.com

INFINITE RED

php[architect]

OmniTI

CIOReview

Drupal Watchdog

stickermule

ActiveState
THE OPEN SOURCE LANGUAGES COMPANY

Roave

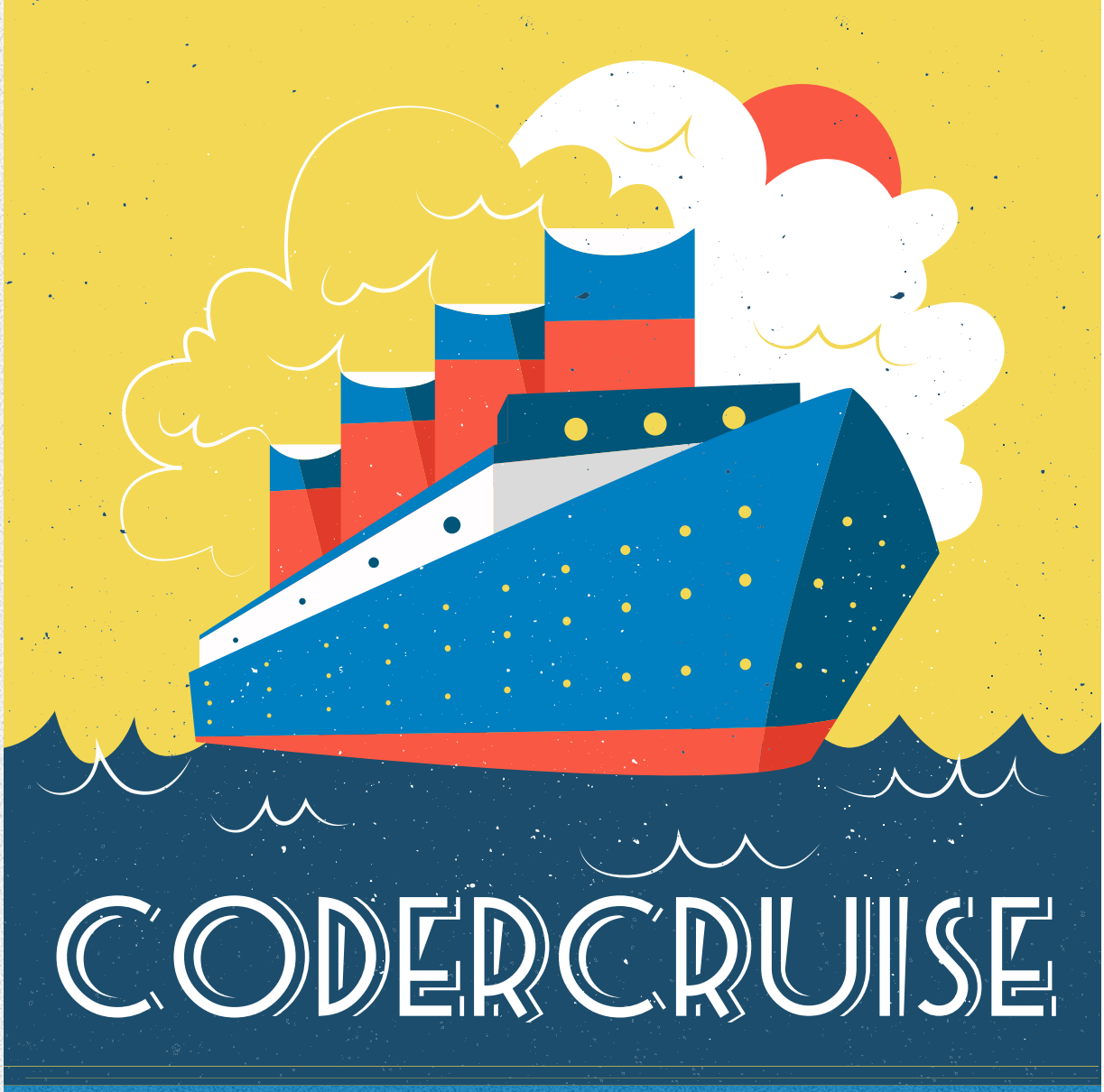
When I Work

contentful

GitHub

Engine Yard

salesforce



CODERCRUISE

7 days at sea, 3 days of conference

Leaving from New Orleans and visiting
Montego Bay, Grand Cayman, and Cozumel

July 16-23, 2017 — *Tickets $295*

www.codercruise.com

Presented by One for All Events

documentation for `docker run`[4] and Docker Compose[5]. php[architect] also carries my *Docker For Developers* book[6] available on their website.

If you want to see an application which is completely set, clone the example app located on GitHub at dragonmantank/dockerfordevs-app[7]. Then, check out the `demo/docker-compose-deploy` branch. This branch will have all of the modifications needed for running the application and deploying it to a swarm.

To get the sample application running, do the following:

```
docker-compose up -f docker-compose.yml \
 -f docker-compose.dev.yml -d
```

This will build the images, configure them to work in the development environment, and start up the containers. You can install the dependencies through Composer by doing:

```
$ docker-compose -f docker-compose.yml \
 -f docker-compose.dev.yml run \
 --entrypoint composer composer install
```

This will use the special `composer` service we have set up which has Composer[8] installed and a PHP environment with all of our system extensions installed. Next, we will run our database migrations. First, run the following command to set up a config file:

```
$ docker-compose -f docker-compose.yml \
 -f docker-compose.dev.yml \
 run --entrypoint php \
 -w /var/www phpserver vendor/bin/phinx init
```

This will create a `phinx.yml` file in the root of the project. Open it up and edit the `development` section to match our Docker setup. This means:

- `host` changes from `localhost` to `mysqlserver` to match the MySQL service name
- `name` changes to `dockerfordevs`, to match the `docker-compose.yml` `MYSQL_DATABASE` entry
- `password` changes to `docker`, to match the `docker-compose.yml` `MYSQL_ROOT_PASSWORD` entry

We can then run the following to set up the database:

```
$ docker-compose -f docker-compose.yml \
 -f docker-compose.dev.yml run \
 --entrypoint php \
 -w /var/www phpserver vendor/bin/phinx migrate
```

Finally, copy `config/autoload/local.php.dist` to `config/autoload/local.php` to copy over the configuration for the database into the PHP application itself.

4 `docker run`: http://phpa.me/docker-run-ref

5 *Docker Compose:* https://docs.docker.com/compose/

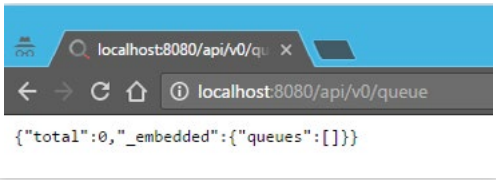
6 *Docker For Developers book:* http://phpa.me/docker-devs

7 *dragonmantank/dockerfordevs-app:* https://github.com/dragonmantank/dockerfordevs-app

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

If it all goes well, you should see the JSON output like in Figure 1.

Figure 1



Set up a Swarm

Swarm is nothing without a set of machines to deploy to. Swarm requires a set of machines, called nodes, which are controlled by servers in the cluster called managers. We'll set up a small cluster with three nodes and one manager. Keep in mind you can deploy multiple managers if you need to for fail over. This is a basic setup, so we will just create one.

I'll be using Docker Machine with Hyper-V to create virtual machines on my local PC for this. You can use any machine type that Docker Machine supports, such as VirtualBox, Digital Ocean, or AWS. They may require slightly different parameters for the machine creation, so please review the docker-machine create documentation[9] if you need to use something else.

Let's create our manager node:

```
$ docker-machine create --driver hyperv \
 --hyperv-virtual-switch "WIFI" manager1
```

This will generate a new machine for us named `manager1`. We can now set up the Swarm mode by SSHing into the remote manager and running `swarm init` (see Listing 1).

We can now create two other nodes and use the `docker swarm join` command and the token to attach them to

Listing 1

```
1. $ docker-machine ssh manager1
2. manager1$ docker swarm init
3. Swarm initialized: current node (wtcyiri2t8xbeezjlklik8omw)
4. is now a manager.
5.
6. To add a worker to this swarm, run the following command:
7.
8.     docker swarm join \
9.     --token SWMTKN-1-0u6f9a5wi7n9gseo5kbdu7zegi1n6bogdljc0gt
l98oiiulzig-8ozfdtwsewpmjnp3a34xal0w1 \
10.     172.16.0.204:2377
11.
12. To add a manager to this swarm, run 'docker swarm join-token
13. manager' and follow the instructions.
14. manager1$ exit
```

9 *docker-machine create documentation:* https://docs.docker.com/machine/reference/create/

Listing 2

```
$ docker-machine create --driver hyperv --hyperv-virtual-switch "WIFI" node1
$ docker-machine create --driver hyperv --hyperv-virtual-switch "WIFI" node2
$ docker-machine ssh node1
node1$ docker swarm join --token [token] 172.16.0.204:2377
This node joined a swarm as a worker.
node1$ exit
$ docker-machine ssh node2
node2$ docker swarm join --token [token] 172.16.0.204:2377
This node joined a swarm as a worker.
node2$ exit
$ docker-machine ls
NAME     ACTIVE  DRIVER  STATE    URL                       SWARM  DOCKER    ERRORS
manager1 -       hyperv  Running  tcp://172.16.0.204:2376          v1.13.1
node1    -       hyperv  Running  tcp://172.16.0.201:2376          v1.13.1
node2    -       hyperv  Running  tcp://172.16.0.202:2376          v1.13.1
```

the swarm as in Listing 2.

We now have three machines which we can deploy our application to. Everything we do to the swarm we should do through the manager, so it gets properly propagated to the rest of the nodes. Run `docker-machine env manager` to get the eval expression you need to connect to `manager1`. All the commands from here on out will be run on the `manager1` node.

Deploying to the Swarm

Docker Engine 1.12 introduced the idea of services, which were configurations you could set up on a Swarm cluster. You can still do this by hand, but it is a pain to maintain, as Docker was ultimately working toward a single command which would allow you to handle all of this through a bundles system. You might still see some references to things like "Distributed Application Bundles," a holdover from those original experiments.

With Docker Engine 1.13 we now have a `docker stack deploy` command which can work with Docker Compose files directly without having to create these DAB files. There are a few things we need to be aware of when working with this deployment service.

The first is, generally, you will be deploying one application to a swarm, and the idea is the application will scale up and down inside the cluster. You will not be able to deploy two applications that use the same ports. For example, if you are building our demo app which uses port 80 and port 443 in production, you can't turn around and also deploy a WordPress app which also uses port 80 and 443. Ports are unique per "stack," or collection of images.

