



php[architect]

The Magazine For PHP Professionals

The State of PHP

Finite-state Machines With PHP 8.1

Universal Vim Part Three

The Workshop:

Cheating at SPA with Breeze & Inertia

Education Station:

Making Our Own Web Server: Part 1

PHP Puzzles:

Converting Float Strings

Security Corner:

Cybersecurity Checkup

DDD Alley:

Application Event Walkthrough

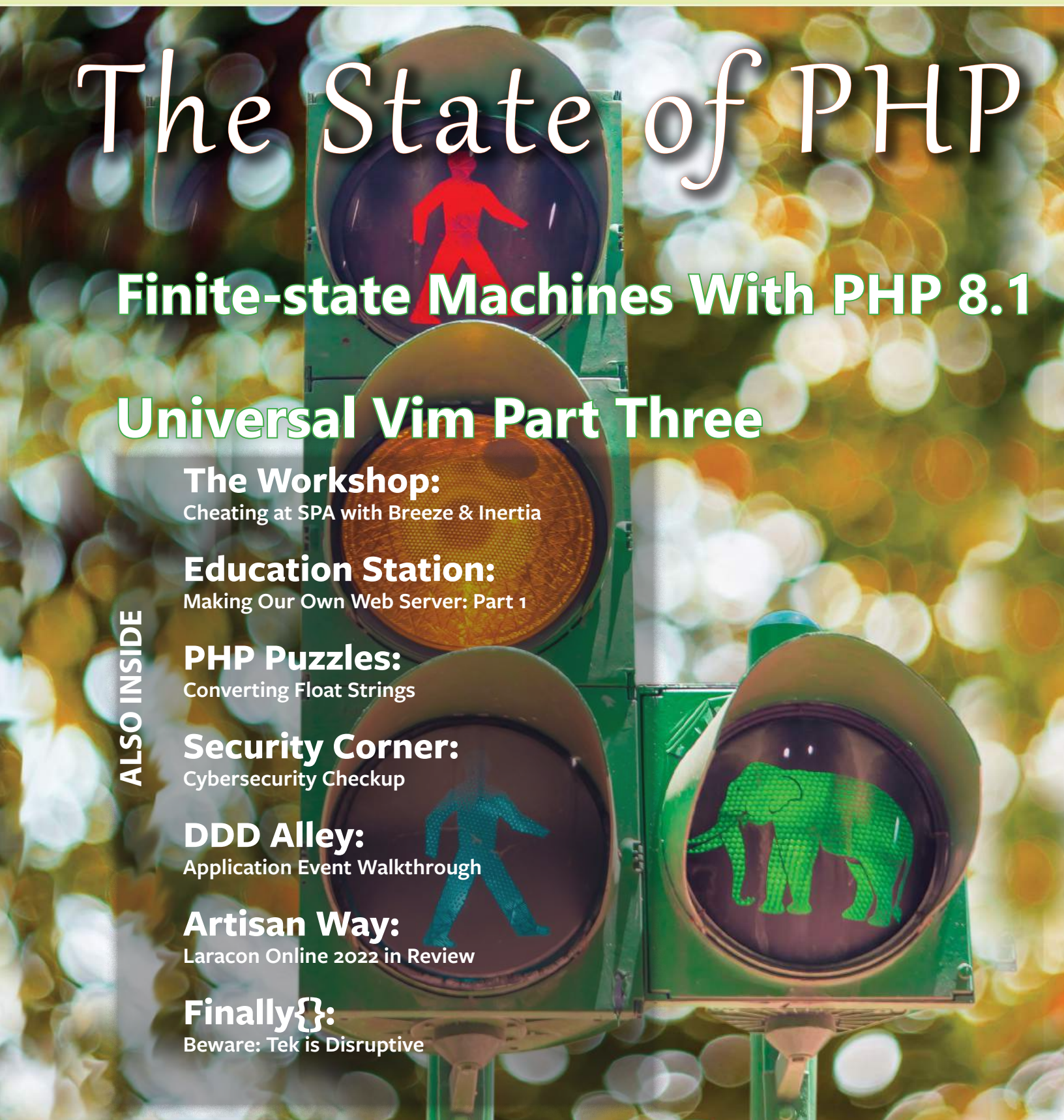
Artisan Way:

Laracon Online 2022 in Review

Finally{ }:

Beware: Tek is Disruptive

ALSO INSIDE



Optimal PHP Hosting for **Zero Downtime and Best Performance**

Multiple performance tests show Cloudways improves **loading times for websites by 200%**! With innovative features like an **optimized stack**, advanced built-in caches, CloudwaysCDN, PHP 7.3 ready servers and so much more, Cloudways enables you to build apps with **unmatched performance** and **higher conversion rates**.



Promo: **PHPARCH**
20% off for 3 months

www.cloudways.com



CONTENTS

OCTOBER 2022
Volume 21 – Issue 10



php[architect]

- 2 One Year Later**
Eric Van Johnson
- 3 Finite-state Machines with PHP 8.1**
Scott Keck-Warren
- 7 Universal Vim Part Three: Putting the You in Utility**
Andrew Woods
- 11 Cheating at SPA with Breeze & Inertia**
The Workshop
Joe Ferguson
- 17 Cybersecurity Checkup**
Security Corner
Eric Mann
- 20 New and Noteworthy**
- 21 Making Our Own Web Server: Part 1**
Education Station
Chris Tankersley
- 25 Application Event Walkthrough**
DDD Alley
Edward Barnard
- 31 Converting Float Strings**
PHP Puzzles
Oscar Merida
- 33 Laracon Online 2022**
Artisan Way
Eric Van Johnson
- 36 Beware: Tek is Disruptive finally}}**
Beth Tucker Long

Edited in a stateful way

php[architect] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

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!

Contact Information:

General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Copyright © 2022—PHP Architect, LLC
All Rights Reserved

Although all possible care has been placed in assuring the accuracy of the contents of this magazine, including all associated source code, listings and figures, the publisher assumes no responsibilities with regards of use of the information contained herein or in all associated material.

php[architect], php[a], the php[architect] logo, PHP Architect, LLC and the PHP Architect, LLC logo are trademarks of PHP Architect, LLC.

One Year Later

Eric Van Johnson

In an industry where you must keep learning, improving your skill set, and adjusting to new concepts and implementations; taking on new challenges that you are initially uncomfortable with is normal. Because of this, when John and I were presented with the option of taking over operations of php[architect], we didn't hesitate to jump at the opportunity.

It's been a year since we made that decision, and we celebrated it by completely forgetting about it. It just kind of "occurred to us" that a year had passed.

As some of you know, the php[arch] magazine was established in 2002. I've been a regular subscriber since 2003 and php[architect] has been essential to my life and career. I didn't become a full-time developer until 2009. From 2003 until 2009, I subscribed to the magazine because I was interested in PHP, and there was no better source for what was happening in the PHP world. After I went full-time as a PHP developer, I continued using php[architect] as more of a continuing education source of information. Thanks to php[architect], I managed to build my skills to a level where I could make a living and eventually enough to build a company.

Similarly, php[tek] was my first language-specific conference. I had attended many conferences, from Google I/O to Salesforce, all on the dime of the company I worked at. but I honestly couldn't express how much php[tek] changed my PHP trajectory. Hearing from other professional developers about best practices, coding patterns, and solutions to problems was invaluable. You learn about things you don't know you want to learn but are quickly happy you did. There's the ability to sit down with those same developers, as well as other developers from a spectrum of companies and industries around the world, and discuss real-world challenges and solutions they are dealing with.

The biggest thing I walked away from php[tek] with was the knowledge that I was a part of a community which became

a large part of my PHP identity and it's because of that community I was so happy when we decided to bring php[tek] back in 2023. So yeah, if you didn't know, now you do, php[tek] will be returning to Chicago, May 16th-18th, 2023. Follow the official Twitter handle @phptek¹ and the conference website at <http://tek.phparch.com>² to stay current on pricing, availability, and speakers.

Our resident YouTuber, Scott Keck-Warren, returns to print this month with his feature article *Finite-state Machines With PHP 8.1*.

Next, Andrew Woods wraps up his Vim series with *Putting the You in Utility*. In Education Station, Chris Tankersly discusses *Making Our Own Web Server: Part 1*, where he starts a journey in building a "web server" in PHP. Oscar Merida shares his solution *Converting Float Strings* in his latest PHP Puzzle Column. Eric Mann's Security Corner prepares you for your *Cybersecurity Checkup*.