The second is that volumes must be configured to be available across the entire swarm, or at least the nodes you plan on letting containers scale around on. In general, each node must be configured separately to talk to whatever distributed volume system you want to use, be it Flocker, NFS, or something else. This sample project is not taking that into account, so our MySQL container will not scale like other containers would (though, in general, you would not contain your database).

The third is that we cannot build images on demand like we can with a local setup of Docker Compose. This is due to the way the commands are sent out to the individual nodes and how they handle scaling. They need to be able to pull an image directly from a repository. For our sample application, we'll need to build our images, push them to a private repo (https://hub.docker.com, in this case), and have our compose file point to them.

Building the Images

There isn't anything special about our images, other than they have to exist as built artifacts before we deploy to production. If you take a look at our `docker-compose.yml` file, you'll see the `phpserver` and the `nginx` services both have a `build:` key. This build key is further expanded into a `dockerfile:` and a `context:` setting. We can reproduce this with the `docker build` with the `-f` parameter to pass it a specific Dockerfile to use and specifying a build context of `./` instead of the path where the Dockerfile lives.

Before we build the images, I'm going to take a quick detour on the subject of tagging images. There are three types of tags an image can have:

- **word**: `php` - An officially released image from Docker.
- **username/word**: `dragonmantank/php` - A community image for PHP released by the user dragonmantank on hub.docker.com.
- **hostname:port/username/word**: `myregistry.lan:5000/dragonmantank/php` - An image being pushed to a private registry.

You should be very familiar with the first two syntaxes, as the images you build from in your Dockerfiles are probably made from an official release or a community release of an image. In a real life production setup, you will most likely want to consider setting up a private registry since you may not want your custom images available to everyone. That is a bit out of scope for this article, but you can investigate the official documentation.[10]

Of course, after each of the syntaxes you can specify a `:version` to set up different versions of the same image. For the official PHP image there are options such as `php:7-fpm`, `php:apache`, `php:5.6-cli`, and others specific to releases of PHP in different setups.

Docker Hub does have a paid version which uses the same syntax as the second option. I'll be using the free private repository they give each user. We'll create a single repository and host two versions of images, one for the NGINX container and the other for the PHP container. This is a roundabout way of getting multiple images up for free to play with, but I

10 *the official documentation.:* https://docs.docker.com/registry/deploying/

would encourage you to either deploy a free private registry or pay for Docker Hub, which is very inexpensive.

For this article, I've set up a private repo on Docker Hub called `dragonmantank/pa-sampleapp`.

With that out of the way, let's create and tag two images.

```
$ docker build -t dragonmantank/pa-sampleapp:nginx \
 -f docker/nginx/Dockerfile ./
$ docker build -t dragonmantank/pa-sampleapp:phpserver \
 -f docker/php/Dockerfile ./
```

These images should build just as they did under `docker-compose`, but with a different naming scheme. Next, log into hub.docker.com through the CLI and push both of them up.

```
$ docker login
$ docker push dragonmantank/pa-sampleapp:phpserver
$ docker push dragonmantank/pa-sampleapp:nginx
```

After a few minutes (or longer, depending on your upload speed), the images will be pushed up to Docker Hub and will be available to any set of machines I'm logged in to.

Creating a Production Config

Now that the images are available for pulling down, we can turn our attention to the Docker Compose configuration for production.

If you look at the files pulled down in Git, there are three configuration files for our application. The default one, `docker-compose.yml`, contains the bare minimum we need to sketch out our application. It has our volume for MySQL, as well as our three defined services (`phpserver`, `nginx`, and `mysqlserver`). We are not specifying any ports or volume mounts as we will let the secondary configurations take care of that.

`docker-compose.dev.yml`, as we used above, sets up our environment to work on `localhost:8080`, has our build instructions for the images, and does the host volume mounting so we can work with the files live. The `docker-compose.prod.yml` file specifies the exact images we will use, as well as a more production-like port forwarding setup on port 80.

With all that in mind, we now turn to our swarm cluster. Make sure you are connected to it and run the `docker login` command against it to log into Docker Hub. As part of the deploy, we will tell the swarm master to pass the login information to each of the nodes so they can all pull from our custom images.

Once you are logged in, go ahead and deploy:

```
$ docker stack deploy -c docker-compose.yml \
 -c docker-compose.prod.yml \
 --with-registry-auth pa-sampleapp
```

If it all goes well, you should end up back at your CLI prompt with no errors. We told Docker to deploy our compose file as a Service Stack, so it converted our compose files into a stack definition, and deployed it all to a stack named `pa-sampleapp`. It also passed along our Docker Hub credentials with the `--with-registry-auth` parameter so each node could pull down the private images.

We can check the status of the stack by running `docker stack ps pa-sampleapp`:

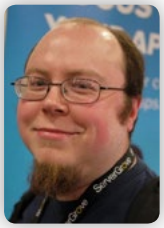
```
$ docker stack ps pa-sampleapp
ID            NAME                          IMAGE
NODE      DESIRED STATE  CURRENT STATE           ERROR
tru6e7ttu964  pa-sampleapp_mysqlserver.1  mysql:latest
node2     Running        Preparing 3 minutes ago
ptk1hnd92l91  pa-sampleapp_phpserver.1    dragonmantank/pa-sam-
pleapp:phpserver  manager1  Running        Preparing 5 minutes
ago
ruubz6inerie  pa-sampleapp_nginx.1        dragonmantank/pa-sam-
pleapp:nginx      node1     Running        Preparing 5 minutes
ago
```

The deploy can take a few minutes, depending on your internet connection and how many layers there are. We can see, though, that the swarm deployed our MySQL server to node2, our PHP-FPM instance to manager1, and our NGINX proxy to node1. Once the images are all out of the "Preparing" state, we can go to the IP of any of those machines on port 80, and we should be greeted with the web application. In my case, I can go to `http://172.16.0.204`, `http://172.16.0.201`, or `http://172.16.0.202` and all of the traffic is routed to the NGINX container, no matter where it needs to go.

From Here

Now that you have your containers in Swarm, you can take a look at the other deploy options[11] available for Docker Compose. You can specify things such as how many replicas to run, what the restart policy is if a node restarts, where the containers can be placed, and many other options. If you need to scale up, you can simply change the `replicas:` key in your production Docker Compose file, run another `docker stack deploy`, and the system will scale out as needed.

I hope this helps push you in the correct direction when it comes to deploying to Docker Swarm. There are some factors to consider on top of this, such as where user uploads live, how you are managing your database, securing the back-channel talk between all of the nodes, amongst other things. From there though, your Docker projects can finally get into Production.



Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. He works for InQuest, a network security company out of Washington, DC, but lives in Northwest Ohio. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers *and works with companies and developers for integrating containers into their workflows.* @dragonmantank

11 *deploy options:* http://phpa.me/compose-deploy-opts

How to Use SELinux (No, I Don't Mean Turn It Off)

Chuck Reeves

"I turn off SELinux because it is hard to configure." "SELinux just gets in the way." Those are just some of the arguments I hear when asking people why they turn off SELinux. Requiring Sudo gets in the way of looking at my logs. iptables is hard to configure (harder than SELinux in my opinion). You wouldn't run a production system with those turned systems off would you? Why not take the time to learn what SELinux is doing and how it locks down your system? SELinux is a powerful security layer developed by the NSA (you know—the people who listen to our phone calls) to lock down the Linux kernel and curve misconfigured servers from leaking government secrets. I'll grant you that SELinux *was* hard to configure and understand back in 2004. Now, it is a mature well-documented product no system administrator should turn off.

SELinux expands the security of the system using kernel modules to add a Mandatory Access Control (MAC) layer to all the inodes and processes the kernel manages. This level of security expands the standard Discretionary Access Control (DAC) you currently set on files. This gives you finer control over what a process has access to without the need to configure multiple services.

Mandatory Access Control[1] **(MAC)** refers to a type of access control by which the operating system constrains the ability of a subject or initiator to access or perform some sort of operation on an object or target. In practice, a subject is usually a process or thread; objects are constructs such as files, directories, TCP/UDP ports, shared memory segments, IO devices, etc. Subjects and objects each have a set of security attributes. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system kernel examines these security attributes and decides whether the access can take place. In Linux, a MAC enables additional checks beyond the basic user+group file permissions.

Discretionary Access Control[2] **(DAC)** is a type of access control defined by the Trusted Computer System Evaluation Criteria[1] "as a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)". In Linux, users and groups are given file permissions for read, write, execute operations from the DAC.

A Brief History

In 1989, the NSA working group, Trusted UNIX (TRUSIX), published the "Grey Silver" Rainbow Book—it's a long title. You can Google it by its document title: NCSC-TG-020-A[3]. The group noticed the current DAC model had a finite number of group to user combinations as insufficient for the government (the DoD in this case). The group pointed out that DAC was "designed for efficiency, flexibility, and ease of use … and encourages the sharing of information," which governments do not like.

The NSA got to work and created SELinux to demonstrate how to implement a MAC in a POSIX system. In those days, it had to be patched into the kernel. In December 2000, the NSA open sourced SELinux which was finally merged into the 2.6 kernel in 2003. You might have known it as the Flux Advanced Security Kernel (FLASK).

The NSA expressed it was only a prototype in 2000. It was poorly documented (as most OSS projects) and difficult to configure. The goal of SELinux is to "confine the actions of any process, including an administrator process." It is this goal which is

1 *Mandatory Access Control:* http://phpa.me/mandatory-ac

2 *Discretionary Access Control:* http://phpa.me/discretionary-ac

3 *NCSC-TG-020-A:* http://phpa.me/NCSC-TG-020-A

counter-intuitive to any Linux administrator (after all, we should bow before them because they are root).

What You Need to Know

SELinux can be used to manage users, sockets, memory, directories and network connections. It will apply a "context" to each inode which identifies a user, role, type, and level. It is important to note SELinux users are not the same as the system users. Each context is then attached to a policy. When a process wants to access an inode, a decision to allow access is made, then cached in the Access Vector Cache (AVC). It is important to note that SELinux makes it decision after the traditional DAC is checked.

A process can either be an unconfined process (which just falls back to the standard kernel DAC) or a confined process. Confined processes are what SELinux attaches its role to. A confined process will then be limited to their own roles and policies. The TL;DR; if a process is running with `<context_foo>` then anything with `<context_foo_type>` is allowed access. Note, the level part is optional which can be thought of like "Top Secret" versus "Secret" access. Level is mainly used if you need to run Multi-Level Security (MLS) which can take up its own article.

Since a process is used for a check instead of just a user, SELinux can block access to the resources utilized by other processes. For example, when Heartbleed was an issue in OpenSSL, I noticed SELinux was blocking memory allocated from other processes. I was still able to get arbitrary data that was loaded into memory, but only the memory that the HTTPD process had allocated. The audit logs showed that SELinux was preventing httpd from memory outside its allocation.

Contexts

Contexts are typically set on nodes in a number of ways. Most of the time, they will get set automatically from the parent node. RPMs will set contexts during installation along with management tools (Ansible, Chef, Puppet). Contexts will change when a file transitions to a new location (moving an uploaded file). And, finally, the admin can set the context using tools like chcon or restorecon.

SELinux adds the -Z flag to a lot of the standard processes to help you determine what context is applied to a file:

```
$ ls -Z
-rw-rw-r--. apache apache system_u:object_r:httpd_sys_
content_t:s0 composer.json
-rw-rw-r--. apache apache system_u:object_r:httpd_sys_
content_t:s0 composer.lock
drwxrwxr-x. apache apache system_u:object_r:httpd_sys_
content_t:s0 public/
drwxrwxr-x. apache apache system_u:object_r:httpd_sys_
content_t:s0 vendor/
```

Here we can see the public folder has the context of `system_u:object_r:httpd_sys_content_t:s0`. When apache wants to access this file, it needs to be considered a system user with access to the `httpd_sys_content_type`. We can also use `ps` to find the context of our processes:

```
$ ps -Z aux | grep httpd
unconfined_u:system_r:httpd_t:s0 root     3363  0.0  2.2 399820
11176 ?        Ss   Dec09   0:04 /usr/sbin/httpd
unconfined_u:system_r:httpd_t:s0 apache   4463  0.0  1.0 399820
5284 ?         S    11:48   0:00 /usr/sbin/httpd
```

Booleans

Booleans are toggles for policies. They help curve the need to over label everything and overly complicate your policies. For example, allow HTTPD to make network connections or allow FTP to access home directories. To find out which Booleans are enabled or disabled; you can run the `semanage boolean -l` command (there are a lot of Booleans so use grep to search the list).

```
$ semanage boolean -l | grep http
14:httpd_can_network_relay        (off  ,  off)  Allow httpd to
act as a relay
22:httpd_can_network_connect_db   (on   ,   on)  Allow HTTPD
scripts and modules to connect to databases over the network.
27:httpd_use_gpg                  (off  ,  off)  Allow httpd to
run gpg in gpg-web domain
# and so on ...
```

Notice `off` appears twice for `httpd_can_network_relay`. That just means it is off now and when the system boots. Toggling Booleans is done with the `setsebool` command and be sure to pass the `-P` flag to make it persistent:

```
$ semanage boolean -l | grep httpd_can_network_connect
113:httpd_can_network_connect      (off  ,  off)  Allow HTTPD
scripts and modules to connect to the network using TCP.
$ setsebool -P httpd_can_network_connect 1
$ semanage boolean -l | grep httpd_can_network_connect
113:httpd_can_network_connect      (on   ,   on)  Allow HTTPD
scripts and modules to connect to the network using TCP.
```

That's about all you need to know about SELinux. Let's go ahead and turn it back on and see what happens.

Turning It Back On

> *If you need instructions for installing SELinux, Google is your friend. You can find instructions for CentOS 7[4], Debian[5], Ubuntu[6], and more. For popular distros, all it may require is installing the right packages.*

There are 3 modes that you can run SELinux in (you know one already):

1. `ENFORCING`: Run SELinux in all its glory.
2. `PERMISSIVE`: Run SELinux, but just log out denials

4 *CentOS 7:* http://phpa.me/selinux-centos7

5 *Debian:* https://wiki.debian.org/SELinux/Setup

6 *Ubuntu:* https://wiki.ubuntu.com/SELinux

instead.

3. `DISABLED`: Turn it off.

If you are running brand new application ENFORCING will be a fine mode to start in. But, if you have turned it off or if you have an existing application, PERMISSIVE is the mode you want. PERMISSIVE allows you to check if SELinux is configured correctly for your application without blocking access you might need.

Open your favorite editor and change the SELINUX parameter `/etc/selinux/config` to `PERMISSIVE`. To help you understand the deny errors, install `setroubleshoot` and `setroubleshoot-server`. Now you will have a lot of files which do not have any contexts set to them. Rather than running `chcon` on every file, you can `touch /.autolabel` and reboot your server. This will tell SELinux to set default contexts on all the nodes. Now go ahead and do everything in your application. By running your application, you will start to see messages appear in the syslog and audit logs.

Troubleshooting Example

Imagine you have the simple script in Listing 1 output data from the database.

Checking the audit log we might see something like:

```
type=AVC msg=audit(1476826669.529:785): avc:  denied  \
 { name_connect } for  pid=26303 comm="httpd" dest=3306 \
 scontext=unconfined_u:system_r:httpd_t:s0 \
 tcontext=system_u:object_r:mysqld_port_t:s0 tclass=tcp_socket
```

It's confusing right? Well, there is an easier way to correct SELinux. Checking syslog we see the following:

```
May 13 23:36:07 Tek setroubleshoot: SELinux is preventing \
  /usr/sbin/httpd from name_connect access on the tcp_socket. \
  For complete SELinux messages. run sealert -l \
  ae1bf676-1345-4407-b7a0-7bfceedba0bc
```

Look at that; we get the exact command we need to run. You will see something similar to Listing 2.

Look how easy that is! We get a more user friendly error message and some suggestions on how to fix the issue (and how confident the suggestion is). Now we just need to decide which Boolean to enable. If your application only makes database connections, then the `httpd_can_network_connect_db` is the better choice. `httpd_can_network_connect` could allow malicious PHP code to upload sensitive data to a third party. As a general rule of thumb, choose a more restrictive Boolean and only use a more permissive one if you really find you need to do so.

Custom Policies (or Modules)

At the end of the day, the default contexts and Booleans might not work for your needs. In that case, you can create a custom policy using the `audit2allow` command. `audit2allow` can parse an entry in the audit log and build out the basics for a policy profile. Even though there exists a policy to let HTTPD connect, let's use this example to see what is going

Listing 1

```php
<?php

// Test network connection
try {
    $pdo = new \PDO('mysql:host=10.0.2.2;dbname=selinux',
                    'se', 'selinux');

    $stmt = $pdo->query('SELECT * FROM Persons');
    while ($row = $stmt->fetch()) {
        var_dump($row);
    }
} catch( PDOException $e ) {
    echo  $e->getMessage();
}
```

Listing 2

```
$ sealert -l ae1bf676-1345-4407-b7a0-7bfceedba0bc
SELinux is preventing /usr/sbin/httpd from name_connect
access on the tcp_socket .

***** Plugin catchall_boolean (47.5 confidence) suggests
**********************

If you want to allow HTTPD scripts and modules to connect to
the network using TCP.

Then you must tell SELinux about this
by enabling the 'httpd_can_network_connect' boolean.
Do
setsebool -P httpd_can_network_connect 1

***** Plugin catchall_boolean (47.5 confidence) suggests
**********************
If you want to allow HTTPD scripts and modules to connect to
databases over the network.

Then you must tell SELinux about this by enabling the
'httpd_can_network_connect_db' boolean.
Do
setsebool -P httpd_can_network_connect_db 1
```

Listing 3

```
module mypolicy 1.0;

require {
    type mysqld_port_t;
    type httpd_t;
    class tcp_socket name_connect;
}

#============= httpd_t ==============

#!!!! This avc is allowed in the current policy
allow httpd_t mysqld_port_t:tcp_socket name_connect;
```

Listing 4

```
# grep postdrop /var/log/audit/audit.log | \
 audit2allow -M postfixlocal
# cat postfixlocal.te
    module postfixlocal 1.0;
    require {
            type httpd_log_t;
            type postfix_postdrop_t;
            class dir getattr;
            class file { read getattr };
    }
    #============= postfix_postdrop_t ==============
    allow postfix_postdrop_t httpd_log_t:file getattr;
```

on (notice the large `M` flag which is used to make a module package):

```
# grep mysql /var/log/audit/audit.log \
 | audit2allow -M mypolicy > mypolicy.te
```

Here you will see 2 files named `mypolicy.te` and `mypolicy.pp`. The `.te` file is human-readable so let's take a look at it with `cat mypolicy.te` (see Listing 3).

Here we can see the policy file shows two types. `mysqld_port_t` and `httpd_t` are allowed to open a TCP socket (and a warning that a samilar policy exists). Go ahead and compile using `checkmoduel` and `semodule_package`:

```
# checkmodule -M -m -o mypolicy.mod mypolicy.te
checkmodule: loading policy configuration from mypolicy.te
checkmodule: policy configuration loaded
checkmodule: writing binary representation (version 10) to
mypolicy.mod

# semodule_package -o mypolicy.pp -m mypolicy.mod
```

We now have a `mypolicy.mod` file which can now be loaded using `semodule`:

```
# semodule -i mypolicy.mod
```

Danger Zone

While `audit2allow` is a powerful tool, it is not the smartest. Be sure to review the policy to see what it opens. I have seen `audit2allow` create a policy which would allow any user (included guest users) the ability to open a TCP socket to any port and any host. Consider the example in Listing 4 (taken from the CentOS wiki).

Here `audit2allow` thinks postdrop needs access to the HTTPD error log. Ask yourself the question, "Do I really want this process to have this level of access?" If the answer is a resounding "no," we can go just continue to let SELinux continue blocking acess. But what about the audit.log? Now we are just getting noise from postdrop.