DDD Alley continues its march forward on great Domain coding with *Application Event Walkthrough* and touches on the Application Event Command, Interface, Factory, and much more. This column is worth the cost of this month's subscription itself. Joe Ferguson shows you how to cheat with his The Workshop submission, *Cheating at SPA with Breeze & Inertia*. I pulled double duty this month with a rundown on *Laracon Online 2022* in Artisan Way. Beth Tucker-Long wraps up this month's issue with her piece, *Beware: Tek is Disruptive*, and looks back to the beginning of her journey. It's an inspiring read.

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 announcements. Make sure you're connected with us.

- Subscribe to our list: <https://phpa.me/sub-to-updates>
- Twitter: [@phparch](https://twitter.com/phparch)
- Facebook: <https://facebook.com/phparch>

Download the Code Archive:

https://phpa.me/October2022_code

¹ <https://twitter.com/phptek>: <https://twitter.com/phptek>

² <http://tek.phparch.com>: <http://tek.phparch.com>

Finite-state Machines with PHP 8.1

Scott Keck-Warren

Keeping track of the state of entities in our application can be a real pain sometimes. This article will show how to use Finite-state Machines to ease that pain.

As developers, we're constantly managing where entities are in some kind of flow. Entities like blog posts, multi-step user registration, and even UI elements can exist in multiple states, and we're responsible for ensuring they're always in a valid state. If something unexpected happens in those flows, it can cause bugs and ultimately us to lose clients. Thankfully our industry has decades of research and thought into the best ways to manage these flows by using Finite-state Machines.

Finite-what Nows?

A Finite-state Machine (FSM) is an abstract model that can be in exactly one of a finite (see that word popping back up again) number of states at a time. FSMs move from one state to another based on some kind of input in what's known as a transition.

When we define an FSM we do so by defining:

1. The list of states
2. The FSM's initial state
3. The transitions

When my professor first told us about FSMs, my only thought was how much work it seemed. I think most developers have a similar reaction and attempt to solve problems with more questionable methods that potentially bite them later versus using FSMs. I know I've done this before, and I've fixed systems that have had 20 variables that were replaced by a single FSM.

Once you learn about FSMs and how they're constructed, you'll see them everywhere. Stoplight? That's an FSM. Door lock? Yup! Vending machine? Absolutely!

Let's look at a stoplight FSM. In the United States, we have traffic lights that cycle through red (stop), green (go), and yellow (about to turn red so slow down) in a loop. There are also alternatives to this, like flashing red and flashing yellow, but I will ignore that complexity to save space.

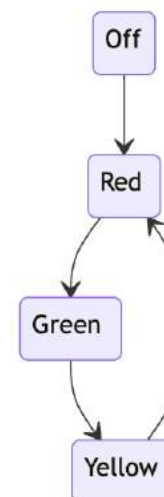
If we model that behavior into an FSM, we get:

1. List of states: Off, Red, Yellow, Green
2. Initial state: Off
3. Transitions
 1. Off to Red on boot
 2. Red to Green after X seconds
 3. Green to Yellow after Y seconds
 4. Yellow to Red after Z seconds

(See Figure 1)

In this case, the input to determine when the transitions

Figure 1



occur is based on a timer. Still, it could just as easily be a user clicking on a UI element or receiving a webhook from an external provider. You might notice that we didn't define transitions that would cause problems like Yellow to Green. That's because we never want this to happen, so we'll add code to prevent this transition from ever occurring.

The last two pieces of terminology we're going to cover are entry actions and exit actions. These are actions that are performed when the FSM enters or exits a state (respectively). Examples would include sending a notification and creating jobs to process data.

How We Can Use Finite-state Machines in Web Applications

Now that you have an overview of what an FSM machine is, let's create one in PHP.

For our example, we'll create an FSM for a blog post's state through its publishing flow. A post will start out in a "Draft" state. When the author is ready to have their editor review the article, it can be marked as "Ready for Review". The editor can either send it back to "Draft" to get updates or schedule it for being published. Then when the publish date occurs, the state automatically switches to "Published".

1. List of states: Draft, Ready for Review, Scheduled, Published

2. Initial state: Draft
3. Transitions
 1. “Draft” to “Ready for Review”
 2. “Ready for Review” to “Draft”
 3. “Ready for Review” to “Scheduled”
 4. “Scheduled” to “Published”

See Figure 2.

Figure 2



Enumerations in Php 8.1

PHP 8.1 added a bunch of new features, and Enumerations are by far my favorite. Enumerations, or enums for short, allow us to define a new structure much like a class, but that can only hold a set of allowed values. They’re a powerful structure for modeling all kinds of domain logic, including finite-state machines. We can create an enum to model our published states using the code below. The case keyword is used to distinguish the valid values for our enum. We can define each of our states for our FSM as a different case.

```
enum PublishedState {
    case Draft;
    case ReadyForReview;
    case Scheduled;
    case Published;
}
```

At a very basic level, we can assign an enum value to a property inside our class.

```
$this->state = PublishedState::Published;

var_dump($this->state);
// enum(PublishedState::Published)
```

But it’s impossible to set it to an invalid case when we define the type of the property.

```
// PHP Fatal error: Uncaught TypeError:
// Cannot assign string to property
// BlogPost::$state of type PublishedState
$this->state = "Junk value";
```

It also has the added bonus of allowing us to pass these as a parameter to a function, so we’re always sending valid data.

```
function updateState(PublishedState $newState): void
{
    // ...
}
```

Now you might be thinking, couldn’t I just use a string or an integer to keep track of my values? The short answer is that you could, but by using enums we’re given an extra level of type safety.

For example, without enums, we might use some class constants to keep track of our valid states.

```
class PublishedState {
    public const DRAFT = "Draft";
    public const READY_FOR_REVIEW = "Ready for Review";
    public const SCHEDULED = "Scheduled";
    public const PUBLISHED = "Published";
}
```

When we need to reference these states we’re using strings, so our function declaration might look like the following:

```
public function updateState(string $newState): void
{
    $this->state = $newState;
}
```

Now you might be saying no, no, no, that’s not going to work. You can pass any kind of value in for \$newState, and then you’ll be in an invalid state. And you’re exactly right that it is a problem with this simple implementation; it’s really easy to pass ANY value to this function and perform invalid transitions.

```
// invalid state
$blogPost->updateState("PHP Architect");

// invalid transition
$blogPost->updateState(PublishedState::PUBLISHED);
```

By using our enum, we automatically get two bonuses. The first is that because we have the parameter defined as a PublishedState and not a string, our IDE will give us a more helpful hint when we try to use the function. The second is that we’ll have a run-time check (and if we’re using a static code analysis tool like Psalm before then) to ensure we don’t pass an invalid enum value.

```
//result: Fatal error: Uncaught Error: Undefined constant
//      PublishedState::ReadyForReview
$post->updateState(PublishedState::ReadyForReview);
```

We'll have to solve the invalid transition part in the future because we have a couple of other topics to cover before we can solve that.

Representing States

One of the problems with creating FSMs using enums is that we need some way to represent our states outside the enumerations. Ultimately, we're going to need to store our state inside some kind of a persistence layer. Meaning we're going to need to be able to have some way to represent these states as a scalar value. There are two main ways that we can do this.

The first option is to use integers. Integers are an ideal solution because they're quick to insert, search, and update when we store them in a database. Their downside is that it's hard to quickly convert an integer into the state without doing some kind of lookup (or if you've memorized all the states in every state machine in your system).

The other option is to use strings. Strings are less than ideal from a database normalization standpoint and are slower to insert, search, and update. However, they're easier to read, so I'll use them for this article to make it easier for the reader. To apply our chosen representations to our enums, we can convert our enum to what's called a "Backed Enum" because it's "backed" up by a scalar value.

```
enum PublishedState:string {
    case Draft = "Draft";
    case ReadyForReview = "Ready For Review";
    case Scheduled = "Scheduled";
    case Published = "Published";
}
```

We can also use integer values if we're so inclined.

```
enum PublishedState:int {
    case Draft = 1;
    case ReadyForReview = 2;
    case Scheduled = 3;
    case Published = 4;
}
```

See how much easier the string version is to parse quickly.

PHP has some great logic built into its implementation to prevent us from shooting ourselves in the foot. When we're using a Backed Enum, we must create a **unique** scalar equivalent for all values. If we duplicate or skip a value, we'll get errors instead of allowing us to continue. As someone who has accidentally created duplicate public consts in the past, this is super helpful. See listing 1 and 2.

Now that we've defined our scalar equivalents, we can grab it by using the value property.