Conclusion

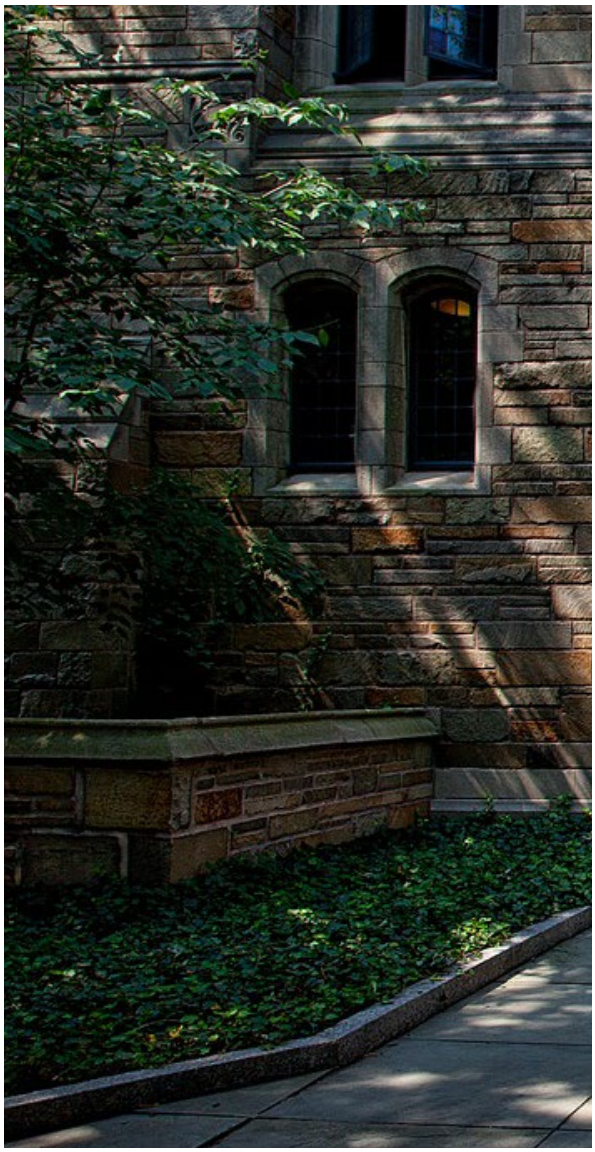
SELinux is not as hard to configure as you think. It is now easier than iptables to set up and allows you more control over the security of your system. That's not to say you don't need iptables, but SELinux gives you another way to restrict what each application on your machine can do. By restricting the directories, ports, and files which an application can use, the potential damage from compromising it can be mitigated.

A PHP developer for the past 15 years, and the secretary of WurstCon. I have contributed to many projects ranging from: online sweepstakes, custom CMS, eCommerce and Marketing, and Custom software. Currently working for Sales and Orders, which relies heavily on statistical algorithms to build out suggestions for ad spends. When I'm rarely not developing, I spend my time playing Magic the Gathering and Dungeons and Dragons. @manchuck

Pursuing a Graduate Degree as Professional Development

Jack D. Polifka

Professional development is common for many developers. One form of professional development that appears to be less utilized is higher education, especially the pursuit of a graduate degree. In response to this, I want to explain several of the benefits of pursuing a graduate degree by sharing my experiences of conducting thesis research for my master's degree.



Professional development for developers can take many forms including, but not limited to, attending a conference, being part of a Meetup group[1], contributing to an open-source project, self-paced or instructor training, and pursuing certifications (e.g. Oracle MySQL 5.6 Developer certification or Zend's Certified PHP Engineer). Today, developers have an abundance of options for learning the skills they need to do their jobs. Many in the field are self-taught, and there's no shortage of "boot camps" promising to have developers career-ready in just a few short weeks.

One professional development activity that many developers seem to overlook is the pursuit of a graduate degree, master's or doctoral. According to the results of a 2016 Stack Overflow survey[2] of 40,183 non-student developers, only 19.7% have a master's degree in Computer Science (or a related field), and only 2.1% have a doctoral degree in Computer Science (or a related field). Often, these degrees involve conducting research to address a specific question. Results are then reported through various outlets such as academic conferences and journals.

Pursuing a graduate degree and conducting academic research are experiences which can significantly augment a programmer's abilities. To this point, I want to share my experiences about conducting research while pursuing a master's degree in Human-Computer Interaction (HCI) and several benefits that came along with earning the degree. For each benefit, I'll share part of my experience conducting thesis research as it applies to the benefit and why it is valuable for a developer. Afterward, I'll describe some of the factors developers should consider before pursuing a degree.

Human-Computer Interaction

First, HCI is an academic discipline that researches the design and usage of computer technology. In particular, it focuses on how people interface with computers and other related technology. As new ways to interact with computers are invented, this field looks at how to improve the quality of the interaction between people and machines by improving the usability of user interfaces. Some current research includes studying how users can more easily customize applications for their needs, new interfaces for ubiquitous embedded devices, using augmented reality to perform complicated tasks, and social computing.

Benefits

Most benefits I will describe are in the order they manifest when pursuing a graduate degree and can be obtained from any field of research (e.g. physics, psychology, mathematics, etc.). However, the last benefit, "Usability," may only be associated with HCI or related fields.

Being More Independent

Some forms of research are done in isolation from others for extended periods of time. There are moments when a researcher has the opportunity

1 *Meetup group: https://www.meetup.com*

2 *2016 Stack Overflow survey: http://phpa.me/so-survey-2016*

to work in teams or consult with fellow researchers; however, these opportunities are fewer when compared to something like a developer wanting feedback from a co-worker (often, this feedback comes from events like code reviews). Periods of working alone happen because research often cannot be broken down into smaller tasks which allow multiple people to work simultaneously. Research also requires consistency, which is facilitated by one person doing something over and over in the same way. Research doesn't follow pre-described roadmaps where each future step and task are fully detailed in advance. Rather, future steps are dependent on the results of the current step in progress.

One major benefit from working by oneself for long periods of time is a person becomes more independent. Being more independent often makes someone more confident in their own abilities, programming included, and to have less hesitation to take on challenges. Simply put, being more independent means becoming more productive because one is less skeptical of their abilities.

Being independent will help you develop some practical skills as well, beyond self-confidence. You'll have to plan and schedule your research by breaking it down into more discrete tasks and activities. Each task or activity will need to be estimated, and you may later come back and refine your estimates as new information becomes available. You'll have to prepare progress reports, secure permission to use human subjects or research facilities, and eventually, you might write and defend a thesis. These skills come in handy later if you ever need to manage a software development project, as a solo freelancer or a technical lead within a larger programming team.

Creativity

The purpose of research, in its simplest form, is to provide answers to unresolved questions. To answer

these questions, researchers often need to be innovative in the methods they use to reach conclusions. One type of innovation in research methodology is by applying a well-established research technique from one field of study to another field of study (e.g. applying a technique such as eye-tracking from psychology research to chemistry research)[3] [4]. Experimental research often requires more than one experiment to test a hypothesis. By conducting multiple experiments, a better understanding is established for a given subject. Each experiment comes from a different point of view and acts as a way to tease out different influences and correlations.

Understanding how an idea from one context can be applied to another is one dimension of being creative. Creativity, in any sense, benefits developers because it broadens the number of techniques or solutions they can use to address a given problem. With more solutions comes two things:

1. the ability to respond quickly to programming needs,
2. and the ability to provide better-designed solutions.

Being able to respond quickly to a programming need is due to the breadth of solutions available. This should not be confused with experience, which informs someone of what solution they ultimately select. The ability to provide better-designed solutions has the opportunity for less technical debt and to have code that is better optimized for future changes if needed.

Data-Driven Decision Making

As stated before, the future steps of a research project are not fully detailed. Future steps are dependent on the results of the current step in progress. To determine what the next step in a research project is, the observations collected during one phase of research are analyzed then used to inform what directions are possible or make sense next. Using data from one phase of research to inform the steps that are taken next can be called making data-driven decisions.

The ability to make data-driven decisions can be invaluable for developers when they face unfamiliar issues. Though situations like this are more likely to happen to those with entrepreneurial ventures, doing some quick statistical analysis with data or asking five users what they think about a given interface[5] can go a long way in selecting the most logical answer for a situation.

More frequently, established web applications are collecting and turning to data-driven decision making when evaluating new features, layouts, and interactions. Sometimes it's as simple as an A/B experiment to see what text results in higher sign-ups to a mailing list and even more complicated experiments are possible. Having a solid grounding in understanding how to collect and interpret the data you can collect is beneficial. You'll be able to challenge long-held assumptions about how to build a web application and defend your design decisions.

The ability to make data-driven decisions can be invaluable for developers when they face unfamiliar issues.

Iterative Process

The action of repeating a process to refine and improve the results of a given product or project is using the iterative process. An iterative process can be used to great success in research to obtain answers that better explain a given question. Suppose an experiment is repeated three times with slight modification to each based on the results of the previous experiment. In that scenario, you would have a better answer than if only one experiment was completed.

The iterative process can be used similarly when developing a new piece of functionality or interface. A developer could build an initial version of a piece of functionality or interface, share it with a few people, evaluate and incorporate the feedback, and start over. Use of the iterative process can lead a developer to build better products through incremental refinement.

Iterative design processes with short feedback loops are a cornerstone of "agile" development methodologies. Software design requires adapting to new and changing requirements which may only become apparent once you start digging into the details of any potential solutions. Understanding the right questions to ask after each iteration will keep your sprints productive.

Communication

Researchers need to be effective in their communication skills to share the results of their research. Strong speaking and writing abilities are required since academic conference presentations, and scholarly journals are often used as outlets for reporting research outcomes. One of the major benefits of research is sharing new knowledge with others, but that benefit is lost when research results are not disseminated.

Communication, as related to reporting research results, often has to be completed in ways that make it understandable to experts who are from different fields of study. This results in someone being able to customize their thoughts for different audiences. Developers who are able to effectively express

3 *Hinze, S. R., Rapp, D. N., Williamson, V. M., Shultz, M. J., Deslongchamps, G., & Williamson, K. C. (2013). Beyond ball-and-stick: Students' processing of novel STEM visualizations. Learning and instruction, 26, 12-21.*

4 *Williamson, V. M., Hegarty, M., Deslongchamps, G., Williamson III, K. C., & Shultz, M. J. (2013). Identifying student use of ball-and-stick images versus electrostatic potential map images via eye tracking. Journal of Chemical Education, 90(2), 159-164.*

5 *Nielsen, J. (March 19, 2000) Why you only need to test with 5 users. Retrieved from http://phpa.me/test-with-5-users.*

ideas allows them to be successful in a number of different scenarios including, but not limited to: company reports, emails with co-workers, professional blogging, and group presentations.

Writing and presentation skills require practice to improve. Outside of an academic setting, it's probably easier to avoid having to write or give a talk if you're introverted or are afraid of public speaking. Being forced to develop these skills early on will help you if you're ever on the spot or need to communicate with a variety of project stakeholders.

Usability

HCI and related fields focus on understanding the relationships between users and their use of computer interfaces. Usability is the level of ease of interaction with a given website or software interface. Research in HCI and related fields often focuses on developing techniques for improving usability or the evaluation of such techniques through experimental usage.

Understanding what usability is and how to test for it are very useful for developers. If your users cannot easily use something you programmed in production, their time is being wasted. Users should be spending their time on the task at hand instead of trying to learn how to use your application. Using the ideas of data-driven decisions and the iterative process along with well-established usability techniques, developers can do more to make sure their applications are user-friendly.