Listing 1.

```
1. // results in:
2. // PHP Fatal error: Duplicate value in enum
3. //      PublishedState for cases Draft and ReadyForReview
4. enum PublishedState:string {
5.     case Draft = "Draft";
6.     case ReadyForReview = "Draft";
7.     case Scheduled = "Scheduled";
8.     case Published = "Published";
9. }
```

Listing 2.

```
1. // results in:
2. // PHP Fatal error: Case ReadyForReview of backed enum
3. //      PublishedState must have a value
4. enum PublishedState:string {
5.     case Draft = "Draft";
6.     case ReadyForReview;
7.     case Scheduled = "Scheduled";
8.     case Published = "Published";
9. }
```

```
// this is "Scheduled"
echo PublishedState::Scheduled->value, PHP_EOL;
```

```
// also "Scheduled"
$state = PublishedState::Scheduled;
echo $state->value, PHP_EOL;
```

Scalar to Enum

When we need to get from the scalar value back to the enum, we can use the `from()` method. This method takes the string or integer value and converts it back to the enum.

```
$state = PublishedState::from("Scheduled");
var_dump($state); // enum(PublishedState::Scheduled)
```

If a value is passed that doesn't match one of the defined values, there will be an error.

```
// Fatal error: Uncaught ValueError: "junk" is not
//      a valid backing value for enum "PublishedState"
$state = PublishedState::from("junk");
```

To make this safer, PHP 8.1 gives us the `tryFrom()` function that will return null instead of throwing an error.

```
$state = PublishedState::tryFrom("junk");
var_dump($state); // null
```

A tip before you start writing a lot of code to handle these conversions, check the documentation of your database

manager to see if it will help you do some of the heavy lifting. Many Object Relational Managers have support, or are adding support, to convert the database values to enums automatically and back.

Validating Transitions

Now that we've associated values with our enum, we can tackle the issue of validating our transitions. We could keep all of this logic inside our `BlogPost` class; however, it would be better if we could locate it close to where we define our states inside of our `PublishedState` enum. Thankfully, enums can contain methods. They can also implement interfaces so we could have a dedicated `FiniteStateMachine` interface to ensure our FSMs all have a common interface. To start, we'll add a check to our `updateState()` function to make sure we have a valid transition and throw an exception if not. See listing 3.

Listing 3.

```
public function updateState(PublishedState $newState): void
{
    if (!$this->state->isValidTransition($newState)) {
        $message = "Unable to transition from ";
        $message .= "{$this->state->value} to {$newState->value}";
        throw new Exception($message);
    }

    $this->state = $newState;
}
```

Now we can add the function to `PublishedState` to support this: See listing 4.

Astute readers might have noticed we don't have `PublishedState::Scheduled` as a key in our transitions. That's because

Listing 4.

```
1. enum PublishedState:string {
2.     function isValidTransition(PublishedState $newState) {
3.         $transitions = [
4.             PublishedState::Draft->value => [
5.                 PublishedState::ReadyForReview,
6.             ],
7.             PublishedState::ReadyForReview->value => [
8.                 PublishedState::Draft,
9.                 PublishedState::Scheduled,
10.            ],
11.            PublishedState::Scheduled->value => [
12.                PublishedState::Published
13.            ],
14.        ];
15.
16.        return in_array($newState, $transitions[$this->value]);
17.    }
18. }
```

it's the terminal state and has no valid transitions. You can add it to your own implementation if you want to ensure you have that level of completeness, but I think it clutters up the function.

Finally Done!

There you go. After a little elbow grease and maybe a lot of head-scratching, we've arrived at a basic but fully functioning FSM written using native enums in PHP 8.1. Our code is fairly easy to read and maintain, and we can easily create new enums for new FSM machines without too much trouble. We might want to create a trait for our `setState()` function, but that could be overkill.

State Machines Before Php 8.1

Now if you're like me and you're still stuck maintaining projects that need to support versions of PHP before 8.1 you might be asking how you could do that. We've been doing this for years without enum support so we can do it with a little effort and a little less type safety.

To start, we can convert our enum to a class with public consts to replace the cases. See listing 5.

Then we'll update our model to use the string version. See listing 6.

For a little more type safety, we can make our `PublishedStateString` a Value Object (see our YouTube channel for more info on Value Objects). See listing 7.

Then we can use this in our `BlogPostString`.

```
function updateState(PublishedStateString $newState)
{
    // ...
}
```

When we're finally ready to support just PHP 8.1 and greater, we can easily swap in our enumeration version.

In Review

Finite state machines are a powerful tool in our programmer's toolbox. They allow us to create a set of rules for valid states an entity can exist in and for how it can move from one state to another safely. We can use PHP 8.1's enum support to create an implementation and fall back to using classes if we can't yet support only 8.1.



Scott Keck-Warren is the Directory of Technology at WeCare Connect and has been working professionally as a PHP developer for over a decade, as well as leading technical teams. Scott creates videos for the PHP Architect YouTube Channel at www.youtube.com/c/Phparch. [@scottkeckwarren](https://twitter.com/scottkeckwarren)

Universal Vim Part Three: Putting the You in Utility

Andrew Woods

We complete our quest to craft a universal vim experience. We add a few utilities to increase your speed, agility, and efficacy to be more effective in Vim.

In parts one and two, we established and improved our vimrc configuration. We built up the UI and provided general text formatting. Then we added searching and filtering capabilities with FZF and RipGrep. This month we'll complete our transformation of Vim into an IDE-like user experience. Here we take a different approach. I selected several Vim plugins to help complete *your* vimrc. There are a lot of blog posts on the web about what people consider the essential Vim plugins. I mean, *a lot!* However, you're reading this article series for its perspective. The PHP Architect audience consists primarily of developers. But I suspect quite a few of you are also content creators. The chosen plugins are good for both audiences. They'll help increase your efficacy with writing. You're going to want to write documentation!

Some of these plugins have good alternatives, while others are singular in the category. There were many obvious alternatives that I found. We'll skip the installation process this time. The focus will be on what makes these plugins great and why you should use them. You'll learn how to use them. There will be a discussion of features and trade-offs. After all—that's the stuff we love to nerd out about.

Startify

Source: <https://github.com/mhinz/vim-startify>¹

There are so many plugins we could discuss; where do we start? Begin at the beginning. When you start Vim with no arguments, it doesn't show you much. Other IDEs, like PhpStorm, help you get back to where you've been. They re-open all the files and panels you had open during your previous session. Vim is not so helpful out of the box. Vim is designed to be built up into the editor you want it to be. Its features are not easily discoverable. The `:mksession` command is a good example of this. You could use `:mksession`, but you have to know that it exists. I suspect many developers don't know it's a vim feature.

Startify takes a different approach by not re-opening the files automatically, providing a great solution to fill the gap. Out of the box, Startify shows you a “menu”, which is basically a few sections, each containing a list of items. The first section is your recently changed files in or below the current

directory. Startify lists the recently edited files in other directories in a separate section.

The moniker MRU in these two section headings stands for **Most Recently Used**. The best part of Startify has to be the thoughtful quote from a cow. The sections and items shown by Startify are customizable. For example, some people run `git status` before opening Vim to see what they need to finish today. You could add that output under the cow's thoughts. You can call the `:startify` function ad hoc. Some people set up their configuration to display it when they open a new empty buffer. That feels a little excessive, but maybe it isn't. Try mapping it to the leader key so you can call it at will. Startify lists the items in each section—beginning with a box containing a single character. You can type that single character, move the cursor over it, and press enter. Both are pretty natural to execute. What isn't obvious is that there are multiple ways to open the desired item. Each item can be opened in a horizontal split, vertical split, or a tab, by pressing the `s`, `v`, or `t` keys, respectively. You can also select multiple items to open; all the items will be opened when you press enter.

Startify handles sessions a little differently. It keeps all the session information in your `~/vim/sessions` folder. Neovim uses `$XDG_DATA_HOME/nvim/session`. You can change this with `g:startify_session_dir`. This makes it easy to see all your sessions. Startify additionally provides a set of session-related functions.

<code>:SLoad</code>	load a session
<code>:SSave</code>	save a session
<code>:SDelete</code>	delete a session
<code>:SClose</code>	close current session

Startify is one of those plugins without an alternative—at least that I could find. This may be partly because there isn't a good name for this category of plugin. Searching the web for “IDE file management” is likely to turn up something like NerdTree, which isn't what we want. I've touched on the highlights of Startify, but there is more to discover. Read through the `:help startify` documentation to learn more about how to configure it. Hopefully, you'll find some inspiration on how to adapt Startify to your needs.