Usability is more than just having a nice, uncluttered user interface. Usability is concerned with understanding the tasks a user will be using an application to perform and, with that understanding, making sure completing that task is straightforward. As an example, assume you have users who will be booking a hotel room for one or more days. You can have them select the start and end dates in a number of ways, from entering a date in a text field, to picking the day, month, and year in `<select>` elements, or even using a JavaScript date picker which shows a monthly grid. Each one could do the job but may introduce errors or complications along the way. If you're taking date inputs via a text field, how are you going to validate or format user-supplied dates consistently?

Considerations Before Pursuing a Degree

While there are a number of benefits of pursuing a graduate research degree, there are some factors to consider before pursuing a degree. One of the first items to consider is the resources someone will need to invest into the degree. The two main resources people will often need to invest are money and time. The cost of a higher degree can be offset in part by teaching and research assistantships but are often reserved for full-time students (assuming you're a working professional). If someone is not a full-time student, though, it may take them some time before completing the degree. This leads to a balancing act that often happens involving these two resources.

Another item to consider is one's own interest in the degree. Several parts of a graduate degree involving research are done alone requiring individuals to have both self-drive and a high level of interest. Without both of them, someone may lose focus of what they are doing and work on other activities instead of their degree.

Lastly, the skills gained by pursing a graduate degree don't always translate to working with specific language or technology. Instead, they improve more universal skills like the ones previously described in this article. If your goal of pursuing a graduate degree is to learn a specific language or technology, there are probably better ways to learn about that subject matter. Research graduate degrees are about learning to conduct research and advancing knowledge, not simply learning about already establish content.

Conclusion

While the idea of pursuing a graduate degree as a form of professional development may not be as programming focused as other forms, it would seem to have some benefits that other methods do not. These include becoming more independent, heightened creativity, the ability to make data-driven decisions and use the iterative process, strong communication skills, and an understanding of usability and how to measure it. Because of these benefits, the pursuit of a graduate degree may be something for established programmers to consider.

Additional Reading

On the Value of a Degree…, an article in the September 2016 issue[6] by Eli White. While the aim of the article doesn't seem to talk directly about graduate degrees, it does add to the larger discussion about the value of higher education for developers. The article talks about how a college degree is a strong asset for developers but may not be an equivalent to real-world experience. Theory learned in the classroom is not the same as writing code "in the wild" where things are often more abstract. This doesn't mean college degrees are useless, though. Rather, one of their major strengths is in how they can act as a bridge for inexperienced developers to their first programming job.

Jack is a software engineer for the Graduate College at Iowa State University. His past is a mix of academics and industry with a Master of Science in Human Computer Interaction, two and a half years at Opticsplanet.com, and a Bachelor of Science in Web and Digital Media Development.

6 *September 2016 issue:* http://phpa.me/sept-2016-issue

Validate the Complexity of Your Writing With TextStatistics

Matthew Setter

While code can often be the foundation of what we do on a daily basis; it's not the only thing which is important. Well, it shouldn't be, if it is. Another, dare I say, equally important part is documentation. This can be class and function documentation, end user documentation, tutorials—even creating broader content such as screencasts and online training.

Given that, we have to regularly ask ourselves:

> *Is the documentation we're writing able to be read and comprehended by the majority of people who will read it?*

Welcome to TextStatistics

Perhaps you've not stopped to consider this previously. If not (or even if you have), in this month's edition of Education Station, I'm going to introduce you to a library which can help you gauge the complexity of your writing. If you work with a content management system, these tools can help your content authors assess the complex nature of their writing, and hopefully simplify it to reach site visitors.

The library is called TextStatistics[1], by Dave Child. The library, as the repository describes it:

> *Generates information about text including syllable counts and Flesch-Kincaid, Gunning-Fog, Coleman-Liau, SMOG and Automated Readability scores.*

These terms might not mean a lot to you. So, before we dive into the library, let's look at what they mean. Collectively, Flesch-Kincaid, the Gunning Fog Index, Coleman-Liau, and SMOG are all ways of calculating a readability score.

What is a Readability Score?

According to readable.io[2], a readability score is:

> *A computer-calculated index which can tell you roughly what level of education someone will need to be able to read a piece of text easily.*

It goes on to say that:

> *There are several algorithms available for measuring scores, and these use factors like sentence length, syllable count, percentage of multi-syllable words and so on to calculate their own readability score.*

To summarize, these algorithms are all means of estimating the complexity of a given piece of text. Once you've determined its complexity, you can then know, with a rough degree of certainty, what level of education a person will require to fully understand it. You can then more accurately write to suit your reader.

Gunning Fog

But what are these tests? Let's take a more detailed look, starting with Gunning Fog[3].

> *In linguistics, the Gunning fog index is a readability test for English writing. The index estimates the years of formal education a person needs to understand the text on the first reading. A fog index of 12 requires the reading level of a U.S. high school senior (around 18 years old).*

Coleman-Liau

Next, there's Coleman-Liau[4]:

> *The Coleman–Liau index was designed to be easily calculated mechanically from samples of hard-copy text. Unlike syllable-based readability indices, it does not require that the character content of words be analyzed, only their length in characters.*

SMOG

Then, there's SMOG[5]:

> *SMOG Readability Formula estimates the years of education a person needs to understand a piece of writing.*

1 *TextStatistics:* http://phpa.me/gh-text-statistics

2 *According to readable.io:* https://readable.io

3 *Gunning Fog:* http://phpa.me/gunning-fox-index

4 *Coleman-Liau:* http://phpa.me/coleman-liau-index

5 *SMOG:* http://phpa.me/smog-readability-formula

> *McLaughlin created this formula as an improvement over other readability formulas.*

Flesch-Kincaid

Finally, there's theFlesch-Kincaid, which we'll focus on, primarily, for the rest of the column.

> *The Flesch–Kincaid readability tests are readability tests designed to indicate how difficult a passage in English is to understand. There are two tests, the Flesch Reading Ease, and the Flesch–Kincaid Grade Level.*

In a nutshell, the higher a piece of text scores on the *Reading Ease* test, then the lower it will score on the *Grade Level* test. Said another way, the easier a piece of text is to read, then the lower the level of education is required to read it. You can see the formula for how the Reading Ease test is calculated in Figure 1.

Figure 1.

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

And in the table below, you can see how the scores marry-up with the grade levels.

Score	School Level	Notes
100.00–90.00	5th grade	Very easy to read. Easily understood by an average 11-year-old student.
90.0–80.0	6th grade	Easy to read. Conversational English for consumers.
80.0–70.0	7th grade	Fairly easy to read.
70.0–60.0	8th & 9th grade	Plain English. Easily understood by 13–15-year-old students.
60.0–50.0	10th to 12th grade	Fairly difficult to read.
50.0–30.0	College	Difficult to read.
0–30.0	College Graduate	Very difficult to read. Best understood by university graduates.

Now that we have all this, how do we go about measuring it (or any of the other tests)? Good question. That's why we have the subject of this month's edition: `TextStatistics`.

It provides methods for all of the tests above, along with a range of other methods, for judging the complexity of a piece of text in simpler ways. These include the number of words, sentences, syllables and so on. We're going to work through the tests first, and then finish up with the more simple methods of text analysis.

Installing It

It should go without saying, but we're going to install the library using Composer. To do so, run the following command from the command line, in the root directory of your project:

```
composer require davechild/textstatistics
```

This will, after a short period, import the package into the `vendor/` directory for your project (or make it the first one, if you're just experimenting with it).

Let's Write Some Code

Now we're ready to see how it works. Honestly, there's not a lot to it. To sum it up, all we need to do is pass a piece of text to one of the respective functions, and it will report back a score.

To get started, in a new file, let's be unoriginal and call it `index.php`, add the following code:

```
<?php

require_once ('vendor/autoload.php');

use DaveChild\TextStatistics as TS;
```

With that, we're ready to go, but we'll need a piece of text to analyze. To get a proper gauge, we'll need at least two. We'll take the following quote, from Iron and the Soul[6], by the legendary Henry Rollins.

> *When I was young I had no sense of myself. All I was, was a product of all the fear and humiliation I suffered. Fear of my parents. The humiliation of teachers calling me "garbage can" and telling me I'd be mowing lawns for a living. And the very real terror of my fellow students. I was threatened and beaten up for the color of my skin and my size. I was skinny and clumsy, and when others would tease me I didn't run home crying, wondering why.*

Now, let's get a piece of text to contrast it with, something which is classically considered to be more sophisticated, more complicated. The excerpt below is from Act 2, Scene 1 of The Tempest by William Shakespeare.

> *Beseech you, sir, be merry; you have cause, So have we all, of joy; for our escape Is much beyond our loss. Our hint of woe Is common; every day some sailor's wife, The masters of some merchant and the merchant Have just our theme of woe; but for the miracle, I mean our preservation, few in millions Can speak like us: then wisely, good sir, weigh Our sorrow with our comfort.*

Update the code to include variables for both the pieces of

6 *Iron and the Soul:* http://www.artofmanliness.com/trunk/?p=1748

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

text, as in Listing 1.

Then, add the code in Listing 2, to run the Flesch-Kincaid Reading Ease test on them.

This will render the following output.

```
Flesch-Kincaid Reading Ease:
  Henry Rollins: 77.8
  Shakespeare: 61
```

Given these scores, it can be reasonably expected that you'll need at least a 7th-grade education to read the text by Henry Rollins, but a 10th- 12th-grade school education to read the text from Shakespeare. Let's validate that assumption by calling another function: `fleschKincaidGradeLevel()`, to calculate the grade level.

To do so, add the code snippet from Listing 3 to `index.php`.

When you run it, it will output the following results, confirming my quick calculation:

```
Flesch-Kincaid Grade Level:
  5.5
  12
```

It shows that I was wrong. I both estimated too high for the piece by Henry Rollins and too generally for Shakespeare.

One Thing I Want to Say

I want it to be clear that I'm **not** *inferring the information contained in one piece of text is any better nor worse than the other. Specifically, I have enormous respect for Henry Rollins and have learned a great deal from him.*

What I'm seeking to convey here is that we're only considering how the information is framed and conveyed. If we want to debate the value of the information itself, we can do that another day.

Another Algorithm

Let's have a look at another test, the Spache Readability Score[7]. The score:

> *Compares words in a text to a set list of everyday words. The number of words per sentence and the percentage of unfamiliar words determine the reading age.*

Here's the formula:

```
Grade Level = (0.121 x Average sentence length)
   + (0.082 x Percentage of unique unfamiliar words)
   + 0.659
```

TextStatistics contains the method `spacheReadabilityScore`. Let's run it by adding the Listing 4 code to `index.php`.