¹ <https://github.com/mhinz/vim-startify>:
<https://github.com/mhinz/vim-startify>

Undo Tree

Source: <https://github.com/mbbill/undotree>²

Everyone makes mistakes. Sometimes we wish we could return to the way things used to be. How can we see an earlier state of our files or recover some content? Sure we have Git, but it only knows about committed file changes. It doesn't capture everything. What about the changes between commits? PhpStorm has the "Local History" feature. What about Vim? Vim actually records all your changes. It just doesn't have a good way to expose it to the user. The Undotree plugin solves this issue in Vim. Undotree shows your changes as a tree, empowering you to scroll back through your history.

The `:UndotreeToggle` provides access to your history of changes and a diff of each change. Its documentation shows you how to map it to a function key. However, some keyboards don't make the function keys easy to use. One of the several reasons why using your leader key is the better option. Undotree supports branches of changes. You may be wondering, *how does one make branches of changes?* Let's say you've been writing for a while, then you hit `u` in normal mode to undo your latest change, hit it once more, and a third time. The next change you add will be to create a new main branch. Those three changes you removed are now an alternative branch. With undotree, you could navigate back to the old branch to recover some content.

There's a lot to learn about undotree. If there's one thing to remember, `:help undotree-contents` is it. From there, you can find docs on a plethora of information. Undotree uses several variables to customize itself. One variable of interest is `g:undotree_WindowLayout`. It controls the position of the tree and the diff output. Layout one, the default, will show the tree on the left side of your screen, thus shifting your content to the right. I recommend layout four, which puts the tree on the right and diff output on the bottom. This prevents your content from shifting horizontally when undotree opens and closes. In part one, you'll remember we used a variable to determine the width of Vim's native file explorer, `netrw`, when triggered `:Lexlore!`. Undotree provides variables to control its size. For example, using a value of 30 means 30% of the window. This feels like a comfortable size. Fortunately, `g:undotree_SplitWidth` uses 30 as its default value. One more thing, when you toggle open the undotree panel, it doesn't get focused by default. It feels weird that it's off by default, but it's not a big deal, just something you be aware of. However, there is a variable you can use. Add the `g:undotree_SetFocusWhenToggle` to your `vimrc`, and set it to 1. Problem solved.

```
g:undotree_WindowLayout = 4
g:undotree_SplitWidth = 30
g:undotree_SetFocusWhenToggle = 1
```

² <https://github.com/mbbill/undotree>
<https://github.com/mbbill/undotree>

Alternatives

This isn't a big category; although, it'd be nice to see a couple more projects appear. What kind of innovation could move this category forward? Perhaps some FZF style fuzzy searching through your change history. Some categories are going to be more limited than others.

Mundo

Source: <https://simnalamburt.github.io/vim-mundo/>³

By a wide margin, undotree is the most well-known plugin in this category. It's not the only one, though. There was a project called Gundo. But it seems their development has slowed to a crawl. Eventually, the Mundo team forked the Gundo project and ran with it. The Mundo UI looks a bit nicer than Undotree's; however, I suspect the consistency of Undotree's development pace is keeping it in the lead.

Vim Surround

Source: <https://github.com/tpope/vim-surround>⁴

Whether you're writing prose or code, you need to use quotes. At first glance, you may think it is unnecessary to edit quotes. However, Tim Pope's Surround plugin makes it easy. Whether you want to add, change, or remove quotes from a string, it's easy. PhpStorm users have the "Replace Quotes" functionality. Surround goes beyond the use of just quotes; it toggles between single and double quotes. It also works with parentheses, brackets, and braces.

Surround also supports with HTML/XML tags. Vim understands a tag as a text object, so it's usable in combination with motion commands. This makes for some powerful results. Surround is useful when writing an article or making documentation. You can wrap your article heading with `<h1>` tags, or paragraphs with `<p>` tags. It works great with simple tags. For larger markup structures, I'd probably go a different route.

Sometimes it's easier to show than tell. Let's show a quick example of how to use Surround, starting with the following sentence.

Eric and John bought PHP Architect from Oscar.

We want to put single quotes around PHP Architect. Put your cursor anywhere on the word PHP and type `ys2aw'`. You should now see:

Eric and John bought 'PHP Architect' from Oscar.

Let's break down `ys2aw'` into its parts—`ys 2aw '`. The `ys` part is essentially "add" here. The Surround documentation says

³ <https://simnalamburt.github.io/vim-mundo/>
<https://simnalamburt.github.io/vim-mundo/>

⁴ <https://github.com/tpope/vim-surround>
<https://github.com/tpope/vim-surround>

the mnemonic for `ys` is *you surround*. The `2aw` part is a vim motion command that means *two around word*. The final character `'` (single quote) is the character we want to surround our text with.

Now that we have quotes surrounding our title, we might decide that double quotes would be better. Put your cursor anywhere on *PHP Architect*, and in normal mode, type `cs''`. You should now see *PHP Architect* enclosed in double quotes.

Eric and John bought "PHP Architect" from Oscar.

The `cs` in the expression `cs''` means “change surrounding”. The remaining two characters represent the *from* and *to* characters. We could have used `*` to change the single quotes to asterisks, which is useful for making a phrase bold in markdown. Using `cs'`, would render (*PHP Architect*) in our sentence above.

To remove the quotes, type `ds`. You should now see:

Eric and John bought PHP Architect from Oscar.

There are times when you don't try to figure out the correct motion command to use. Can you just select the text you want to use? Yes. Surround supports using visual selection as a way to add quotes around something.

Let's start our selection by putting the cursor anywhere on *PHP* and press `viw` to select the entire word. Now press `e` to bring the selection to the end of “*Architect*”. Now that the entire phrase “*PHP Architect*” is selected, type `S'`. My mnemonic for this is “Surround selection with character”. In our case, that character is single quote.

What about using it with HTML and XML tags you were talking about before? I'll leave it to you to read the Surround documentation on how to surround content with tags. It's fine for single tags—like a heading. Emmet is probably a better option for more complex HTML or XML.

Ultisnips

Source: <https://github.com/sirver/UltiSnips>⁵

I have a rule: If you have to do something more than a few times, the computer should do it. Granted, this isn't going to get you out of doing your household chores. I use programs and scripts to automate the complex and the menial. Humans should focus on creativity and problem-solving, the things that provide value. Intellisense tools are a boon for developers. They can help you with getting your structure correct. We write functions and methods to help with larger blocks. However, a function does not serve you as well with smaller blocks. There are also some situations where code isn't a good solution—like writing an email or content for a paper. This is where UltiSnips can shine.

UltiSnips is a text expander for Vim. PhpStorm calls them Live Templates. With a couple of keystrokes, you can trigger a snippet. In your snippets, you can add placeholders that allow you to inject values. Granted, a text expander or a snippet manager is not a new idea. But it's sometimes overlooked in the Vim world. Vim's internal abbreviations will carry you far. When you start doing multi-line expansions, you're pushing the limits of Vim's abbreviations. If you're futzing around with newline and tab characters—that's when you need a text expander.

If you're a mac user, you likely have Python 2.x installed. That's not gonna be enough, as UltiSnips requires Python 3. So you'll need to make sure that Python is installed and in your PATH. Also, check `vim --version` to make sure `+python3` is part of vim. Once I got Vim to recognize I had Python 3 installed, UltiSnips worked great. UltiSnips looks for folders with snippets in Vim's runtimepath. That's a variable in vim—`rtp`—containing a list of directories. Likely, your `~/.vim` directory is first in the list. So if you create your directories there, you'll be fine. You can create multiple directories to hold your snippets, which is great for organization. You just need to tell Ultisnips about them in your `vimrc` file. One obvious division is between personal and work. If you're also a podcaster, that will warrant another snippets folder. UltiSnips likes to organize snippets by file type. But what if you have some snippets that transcend file types? UltiSnips has you covered—put them in a file called `all.snippets`.

Alternatives to Ultisnips

Another long-running project is SnipMate, the only real alternative to UltiSnips. Both SnipMate and UltiSnips have been around for ten years. Vim does not have any other snippet manager plugins. In some circumstances, you might be content with a mix of Emmet and Vim's native abbreviations.

Snipmate

Source: <https://github.com/garbas/vim-snipmate>⁶

With Snipmate, you'll find some similarities to UltiSnips. This is a text expander inspired by TextMate. If you don't remember TextMate, it was a text editor by web designers and developers in its day, circa 2010. It was the SublimeText of its day, and its snippet manager was one of its key features at the time. So it makes sense someone would want to port it to Vim. The install process was a little overly complex. I attempted to install it but was never successful; thus, I have not been able to evaluate it. In addition, SnipMate uses Vundle instead of Vim Plug, which all other plugins used for this series. I didn't really want to add another plugin manager to the mix. You may have a better experience. So if you are looking to learn something new, this may be for you.