7 *the Spache Readability Score:* http://phpa.me/spache-readability-score

Listing 1

```
$textHenryRollins = 'When I was young I had no sense of myself.
All I was, was a product of all the fear and humiliation I
suffered. Fear of my parents. The humiliation of teachers
calling me "garbage can" and telling me I\'d be mowing lawns
for a living. And the very real terror of my fellow students.
I was threatened and beaten up for the color of my skin and my
size. I was skinny and clumsy, and when others would tease me
I didn\'t run home crying, wondering why.';

$textShakespeare = 'Beseech you, sir, be merry; you have cause,
So have we all, of joy; for our escape
Is much beyond our loss. Our hint of woe
Is common; every day some sailor\'s wife,
The masters of some merchant and the merchant
Have just our theme of woe; but for the miracle,
I mean our preservation, few in millions
Can speak like us: then wisely, good sir, weigh
Our sorrow with our comfort.';
```

Listing 2

```
$txtStats = new TS\TextStatistics();

echo "Flesch-Kincaid Reading Ease: \n";
printf(
  "  Henry Rollins: %s\n",
  $txtStats->fleschKincaidReadingEase($textHenryRollins)
);
printf(
  "  Shakespeare: %s\n",
  $txtStats->fleschKincaidReadingEase($textShakespeare)
);
```

Listing 3

```
echo "Flesch-Kincaid Grade Level: \n";
printf(
  "  %s\n",
  $txtStats->fleschKincaidGradeLevel($textHenryRollins)
);
printf(
  "  %s\n",
  $txtStats->fleschKincaidGradeLevel($textShakespeare)
);
```

Listing 4

```
echo "Spache Readability Score: \n";
printf(
  "  %s\n",
  $txtStats->spacheReadabilityScore($textHenryRollins)
);
printf(
  "  %s\n",
  $txtStats->spacheReadabilityScore($textShakespeare)
);
```

Running it produces the following output:

```
Spache Readability Score:
  5
  4.1
```

Interestingly, Shakespeare has the lower grade level for this test.

And Just One More

Let's run one more algorithm, the Spache Readability Formula[8]. But, before we do, here's how it works:

1. Count the following as 'familiar' or 'known' words:

- Words appearing on the Spache Revised Word List.
- Variants of words appearing on the Spache Revised Word List that have regular verb form endings –ing, -ed, -es.
- Plural and possessive endings of nouns from the Spache Revised Word List.
- First Names
- Single letters standing alone as words. E.g., "C is the third letter of the alphabet."

2. "Difficult Words," i.e., the words not appearing on the Spache Word List are counted only once, even if they appear later with other endings. Count the following as 'unfamiliar' or 'unknown' words:

- Words not appearing on the Spache Revised Word List (other than First Names).
- Variants of words appearing on the Spache Revised Word List that have irregular verb form endings—unless those variant forms also appear on the Spache List.
- Variants of words appearing on the Spache Revised Word List that have adverbial, comparative, or superlative endings –ly, -er, -est.

With that in mind, Listing 5 shows how we run it using the library.

Listing 5

```
1. echo "Spache Difficult Word Count: \n";
2. printf(
3.   "  %s \n",
4.   $txtStats->spacheDifficultWordCount($textHenryRollins)
5. );
6. printf(
7.   "  %s \n",
8.   $txtStats->spacheDifficultWordCount($textShakespeare)
9. );
```

When run, we'll get the following output:

```
Spache Difficult Word Count:
  23
  18
```

What About Some More Basic Analysis?

Now that we've looked at a series of the core methods, which implement some of the larger algorithms, let's look at a series of the contained methods, which calculate more accurate text statistics. The available methods calculate things such as:

Calculation	Method
Letter count	`letterCount()`
Word count	`wordCount()`
Sentence count	`sentenceCount()`
Syllable count	`totalSyllables()`
Average syllables per/word	`averageSyllablesPerWord()`
Average words per/sentence	`averageWordsPerSentence()`
Text length	`textLength()`
Percentage of words with three or more syllables	`percentageWordsWithThreeSyllables()`

For these final examples, we'll just analyze the text from Henry Rollins (let's not be bashful, I'm totally biased). Copy the snippet from Listing 6 into `index.php`.

Listing 6

```
1. printf("Letter count: %d \n",
2.       $txtStats->letterCount($textHenryRollins));
3. printf("Word count: %d \n",
4.       $txtStats->wordCount($textHenryRollins));
5. printf("Sentence count: %d \n",
6.       $txtStats->sentenceCount($textHenryRollins));
7. printf("Total syllables: %d \n",
8.       $txtStats->totalSyllables($textHenryRollins));
9. printf("Average syllables per/word: %f \n",
10.      $txtStats->averageSyllablesPerWord($textHenryRollins));
11. printf("Average words per/sentence: %f \n",
12.      $txtStats->averageWordsPerSentence($textHenryRollins));
13. printf("Text length: %d \n",
14.      TS\Text::textLength($textHenryRollins));
15. printf("Words with three syllables: %f \n",
16.      $txtStats->percentageWordsWithThreeSyllables(
17.         $textHenryRollins
18.      )
19. );
```

8 *the Spache Readability Formula:* http://phpa.me/spache-readability-score

When you run it, you'll get the following output:

```
Letter count: 346
Word count: 85
Sentence count: 7
Total syllables: 121
Average syllables per/word: 1.375000
Average words per/sentence: 12.142857
Text length: 458
Words with three syllables: 3.409091
```

In Conclusion

That's been a rapid run-through of the TextStatistics package by Dave Child, along with some of the algorithms for calculating text complexity, and some background on what readability scores are and how text readability works.

Perhaps extracting statistics about text isn't your thing, or you haven't considered how useful it could be, until now. If nothing else, this month's edition has given you a greater appreciation of how you write, and how that impacts the ability of others to understand what you're trying to say.

As documentation is essential to the ability of users to use the software which we write and maintain, it's important that we consider our audience's ability to understand and process that information.

You now have a wealth of information about the theory behind it, as well as a library which you can use to monitor and improve your efforts. All the best to your efforts to write excellent documentation, to have happy users, and a massive uptake of your software projects.

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[9].

9 http://www.masterzendframework.com/welcome-from-phparch

Exploring Object Immutability

David Stockton

Continuing from the last couple of months, we're going to be looking further into objects. Objects allow us to combine state and behavior into a single construct. In objects, state is held in variables called properties while behaviors are associated functions we call methods. This month's topic is immutability which in this case is specifically about making objects with unchangeable state.

The concept of a variable, by its very nature and name, indicates something that changes or varies. So why would we want to consider making variables that cannot change, and how does that differ from constants?

If you followed the PHP-FIG when they were designing PSR-7, the topic of immutability was common, often heated, and sometimes contentious. Immutability, in the context of objects, means internal properties cannot be changed after the object is created.

Trade-offs of Immutability

Some languages provide language-enforced immutability. Swift, for example, allows the declaration of a variable using `let` which indicates during the life of that variable it will not change. In PHP, we have to work around this limitation. To implement immutability in PHP, we need to be able to control access to the properties so they cannot be changed.

We also need to have a way to set the values of the variables initially as well. This can be done through the use of a constructor, and `private` or `protected` values, see Listing 1 for an example.

Mutability Via Leaking Internal Object References

For an object with dependencies to be immutable, those dependencies must also be immutable as well. Think about it—if an object is used in a function and its dependencies play a part in the result, if the dependencies change then the result of the function could change. This is certainly not desirable in an immutable system. In Listing 2, let's see how leaking internal object references could lead to a problem.

To create our `ImmutableThing`, we need to create a `MyDependency` object and pass it in.

```
$dep = new MyDependency();
$immutable = new ImmutableThing($dep);
```

At this point, we still have a reference to `$dep`, which is the same exact object stored and used inside our object.

```
$dep->setValue('foo');
```

Listing 1

```
1. <?php
2.
3. class ImmutableThing
4. {
5.     private $a;
6.     private $b;
7.     private $c;
8.
9.     public function __construct($a, $b, $c) {
10.        $this->a = $a;
11.        $this->b = $b;
12.        $this->c = $c;
13.    }
14. }
```

Listing 2

```
1. <?php
2.
3. class ImmutableThing
4. {
5.     private $dep;
6.
7.     public function __construct(MyDependency $dep) {
8.         $this->dep = $dep;
9.     }
10. }
```

Listing 3

```php
<?php

$checkedOut = new DateTime('2017-03-01');
$dueBack = $checkedOut->add(new DateInterval('P3W'));
$now = new DateTime();
$timeOut = $now->diff($checkedOut);

if ($now < $dueBack) {
    echo 'Book has been out for ' . $timeOut->d . ' days';
} else {
    echo "Book is overdue!\n";
    echo 'Book has been out for ' . $timeOut->d . ' days';
}
```

Listing 4

```php
<?php

$checkedOut = new DateTimeImmutable('2017-03-01');
$dueBack = $checkedOut->add(new DateInterval('P3W'));
$now = new DateTimeImmutable();
$timeOut = $now->diff($checkedOut);

if ($now < $dueBack) {
    echo 'Book has been out for ' . $timeOut->d . ' days';
} else {
    echo "Book is overdue!\n";
    echo 'Book has been out for ' . $timeOut->d . ' days';
}
```

> *As of PHP 5, an object variable doesn't contain the object itself as a value anymore. It only contains an object identifier which allows object accessors to find the actual object. Objects and References*[1]

By calling the setter on `$dep`, we've changed the object that is inside of `$immutable`. If instead, `$dep` was an immutable object, then changing the value inside of `$immutable` would not be possible:

```
$dep = $dep->withNewValue('foo');
```

At this point, `$dep` would be a different object than the one in `$immutable` so its immutability is preserved.

DateTimeImmutable

PHP 5.2 introduced a new class for dealing with dates and times, the `DateTime` class[2]. It provides convenient methods for dealing with almost every aspect of these concepts. You can create new objects from many different strings that could represent a date, including things like `three weeks from now` or `last Tuesday`. You can convert from one time zone to another, format a date and time in countless ways, or determine the difference between two different `DateTime` objects in terms of years,

1 *Objects and References:* http://php.net/language.oop5.references

2 `DateTime` *class:* http://php.net/book.datetime

days, hours, minutes, and seconds. In short, it's an impressive and powerful class and a better way to work with dates than `strtotime`, `date`, and other functions. However, there are some problems with it. Suppose you'd like to determine when a library book is due back. You might write code similar to Listing 3.

On the surface, this looks simple, and possibly even correct. We have a checkout date of March 1st. The book is due back three weeks after it is checked out which is the date we store in `$dueBack`. The date right now is stored in `$now`. The default value for creating a new DateTime object is 'now'. We can then calculate how many days the book has been checked out. If $now is before the due date, we can echo how long it has been out. Otherwise, it's overdue, and we'll indicate that as well.