⁵ <https://github.com/sirver/UltiSnips>
<https://github.com/sirver/UltiSnips>

⁶ <https://github.com/garbas/vim-snipmate>
<https://github.com/garbas/vim-snipmate>

Lightline

Source: <https://github.com/itchyny/lightline.vim>⁷

In part one, we created a status bar to provide more information about your file. This works pretty well when you don't have any plugins. It's much better than what Vim provides by default, which is basically nothing. Now that we've been installing plugins, using a status bar plugin is a good way to spice up Vim a little bit. People really enjoy status bar plugins. They provide fancy formatting, color-coded modes, and include git branch information.

I've chosen Lightline. It's a younger project than some of the other options, but it has a good philosophy. Lightline strives to be more minimalist and configurable than its fellow status bar plugins. If you use a popular theme, there may be an existing matching lightline color scheme. Here's how you can choose your color scheme for lightline.

```
let g:lightline = {
\\    'colorscheme': 'default',
\\ }
```

One interesting thing about Lightline is that all the settings are kept in the `g:lightline` variable. The lightline developer can just add new keys and values to the object as needed. It keeps things tidy. More plugins should take this approach.

Honorable Mentions

There are several other plugins that may interest you. These are the ones that didn't quite make it. Not because they aren't worthy but because time and (page) space are limited. Let's start with Multiple Cursors. Vim provides block editing and macros, so I've never given this much thought in the past. But some of the videos I've seen have my interest piqued. Developers that love SublimeText often say multiple cursors is one of the reasons why they use it. With this plugin enabling multiple cursors in Vim, maybe you can sway them to the light side of The Force. Color codes in CSS are not the easiest to understand just by looking at them. When you write a hex code like `#f7ff00`, what human color is that? What does that look like? The Vim CSS Color plugin will render the color that the value represents. PhpStorm has a similar feature, but displays the color as a colored square in the gutter. This kind of visual feedback makes it easy to understand color codes. This also comes in handy if you want to create a vim color scheme. Learning to make a Vim color scheme is an endeavor every Vim user should try. When you do, you know you'll have at least one tool to help you.

⁷ <https://github.com/itchyny/lightline.vim>:
<https://github.com/itchyny/lightline.vim>

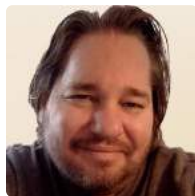
Source:

- Vim Multiple Cursors <https://github.com/terryma/vim-multiple-cursors>⁸
- Vim CSS Color <https://github.com/ap/vim-css-color>⁹
- Vim Pandoc <https://github.com/vim-pandoc/vim-pandoc>¹⁰

Wrap Up

You may be wondering, *Why didn't you talk about NerdTree?* In part, because everyone already does. I wanted to provide a different approach—one where you can maximize your flow and stay focused on your work. NerdTree takes you away from content. The other part is that NerdTree feels a lot less useful when you make the most of FZF's capabilities. FZF navigates through your project files faster. For the few times I actually need to use a file explorer, Vim's built `netrw` is good enough.

We covered a lot in this article and this series. It's true what they say—the best way to learn is to teach someone else. I learned a lot, and hope that you did too. I set out to provide a foundation for any vim user to build upon. The creation of a dotfiles project usually begins with shell configuration or Vim. I encourage you to build upon what we've covered here. Remember, this is not the end. This is not the beginning of the end. This is the end of the beginning.



Andrew Woods is a Software Developer at Paramount. He's been developing for the web since 1999. When not coding or playing guitar, he enjoys films, music, and exploring the city. You can find him at andrewwoods.net, his online home, or on Twitter [@awoods](https://twitter.com/awoods).

Related Reading

- *Universal Vim Part 1: No Plugins Required* by Andrew Woods, August 2022.
<https://phpa.me/vim-aug-2022>
- *Universal Vim Part Two: Fuzzy Search Fun* by Andrew Woods, September 2022.
<https://phpa.me/vim-sep-2022>
- *Power Up with Git* by Andrew Woods, December 2020.
<https://phpa.me/git-dec-2020>

⁸ <https://github.com/terryma/vim-multiple-cursors>:
<https://github.com/terryma/vim-multiple-cursors>

⁹ <https://github.com/ap/vim-css-color>:
<https://github.com/ap/vim-css-color>

¹⁰ <https://github.com/vim-pandoc/vim-pandoc>:
<https://github.com/vim-pandoc/vim-pandoc>



Cheating at SPA with Breeze & Inertia

Joe Ferguson

This month we're diving into a fresh Laravel application using the Breeze package to scaffold our authentication using Inertia and Vue.js. No previous Vue experience is required! We're exploring the ability to quickly build modern single-page applications with Inertia leveraging Vue.js components to build our application. If you previously used Laravel's `make:auth` commands, you'll find Breeze to be an updated and modern implementation of user registration, password reset, and functionality using Vue.js and Inertia by default. React is also supported if you would rather use it over Vue.

Inertia¹ is a JavaScript library that allows us to focus on building server-side rendered pages using JavaScript components instead of traditional views. This brings our application the power of a client-side app and the single-page app (SPA) experience without building a dedicated API. Inertia isn't a framework; rather, a client routing library that uses XHR requests and allows page visits to happen without a full page reload.

It does this without starting a framework war between Vue.js² and React; we're using Vue in our examples today just as an arbitrary example. You should use whichever framework better suits your needs, and I highly recommend using Laravel Breeze to explore the differences between the two frameworks accomplishing the same authentication features. Vue.js is just as popular these days, and often when you inquire about the difference between frameworks, you hear more noise than productive feedback. From my perspective as a grizzled server-side veteran, I use what seems to have the least amount of friction to accomplish the goals. Focus on solving your problem and not worrying about what rank your choices are on the popularity charts.

Let's dive right into the deep end, creating a brand new Laravel application, and installing Breeze³.

```
$ composer create-project laravel/laravel inertia
$ cd inertia/
$ composer require laravel/breeze --dev
$ php artisan migrate
$ php artisan breeze:install vue
$ npm run dev
```

Loading our application and clicking Register in the top right menu takes us to our registration form, where we can fill out and create our user: See Figure 1.

(Registration form provided by Laravel Breeze.)

Listing 1.

```
1. <?php
2.
3. use Illuminate\Foundation\Application;
4. use Illuminate\Support\Facades\Route;
5. use Inertia\Inertia;
6.
7. Route::get('/', function () {
8.     return Inertia::render('Welcome', [
9.         'canLogin' => Route::has('login'),
10.        'canRegister' => Route::has('register'),
11.        'laravelVersion' => Application::VERSION,
12.        'phpVersion' => PHP_VERSION,
13.    ]);
14. });
15.
16. Route::get('/dashboard', function () {
17.     return Inertia::render('Dashboard');
18. }->middleware(['auth', 'verified'])->name('dashboard');
19.
20. require __DIR__.'/auth.php';
```

Figure 1: (Registration form provided by Laravel Breeze.)

1 Inertia: <https://inertiajs.com>

2 Vue.js: <https://vuejs.org>

3 Breeze: <https://phpa.me/laravel-starterkitbreezinertia>



Where did all of this code come from? We can start by opening our routes and see that we're returning Inertia objects: See listing 1.

Our default route, which typically would render a blade template, is instead using `Inertia::render` to render a Vue.js template. We can also see from our routes the `/dashboard` route is being served by `resources/js/Pages/Dashboard.vue`. Listing 2 has been modified to fit this magazine, see `listing2-full.txt` in the code archive for this issue.

Listing 2.

```
1. <script setup>
2. import AuthenticatedLayout
   from '@Layouts/AuthenticatedLayout.vue';
3. import { Head } from '@inertiajs/inertia-vue3';
4. </script>
5.
6. <template>
7.   <Head title="Dashboard" />
8.
9.   <AuthenticatedLayout>
10.    <template #header>
11.      <h2 class="...">
12.        Dashboard
13.      </h2>
14.    </template>
15.
16.    <div class="py-12">
17.      <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
18.        <div class="...">
19.          <div class="...">
20.            You're logged in!
21.          </div>
22.        </div>
23.      </div>
24.    </div>
25.  </AuthenticatedLayout>
26. </template>
```

Note we're importing `AuthenticatedLayout`, which is for users who have been authenticated. For unauthenticated users, we can see that Breeze provides us with `resources/js/Layouts/GuestLayout.vue`, which shows much less information to the user. It will be important to remember to use the correct layout for the components otherwise you'll end up exposing sensitive data or not giving users enough access. Listing 3 has been modified to fit this magazine, see `listing3-full.txt` in the code archive for this issue. Figure 2 shows the Components and Layouts structure.

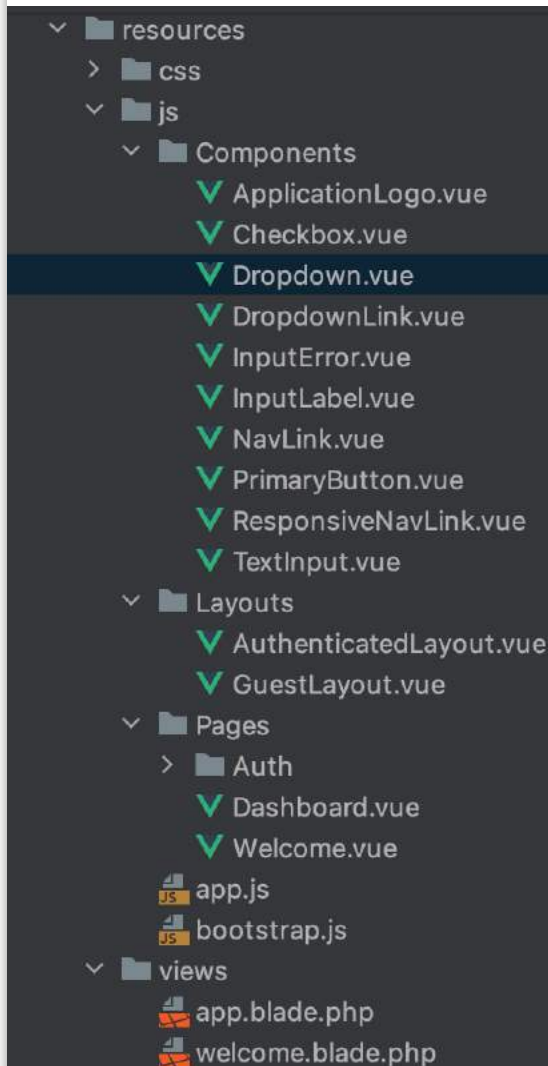
Directory layout showing our view layouts and components

Don't be too alarmed by the number of vue files; there are two primary files we're going to be interacting with to build out our Tasks feature. Listing 4 has been modified to fit this magazine, for the full contents please see `listing4-full.txt` in the code archive for this issue.

Listing 3.

```
1. <script setup>
2. import ApplicationLogo
   from '@Components/ApplicationLogo.vue';
3. import { Link } from '@inertiajs/inertia-vue3';
4. </script>
5.
6. <template>
7.   <div class="...">
8.     <div>
9.       <Link href="/">
10.        <ApplicationLogo class="..." />
11.      </Link>
12.    </div>
13.
14.    <div class="...">
15.      <slot />
16.    </div>
17.  </div>
18. </template>
```

Figure 2.





Listing 4.

```

1. <script setup>
2. ...
3. defineProps(['tasks']);
4. const form = useForm({
5.   name: '',
6. });
7. </script>
8.
9. <template>
10. <Head title="Tasks" />
11.
12. <AuthenticatedLayout>
13.   <div class="max-w-2xl mx-auto p-4 sm:p-6 lg:p-8">
14. ...
15.   <div class="mt-6 ... divide-y">
16.     <Task
17.       v-for="task in tasks"
18.       :key="task.id"
19.       :task="task"
20.     />
21.   </div>
22. </div>
23. </AuthenticatedLayout>
24. </template>

```

Our `resources/js/Pages/Tasks/Index.vue` page is where we display our form to add new Tasks as well as display all tasks in the database in a list below. The Task Vue Component is where we build out the styling and functionality of each task in our list. Listing 5 has been modified to fit this magazine, for the full contents please see `listing5-full.txt` in the code archive for this issue.

If you've never touched Vue.js, you can probably already tell a bit about what's going on. We can see there's a setup section that looks like JavaScript and a ton of HTML below it. Think of the top setup section as the business JavaScript logic, and below, the template or view that displays our information. The power of Inertia is we're passing data from our controller to our frontend via `const props = defineProps(['task']);`—each time we loop over a task, we output the template based on that specific task while Inertia does all the heavy lifting for us.

We use checks such as `<Dropdown v-if="task.user.id === $page.props.auth.user.id">` to ensure we're only showing the Edit and Delete dropdown menus if the task is owned by that user. If you created another user in the same application, you'd notice they won't be able to edit your tasks, and you will not be able to edit theirs.

Inspecting the `resources` folder in our project will show us the `/js/Components` and `/js/Pages`, which currently make up our application. Because we're using Vue.js, we have these Vue templates created for us automatically. The major change for most developers is moving from blade template view files to Vue.js Pages and Components. If you're not quite ready to dive into Vue.js, use `php artisan breeze:install` to use blade templates instead of Vue.

Listing 5.

```

1. <script setup>
2. ...
3. dayjs.extend(relativeTime);
4.
5. const props = defineProps(['task']);
6.
7. const form = useForm({
8.   name: props.task.name,
9. });
10.
11. const editing = ref(false);
12. </script>
13.
14. <template>
15.   <div class="p-6 flex space-x-2">
16.     <div class=""></div>
17.     <div class="flex-1">
18.       <div class="flex justify-between items-center">
19.         <Dropdown v-if="task.user.id === $page.props.auth.user.id">
20.           <template #trigger>
21.             <button>
22.               </button>
23.             </template>
24.           </Dropdown>
25.         </div>
26.         <form v-if="editing"
27.           @submit.prevent="form.put(route('tasks.update', task.id),
28.             { onSuccess: editing = false })">
29.           ...
30.         </form>
31.         <p v-else class="...">{{ task.name }}</p>
32.       </div>
33.     </div>
34.   </template>

```

Next up, we'll create a Task model to store tasks for our user. We can leverage the `make:model` artisan command with flags to also create a migration and a resource controller: `php artisan make:model --migration --resource --controller Task`. We use `--resource` to have artisan create our index, create, store, and other resource controller methods saving us the time of having to copy and paste the boilerplate. Our Task model is located at `app/Models/Task.php`, migration for our tasks table `database/migrations/2022_08_20_201925_create_tasks_table.php`, and finally, our Task controller can be found at `app/Http/Controllers/TaskController.php`. We need to manually add our `Route::resource` configuration in `routes/web.php`.

```

use App\Http\Controllers\TaskController;
# ...
Route::resource('tasks', TaskController::class)
    ->only(['index', 'store', 'update', 'destroy'])
    ->middleware(['auth', 'verified']);

```

Our route resource will be at `/tasks` and utilize a TaskController with index, store, update, and destroy methods while applying the auth and verified middleware. The middleware



requires users to be authenticated and verified by their email addresses before they can access Tasks. We can use artisan to verify our tasks routes have been added. We expect to see index, store, update, and destroy routes based on the array we passed into `only()` in our `Route::resource`.

```
$ php artisan route:list | grep tasks
GET tasks ..... tasks.index > TaskController@index
POST tasks .....tasks.store > TaskController@store
PUT tasks/{task} tasks.update > TaskController@update
DEL tasks/{task} tasks.destroy > TaskController@destroy
```

We'll keep our Tasks simple and take a string name and boolean complete field while connecting to a user via a foreign key. Our migration's up method might look like:

```
Schema::create('tasks', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')
        ->constrained()->cascadeOnDelete();
    $table->string('name');
    $table->timestamps();
});
```

Our Task model is kept basic and utilizes fillable to allow specific fields to be assigned. We'll also be able to access the user's information from the Task object with a `belongsTo` relationship. See listing 6.

Listing 6.