There's a problem, though. And, if you're not initially aware, it will probably make you want to tear your hair out. Go ahead and try it out and see if you can determine the issue. I'll wait.

The problem becomes apparent if we compare `$checkedOut` and `$dueBack`. The `DateTime` class creates mutable objects. What this means is when we were calculating the due date, we were also simultaneously updating the `$checkedOut` date to be three weeks ahead as well. In fact, if you tried a strict comparison between `$checkedOut` and `$dueDate`, you'd find they are both referring to exactly the same object. By calling `add()` on `$checkedOut`, we've mutated the state of the `DateTime` object.

Let's try again, but with a different object for representing dates and times. The `DateTimeImmutable` class was introduced to PHP in version 5.5. It will change the behavior of one of our lines of code compared to the original, but it will make all the difference in the world.

Now, what happens is this: when we use the `add` method to calculate the new due date, we get a brand new object with the values we expect. This leaves the `$checkedOut` date unaltered at March 1st, and the `$dueBack` date at March 22nd. Since I don't know when you might be running this, I cannot predict the output you'll get, but for me, the mutable version told me I had the book checked out for 27 days (but not overdue) and the immutable version says the book has been checked out for only six.

There are other issues of course, but the important ones, we've covered. In my instance, I'm running the code before March 1st, so my six days and 27 days really represent the system telling me it is six days and 27 days *until* the checkout date because I haven't bothered with checking if the flag has been set to indicate whether the difference is positive or negative. For more information on the `DateTimeImmutable` class, please check out my *Getting a Date with PHP* article in the June 2015 issue[3].

PSR-7 And Immutability

If you've built so-called value objects, you've likely either created them with all their attributes with `public` visibility (not recommended), or you've built "getters" and "setters." These allow you to mutate state by changing the properties of an object. With immutable objects, setters are not good because they allow someone to change state after the object is instantiated. However, we may want an object which is very close to one we already have but with something changed. If state can only be injected via the constructor, this can be very inconvenient if an object has many properties.

With the `DateTimeImmutable` class, using the `add`, or `sub` methods will return a new DateTimeImmutable object with a different internally represented date and time. It allows us to use a starting point, perhaps `"now"`, and build a new object based on it. Calculating the due date of the library book based on a starting date and modifying it using an interval is an example.

PSR-7[4] is about HTTP message interfaces. It defines interfaces for objects dealing with HTTP requests and responses, and almost all of it is immutable. But there are many things which need to be considered in an HTTP request or response. For a message, we have protocol versions, headers, and a body. For a server request, we have a target, a method, a request URI, server parameters, cookie parameters, query parameters, potentially uploaded files, attributes, etc. In short, there's a lot going into the object. The designers of PSR-7 decided to use something which looks similar to a setter, but instead, the method names start with "with." So you have methods like `withQueryParams` or `withCookieParams`. These methods use all the values of the object they are called on and then return a new object with only the specified changes in place. This allows you to easily get a new object with only a few things changed without needing to start from a constructor, and you can effectively "chain" these calls together allowing you to create the object you need with just a few calls.

```
$modifiedRequest = $request->withCookieParams($newCookies)
                           ->withQueryParams($newQueryParams)
                           ->withAttribute('foo', 'bar');
```

When we run the code above, starting with whatever `$request` has, we'd end up with an entirely new object in `$modifiedRequest`, but with new cookies, query parameters and a new attribute named foo. However, since `$request` is immutable, the calls above represent the creation of three new objects. But we won't have access to two of them at all.

New State Means New Object

In the example above, we are creating a new object with each call to a `with*` method. We get a new object which is different from `$request` when the `withCookieParams` call is made, another new object with the call to `withQueryParams`, and finally another new object with the call to `withAttribute`. This final object is stored in the `$modifiedRequest` variable, but what happens to the others? We have no real way of getting access

3 *June 2015 issue:* http://phpa.me/june-2015-issue

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

to them. As it turns out, PHP keeps track of how many references there are to an object and when there aren't any left, the object is marked for garbage collection.

Garbage collection is the mechanism PHP uses to reclaim memory no longer needed by your code. If it's not referenced by anything, it is no longer needed. Because of this, in most cases, there's no reason for concern over the intermediate objects which are created and discarded by following this pattern.

Let's look at how one of these "with" methods might work.

```
public function withSamoflange($samoflange) {
    $new = clone $this;
    $new->samoflange = $samoflange;
    return $new;
}
```

That's it. There's not a lot to it, but it does allow for "chaining" or fluent calling of these mutators to build a new object based on some base object plus some changes. With these methods in place, the original dependencies coming in from the constructor, and no "setters," the effect is you have an object that is effectively immutable.

Garbage Collection in PHP

After PHP detects there are no remaining references to an object or value, PHP can garbage collect, or reclaim the memory for use elsewhere. If your immutable code is built similarly to PSR-7, and you make new objects with new state often, especially when you may need to make multiple intermediate objects, there is often a concern PHP will be making dozens, hundreds, or even thousands of extra objects in order to achieve immutability. In fact, this was one of the bigger concerns brought up on the PHP-FIG mailing list in regards to PSR-7 when immutability was announced. However, PHP is pretty good with garbage collection, and these intermediate objects will be dealt with quickly.

So, even though PHP doesn't support immutability directly at the language level, and an implementation of it either means object creation only via constructor or via cloning mutators, immutability is still a good idea.

Implementing and Ensuring Immutability

We've talked already about building the object via passing in values and dependencies via constructor, and providing "cloning mutators" which provide a new object that contains most of the same state. We also talked about the importance of ensuring any dependencies are also immutable. However, there are other considerations as well.

First, we have the visibility modifiers of the dependencies. Public is clearly right out since it doesn't provide any means of ensuring a dependency cannot be changed. Protected is better, but it would still allow someone to extend our immutable object and write code that could change the dependencies. Therefore, we should make the dependencies private. Additionally, to ensure inheritance cannot be used to break immutability, consider marking immutable classes as final so they cannot be extended.

However, there's still more, even if you do all of this. An object's dependency's visibility can be modified via reflection. Objects could be serialized, changed via their string representation, and unserialized. Magic methods could be used to change dependencies. It's also possible to build an anonymous function that can access and change dependencies of immutable objects. It is possible to disable serialization and deserialization by overriding `__sleep` and `__wakeup`, but all of these additional ways of breaking immutability really require someone to go out of their way to do so. Unless, or until PHP provides language-level ways of creating immutable values, this is probably as good as we'll be able to do.

Functional Programming Roots

Much of the argument towards development with immutable objects can be traced to functional programming. Immutable objects significantly reduce the chances of errors in a multi-threaded environment. Since PHP is (generally) not threaded and typically written in a "share-nothing" way, threading issues don't come up as much as they might in other languages. The general idea with a threading issue is that if multiple threads run the same bit of code that changes the same bit of memory, the order they run can affect the results of the code. The typical example is a banking transaction and is often used to explain why database transactions are important.

```
// Thread 1 (withdraw)
$balance1 = getBalance('12345');
$newBalance = $balance1 - 100;
setBalance('12345', $newBalance);
```

```
// Thread 2 (deposit)
$balance = getBalance('12345');
$newBalance = $balance1 + 100;
setBalance('12345', $newBalance);
```

Suppose these two pieces of code are running simultaneously. With threaded code, each line of code from the top runs after the line above it, and each line from the Thread 2 listing runs in the order expected. However, these lines can be intermixed in any order. This means we could have a situation where Thread 2 runs all of its code before Thread 1 or vice versa. We also have a situation where some of Thread 1 could run, then some of 2 and then back to 1. These situations result in the final balance for account `12345` being different. Suppose both threads run the getBalance line, and they both receive a value of $100. Then, each thread figures out the new balance, that is, $0 for Thread 1 and $200 for Thread 2. The final balance at this point could be either $0 or $200 depending on which thread runs the final code last.

If Thread 1 runs all the code and then Thread 2 runs its code, we start and end with a final balance of $100.

Functional programming encourages immutable structures. Typically, variables are created immutably unless you otherwise declare them mutable. In OOP languages like PHP, the opposite is true. You have to jump through hoops to make things immutable, assuming you can do it at all.

Here's the key piece of functional programming—if you call the same function with the same arguments, you get the same result. If a function works on the state of an object and that state can change, then calling the same function with the same object (with a different state) will have a different result.

Bidirectional Cannot be Immutable

One limitation of immutable state is that it is not possible if you have bidirectional linking of objects. Since immutability requires the underlying objects be immutable, it's not possible to create a bidirectional dependency tree. The problem is this: an object could be created with a reference to another immutable object. However, once that object is created, the reference needs to be injected into an already existing immutable object. That cannot be done without changing the state of the original object which means we cannot do it.

```
$a = new Immutable();
$b = new Immutable($immutableA);
$a = $a->withReferenceTo($b);
// $a is a new object now and not the one referenced in $b.
```

Conclusion

Immutability, while it may seem weird at first, can provide for simpler code that is easier to understand. It greatly reduces issues in threaded code, which most of us dealing in PHP don't need to be concerned with, but it is something very useful and important to be aware of. It can help us make our code more functional and reduces side effects and hard-to-find bugs and issues. PSR-7 defines immutable structures for nearly every aspect of the request/response cycle and provides a great way to build reusable code for dealing with incoming or outgoing requests. It is not possible to make all code use immutable variables and objects, but I recommend you look into it and consider using it whenever you decide it makes sense for your application. It can help reduce bugs and improve understanding.

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.

Three Tech Lessons I Learned Cleaning the Kitchen

Cal Evans

We all learn life lessons away from the keyboard. Have you ever looked at them to see if they can make you a better developer? Here are three I've learned that I apply to my software development.

Loading the Dishwasher From the Back

Over Thanksgiving break (U.S.) the lovely and talented Kathy and I traveled back to my hometown, Mobile, Alabama. While at home staying with mom, I was helping to clean the kitchen after a meal. After loading the dishes into the dishwasher, I went off to do other tasks. I came back to find the lovely and talented Kathy putting more dishes in the dishwasher, but she was doing it wrong. She was loading them from the front back; **everyone** knows you load a dishwasher from the back to the front. Right? Fed up, I fell back on very old habits.

"MOM! Kathy's not doing it right!"

Amused at my story, mom explained why I was taught to load from the back of the dishwasher. When I was young we had a small portable dishwasher; if you loaded it from the front, it would tip forward.

A lot of time in tech, we do things because "that's the way we've always done them." Now, I'm not one of those that espouses challenging best practices like SOLID or DRY; some things exist because people with more experience than us have figured them out. But the next time someone tells you, "that's the way we do it here," it's a good time to challenge the idea.