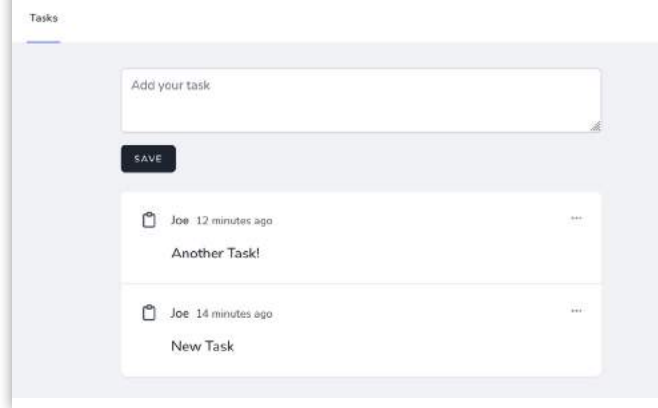
```
1. <?php
2.
3. namespace App\Models;
4.
5. use Illuminate\Database\Eloquent\Factories\HasFactory;
6. use Illuminate\Database\Eloquent\Model;
7.
8. class Task extends Model
9. {
10.     use HasFactory;
11.
12.     protected $fillable = [
13.         'name',
14.     ];
15.
16.     public function user()
17.     {
18.         return $this->belongsTo(User::class);
19.     }
20. }
```

We can also define the tasks that belong to a user by adding a method named `tasks()` to our User model, which returns a `hasMany()` such as `$this->hasMany(Task::class)`. This gives allows us to count tasks assigned to a user; for example, `count($user->tasks);` would return the number assigned to our `$user`. At this point in our application, we can add new tasks that will be persisted to the database without having to

refresh the page or navigate back and forth between creating and showing routes.

Storing our tasks in the database will be handled by our store method in our Task controller. See figure 3 and listing 7.

Figure 3.



Listing 7.

```
1. public function store(Request $request)
2. {
3.     $validated = $request->validate([
4.         'name' => 'required|string|max:255',
5.     ]);
6.
7.     $request->user()->tasks()->create($validated);
8.
9.     return redirect(route('tasks.index'));
10. }
```

Adding tasks to our list without refreshing the page.

Currently, our application allows users to add tasks, but before we allow tasks to be edited, we want to add a policy that only allows a task's user to edit it. We can create our new policy via `php artisan make:policy TaskPolicy --model=Task`. The file will be created at `app/Policies/TaskPolicy.php`, and we want to adjust the update and delete methods to specify who is allowed to update and delete tasks. Note that because we want to restrict updating and deleting to the task owner, we do not need to duplicate `$task->user()->is($user)`. Instead, we can directly call the method via `return $this->update($user, $task);` keeping our code clean and our policy class easy to understand and follow. See listing 8.

We need to build our Task controller's update method to validate our input and save our changes. Our method will return a redirect, and Inertia handles this and refreshes the Task with our changes. We can add our delete method while we're here; both use the `authorize` method call that verifies the update operation and can be performed by our `TaskPolicy`. See listing 9.

When we delete a task, Inertia removes it from the list because it handles the redirect and knows to refresh the `tasks.index` route, which returns an updated list of Tasks,



Listing 8.

```

1. /**
2.  * Determine whether the user can update the model.
3.  *
4.  * @param \\App\\Models\\User $user
5.  * @param \\App\\Models\\Task $task
6.  * @return \\Illuminate\\Auth\\Access\\Response|bool
7.  */
8. public function update(User $user, Task $task)
9. {
10.     return $task->user()->is($user);
11. }
12.
13. /**
14.  * Determine whether the user can delete the model.
15.  *
16.  * @param \\App\\Models\\User $user
17.  * @param \\App\\Models\\Task $task
18.  * @return \\Illuminate\\Auth\\Access\\Response|bool
19.  */
20. public function delete(User $user, Task $task)
21. {
22.     return $this->update($user, $task);
23. }

```

Listing 9.

```

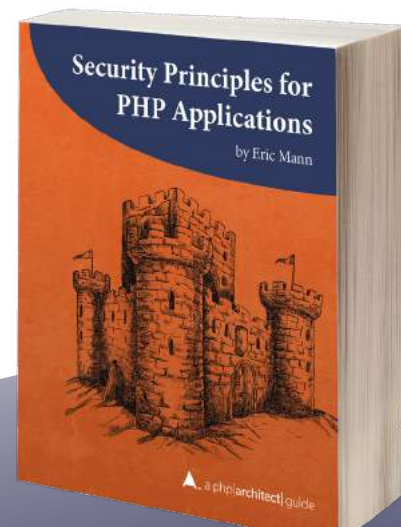
1. public function update(Request $request, Task $task)
2. {
3.     $this->authorize('update', $task);
4.
5.     $validated = $request->validate([
6.         'name' => 'required|string|max:255',
7.     ]);
8.     $task->update($validated);
9.     return redirect(route('tasks.index'));
10. }
11.
12. public function destroy(Task $task)
13. {
14.     $this->authorize('delete', $task);
15.     $task->delete();
16.     return redirect(route('tasks.index'));
17. }

```

now missing the one we recently deleted. Again, we're seeing this happen in real-time without refreshing the entire page, without having to build and secure API endpoints, and worry about REST or GraphQL or any of that typical cruft. If you're looking to build single-page applications or just want to see what all the fuss about Inertia is, I highly recommend jumping into Laravel Breeze and exploring what you can accomplish. We've covered getting started with a fresh Laravel application using Inertia, Vue.js, and Breeze to scaffold our authentication and then extended the templates to add the ability to add, edit, and delete tasks. I relied heavily on the existing design and layout provided by Breeze, which you may find quite similar to the design used in the new Laravel Bootcamp⁴ onboarding tutorial. I hope you're inspired to test drive Laravel Breeze or even give Vue.js a spin now that you've seen it in action.



Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson

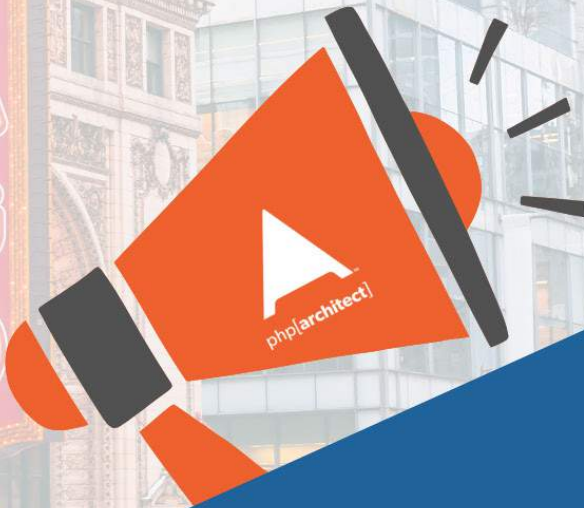


Order Your Copy

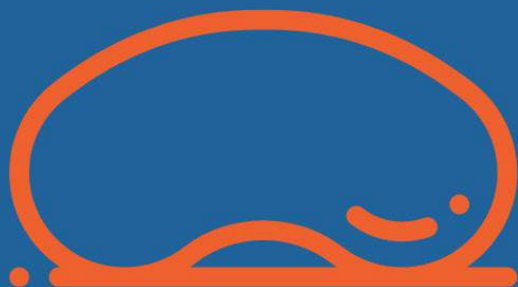
<https://phpa.me/security-principles>

⁴ Bootcamp: <https://bootcamp.laravel.com>

Call for Speakers



Share your amazing stories about Tech Leadership, PHP Development and Web Technologies. We'd love for you to apply to our Call for Speakers for the 15th annual php[tek] conference. Proposals are accepted through Jan 3rd and full travel support is available.



May 16-18, 2023
Sheraton Suites Chicago O'Hare
tek.phparch.com

Cybersecurity Checkup

Eric Mann

October is recognized as Cybersecurity Awareness Month in the United States. It's a great opportunity to stop, take stock of your current security stance, and make incremental improvements where possible.

Nearly every month of the year has a theme. February is all about chocolates and hearts. June is graduation. August through December are all about Christmas — at least in the retail world. Since 2004, the United States has declared October to be Cybersecurity Awareness Month¹ specifically so folks will spend time focusing on threats, protection from them, and overall industry awareness of cybersecurity.

Each year has a theme for the awareness campaign — 2022's theme is “see yourself in cyber.” Ultimately, the goal is to help *everyone* recognize and understand what's often seen as a complex subject. In reality, security is so foundational to what we do in technology that it's relatively simple.


In support of this awareness campaign, we want to detail four actions you can take *today* to increase your position on security.

Enable Multi-Factor Authentication

Nearly every popular web application or social media site today supports multi-factor authentication. Your username is often public information (your name, display name, or even your email address). In most systems, your password is

the one piece of secret information only you know that the system can use to validate you are who you say you are.

Unfortunately, that level of authentication can easily fail either through bugs in the system or user error — or even malicious user activity — on your part.

 One comment I regularly make when presenting on security is that, as the architect of a secure system, you should never trust the user. Any user interacting with your system could be malicious; telling the difference between a legitimate party and a bad actor is difficult so just assume everyone is a bad actor until proven otherwise.

The absolute easiest way to protect against a password breach, reused password, or bad actor managing to break their way past the password is to add extra factors to the authentication flow. We covered exactly how this increases your protection back in July², but it's useful to know which systems you should look into to protect your accounts.