Letting Things Sit

I cook at least once a week. My favorite cooking utensil is the Crock-Pot. The only problem with the Crock-Pot is after six to eight hours of cooking, things are baked onto the sides of the Crock-Pot. The only way to loosen them and clean the pot is to fill it with water and let it soak for a while. After a few hours, things are loose enough to clean it.

In tech, we hit hard problems all the time. Stubborn problems that seem almost impossible to solve. In these cases, there are two ways you can solve them. The first is to brute force it or to let it sit. I prefer to let it sit.

Many times I've found that when I can't see the solution to a problem, if I set it aside and let it rest for a while, the solution starts to appear. Let tough problems sit for a while.

Filling the Keurig

For Christmas, the lovely and talented Kathy and I got ourselves a Keurig (Hush. I know the pods aren't recyclable. I don't use them. I use the little filter baskets.) Very quickly I learned everybody loves using the Keurig; nobody likes refilling the tank. Initially, I would rant about it every time I went to make coffee. It was annoying.

Then, I realized somebody has to fill it. I stopped ranting about it and simply did something about it. Every afternoon when I clean the kitchen, I make sure it's filled for the next morning. When I make my "cup" of coffee (a 24 oz Yeti) I refill the tank so it's full for the next person. Occasionally, when I walk by and see it not full, I'll stop and fill it.

I don't keep the Keurig water tank full because I like standing at the fridge getting water and pouring it into the tank. I do it because somebody has to do it and it might as well be me. In software development, there are a lot of tasks everyone hates. Everybody's favorite whipping boy is documentation. Most developers don't like writing documentation. If you read a lot of documentation, you've probably read docs it was obvious the author did not want to write. So for your projects—work or OS—you can either keep ranting about how people need to write docs, or you can just write them. Regardless of the chore, take one of them —not all of them—on yourself to do.

Sometimes we learn lessons in life we don't bother to apply to coding. No, not every life lesson can be applied, but some can. What life lessons have you learned and applied to software development? Blog about them. Share them with the rest of us.

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

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

Upcoming Events

March

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

Midwest PHP 2017
March 17–18, Bloomington, Minnesota,
https://2017.midwestphp.org

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

PHP Experience 2017
March 27–28, Sao Paulo, Brazil
https://phpexperience2017en.imasters.com.br

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

Elephpants at PHP UK

Photo courtesy of @mbaker via twitter

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

September

PHP South Africa
September 27–29, Cape Town, South Africa
http://phpsouthafrica.com

February Happenings

PHP Releases

PHP 7.1.12:
http://php.net/archive/2017.php#id2017-02-17-1

PHP 7.0.16:
http://php.net/archive/2017.php#id2017-02-16-1

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

News

Mattias Geniar: Mitigating PHP's long standing issue with OPCache leaking sensitive data

In a new post to his site Mattias Geniar looks at an old security issue in PHP, opcache information leakage and how to mitigating the issue. He starts by talking about the vulnerability itself, that the PHP process doesn't validate the userid when fetching cached bytecode. This could result in information from other operations/scripts being exposed to other processes in a PHP-FPM pool. His solution? Upgrade PHP (the bug is fixed back in PHP 5.6.5) and set a few additional opcache ini settings to enforce the validation. Besides 5.6.29, it was also corrected in the PHP 7 releases (7.0.14 and 7.1.0). http://phpdeveloper.org/news/24949

Dev.to: PHP 7.2: The First Programming Language to Add Modern Cryptography to its Standard Library

In this post to the dev.to site Scott Arciszewski talks about a milestone in the PHP language, it being the first language to "add modern cryptography to its standard library" (PHP 7.2). He goes on to talk about what "modern cryptography" is describing concepts like secure primitives and showing example of the high-level API the integration will provide. The post finishes out with a rebuttal against some of the nay-sayers around PHP and its reputation for security. http://phpdeveloper.org/news/24904

Community News: Exakat—Static Analysis Tools for PHP

On the Exakat GitHub account Damien Seguy has put together a pretty complete list of static analyzers you can use for your PHP applications. The list is broken down into the types of scanners: Bugs finders, Coding standards, DIY, Fixers, Metrics, SaaS, Misc. Each section includes a good list of tools and links to each of them (usually just to other GitHub repositories but some go to actual project pages). http://phpdeveloper.org/news/24946

Ian Cambridge: Code Review: Single Responsibility Principle

Ian Cambridge has put together a new post for his site focusing on the Single Responsibility Principle, one of the more well-known (and well understood) parts of the SOLID design principles. He gives two examples and the code they might contain, breaking the SRP mentality. The first is a "manager" (or service) class that, while good in principle, usually ends up performing way too many operations than it should. For each he talks about the problem with the current implementation and offers a suggestion or two of things to fix to make it adhere more to SRP ideals. http://phpdeveloper.org/news/24938

Playing with RabbitMQ, PHP and node

In the latest post to his site Gonzalo Ayuso shares some of the results of his "playing with RabbitMQ, PHP and node", creating a queue system that both languages could talk to easily. He goes through some of the basics of using RabbitMQ , showing the code for each of the languages—pushing a new value into the queue, registering workers, creating Queue builders and using an exchange and receiver to process the message. http://phpdeveloper.org/news/24922

Three Devs & A Maybe: Contributing to PHP with Joe Watkins

The Three Devs and a Maybe podcast has posted their latest episode featuring returning special guest Joe Watkins to talk about contributing back to the PHP language. http://phpdeveloper.org/news/24928

Laravel News: Laravel 5.5 Will Be The Next LTS Release

According to this quick post on the Laravel News site the next version of the framework that will get long term support (LTS) will be version 5.5. Long term support means that the version will be "feature locked" on release but will continue to get bugfixes for issues found until the end of the maintenance window is reached. http://phpdeveloper.org/news/24927

ThePHP.cc: The Death Star Version Constraint

In a new post to thePHP.cc blog Sebastian Bergmann talks about Death Star Version constraint and how it could cause issues in your application if your version definitions are too loose. He gives an example where a user may have specified "*" in their `composer.json` file, leaving it wide open to get whatever the latest version is. He then talks some about semantic versioning and how it should be used in Composer configurations to ensure you're always working with the versions you're expecting. http://phpdeveloper.org/news/24910

AWS Developer Blog: Automating the Deployment of Encrypted Web Services with the AWS SDK for PHP

The Amazon Web Services blog has posted the second part of their series covering the automated deployment of encrypted web services with the AWS SDK. In this new tutorial (part two, part one is here) they continue with the deployment of services: AWS Elastic Beanstalk, Amazon Route 53 and Amazon CloudFront. The tutorial then walks you through each of the services you need to deploy and shares the code (using the AWS PHP SDK) to show how to automate the process. http://phpdeveloper.org/news/24918

Sameer Borate: New features in PHP 7.1

The PHP 7.1.x releases are some of the latest versions of the language. There's plenty of new features that came along with this new release. In this new post to his CodeDiesel blog Sameer Borate looks at some of these new features (including code snippets to illustrate). In the post he covers: void functions (return type), nullable types, symmetric array destructuring, class constant visibility. http://phpdeveloper.org/news/24897

Symfony Finland: A Practical Introduction to TypeScript for PHP developers

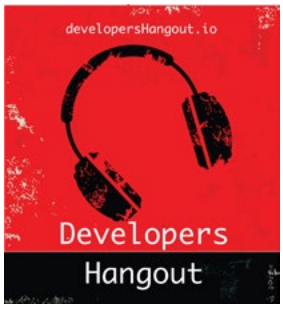
The Symfony Finland blog has posted a practical introduction to Typescript for PHP developers. TypeScript is a free and open-source programming language developed and maintained by Microsoft. It is a strict superset of JavaScript, and adds optional static typing and class-based object-oriented programming to the language. http://phpdeveloper.org/news/24878

SitePoint PHP Blog: How to Make Modern PHP More Modern? With Preprocessing!

The SitePoint PHP blog has posted another tutorial from author Christopher Pitt sharing another one of his "interesting things" you can do with PHP. In this latest article Christopher returns to the idea of "macros" to help with some pre-processing in PHP applications and, ultimately, creating a new language feature without some of the usual overhead. He starts with a brief refresher on macros to do some pre-processing on PHP scripts and allow you to make custom language features that then get interpreted into valid PHP (often with some interesting eval tricks involved). http://phpdeveloper.org/news/24872

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

php[architect] SWAG

Our CafePress store offers a variety of PHP branded shirts, gear, and gifts. Show your love for PHP today.

www.cafepress.com/phparch

ElePHPants

Laravel and PHPWomen Plush ElePHPants

Visit our ElePHPant Store where you can buy purple or red plush mascots for you or for a friend.

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

www.phparch.com/swag

Working Together

Eli White

The world is at a precarious point. There are people in need all over the world. Whether dealing with hunger, disease, discrimination, or more.

For those that don't know, I live in the United States, somewhat close to Washington, D.C. I was recently abroad to be the opening keynote for PHP UK in London, where I specifically spoke about the state of the PHP community.

In that talk, I appealed to the PHP community, those that were at the conference, and those beyond. To come together and unite, to make a welcoming community, that invites everyone to join us. To be a friendly community, that people want to be a part of. And hopefully, I showed the areas in which the PHP community is currently shining (and perhaps a few where it's having issues).

One of the points made was that for how massive PHP has become, and subsequently how many millions of PHP programmers exist in the world, we have only a small percentage of those people in the community. We perhaps even only have a small percentage of PHP programmers in the world that realize there **is** a community.

But while I was in London, something happened. I noticed that the conversations people were having, when they turned to non-technical issues were also the same. I found the TVs playing much of the same news, on the same topics. Perhaps they were from a different angle, but the issues were the same. Some because they were truly universal problems that affect everyone. Others because the actions of one country can profoundly affect others.

> *"There is no higher religion than human service. To work for the common good is the greatest creed."*
>
> *–Woodrow Wilson*

To that end, I'd like to take this moment to extend my charge about reaching out to the community. I'd like to see people reach out beyond the community and I'd like to see the PHP community give back.

– As technologists, we have a lot of power to create and help people. We can find a cause that is in need, and band together to create software that helps make the world better. As a PHP community, we can come together, to reach out towards world (or national) causes and offer our services to help make this a better place.

We can as a community have vast influence, and we can point it toward problems that sorely need our help. If everyone can give even a little bit, we'll end up in a much better place overall. If you know of a way the PHP community can help do some good, publicize it on twitter and let me know by adding @EliW[1]!

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.

1 *@EliW: https://twitter.com/EliW*

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