1 Cybersecurity Awareness Month: <https://phpa.me/cyber-awareness>

2 July Article: <https://phpa.me/security-demystifying-mfa>



⚠️ *There are several systems online today still using SMS-based codes for additional authentication. While this form of multi-factor authentication is better than none at all, you should always endeavor to use something more secure than unencrypted text messages for authentication.*

One of the easiest forms of multi-factor authentication is time-based one-time passwords. These are short, usually 6-digit codes generated by an app on your phone that change every 30 seconds. The codes are based on a secret key known only to you and the application; due to the frequent changes of the codes, it's very difficult for an attacker to impersonate you, even if they know your password. Android and iPhone users can leverage apps like Google Authenticator³ to securely create codes for systems that support them.

A more advanced system uses app-based push notifications as a second authentication factor. Authy⁴ is one of the most popular integrations and converts your phone itself into the additional authentication factor, requiring proactive assertion that you do intend to log into any particular system.

The final form of multi-factor authentication I suggest you investigate is hardware tokens. Like the time-based passwords referred to earlier, *hardware* tokens use a secure key embedded in fault-tolerant hardware to log you in. Once configured, an attacker cannot impersonate you without having physical access to this token. The YubiKey⁵ is one of the more prevalent devices on the market today. It supports one time passwords, physical hardware tokens, and even integrates well with SSH and GPG clients to handle asymmetric encryption and authentication. Newer models support NFC communication with your phone, making it a very versatile device for authenticating anywhere!

Use Strong Passwords

Old-school advice on passwords had us creating complicated strings of gibberish that are hard to remember and even more difficult to (correctly) enter into a login field. They include upper- and lower-case characters, numbers, special characters, no dictionary words, and no repeated characters. All of these rules make a password *look* strong to a human because it feels complicated.

A computer doesn't care.

In July 2021, we talked about what *really* makes a password secure⁶ — entropy. Each of the arcane rules about character types above introduces entropy into a shorter password — the more entropy, the longer it takes for a computer to guess your password. Unfortunately, each of these character sets is

small enough that you need a rather long password to have sufficient entropy to stay secure.

If you use the rules above, your password must be at least 14 characters long to stay adequately secure from a devoted, sophisticated attacker [for at least five years](#). Alternatively, an easy-to-remember *passphrase* comprised of 5 memorably dictionary words would remain secure against the same attacker for nearly 100 *thousand* years.

Which is easier to remember: `bpz.y24X!H@TFA` or “husky conduit fiji mystery unaware?”

Moving through October, take some time to reevaluate how you build your passwords and whether things are *actually* secure or merely complicated enough that they feel that way. Also, consider leveraging a password manager like 1Password⁷ to keep track of things for you. This application synchronizes between all of your devices, keeps your private data encrypted, and can generate truly secure, random passwords (and passphrases) for you whenever needed. What's even better, the tool will notify you if any of your passwords are ever breached by a third party so you can immediately rotate your credentials and stay secure!

Recognize and Report Phishing

According to email security vendor Mimecast⁸, 91% of cyber attacks start with an email. Usually, these emails are from attackers impersonating someone you know or trust to trick you into downloading software, providing credentials, or otherwise providing them with an easy way to break into your system. These kinds of attacks are called “phishing,” and you likely have at least a few emails in your inbox or spam folder *right now* that are attempts to turn you into a victim.

There are X ways to quickly identify a phishing email:

1. Does the sender know you personally? Is this an email from a friend, colleague, vendor, or contact who is addressing you by name, or is the message addressed to “user” or “subscriber” or “[FirstName] [LastName]”?
2. Is the message asking you to do something out of the ordinary? This could be downloading an unfamiliar file or visiting a website you don't recognize. Often a phishing email will ask you to confirm your account credentials with a particular system.
3. Is there a sense of urgency or a threat for non-action?

Whenever you receive an email that looks suspicious, take steps to protect yourself. If the email appears to come from someone you know — contact them to check. Call your colleague or vendor to confirm they sent the message and need you to take action.

If the action requested by the message feels off, report it to your IT department. They have the tools required to check

3 Google Authenticator: <https://phpa.me/google-authenticator>

4 Authy: <https://authy.com>

5 YubiKey: <https://www.yubico.com>

6 July 2021: <https://phpa.me/security-july-21>

7 1Password: <https://1password.com>

8 Mimecast: <https://phpa.me/secure-email-gateway>

attachments for malware and can confirm whether or not the message is legitimate and safe.

In any situation, *never* provide your credentials in direct response to an email. There might be legitimate times a bank, vendor, or other service provider will email you and ask you to change your password. If and when they do, **do not** follow a link embedded in the message; instead, enter the correct address directly in your browser and login as usual.

When in doubt, report the email as potential phishing. If it's legitimate, your team will let you know. The original sender will likely follow up later to help you understand what's happening.

Update Your Software

The sixth most common application security risk encountered by our community, according to OWASP, is that of vulnerable and outdated components. In fact, we discussed (and demonstrated) this particular issue last December⁹. As software developers, we are responsible for ensuring our own applications stay safe and secure by checking that we're not inadvertently shipping vulnerabilities in vendor libraries.

But as users of technology, it's also our responsibility to keep our own systems and endpoints as up-to-date as possible.

Just last month, Apple disclosed a critical remote code execution flaw¹⁰ in the Webkit rendering engine that lies beneath Safari, the default browser on Mac, iPhone, and iPad. This bug could be exploited when a user visited a malicious website and gave attackers full run of the machine. Imagine the damage someone could do if you accidentally clicked an ad posted by a bad actor while browsing the web!

Apple quickly released a patch for this bug but, be honest, how many people actually install updates and reboot their computer immediately when prompted? Right now, go to the terminal and run `uptime` to see when *you* last restarted your machine (which is required to install kernel-level patches like the one released by Apple).


If you frequently defer (or flat out ignore) system updates, this month is the perfect opportunity to change that habit. Go and install your updates now. The rest of the magazine can wait until you're finished.



Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

9 December 2021: <https://phpa.me/security-dec-2021>

10 Apple code execution flaw: <https://phpa.me/apple-zero-day-2022>



From Capone to Cray

WHERE COMPUTERS REALLY CAME FROM

by Edward Barnard

Why computers? Churchill destroyed the first ones to keep the secret. What was it like? Ed shares the answers!

<https://leanpub.com/capone>

New and Noteworthy

PHP Releases

PHP 8.2.0 (RC 3 available for testing):

<https://www.php.net/archive/2022.php#2022-09-29-3>

PHP 8.1.11 (Released):

<https://www.php.net/archive/2022.php#2022-09-29-2>

PHP 8.0.24 (Released):

<https://www.php.net/archive/2022.php#2022-09-30-1>

News

php[tek] is returning

The longest running conference is returning in 2023. It is returning to Chicago, which is a beloved location for many in the community. Tickets are on sale now at the lowest price they will be.

<https://tek.phparch.com>

php[tek] seeking sponsors

Are you looking to get your brand in front of the leaders in the PHP Industry? php[tek] attendees often lead the charge at their respective organizations to bring in new tech and services. If you want your brand noticed, consider one of our sponsorship levels. Here's a link to our prospectus.

<https://phpa.me/tek2023-sponsor>

Asymmetric Visibility

Learn about the desire to make class properties publicly read only, but allowed to be set privately or protectedly. The benefits of a readonly public variable are great in a Idempotent Value Object, but sometimes the object itself needs to be able to set the property while allowing the public to read it without separate getters needing to be defined.

<https://phpa.me/externals-asymmetric-visibility>

Import Laravel Vapor DNS to Cloudflare

Cumulus is an open-source package that works with Laravel Vapor to allow the user to manage their DNS records better when using Cloudflare for DNS. The post Import Laravel Vapor DNS to Cloudflare app...

<https://phpa.me/phpnews-vapor>

Learn PestPHP From Scratch

Pest From Scratch is a free video course from Laracasts. Luke Downing walks you through the setup to becoming proficient with Pest PHP. The post Learn PestPHP From Scratch appeared first on Laravel...

<https://phpa.me/phpnews-learn-pestphp>

RFC: json_validate #PHP 8.3

In this RFC, Juan Carlos Morales proposes to add a new function called json_validate() that verifies if a string contains a valid JSON:

https://wiki.php.net/rfc/json_validate

RFC: Improve unserialize() error handling #PHP 8.3

Tim Düsterhus proposes adding a new UnserializationFailedException, which is thrown when unserialization fails:

https://wiki.php.net/rfc/improve_unserialize_error_handling

RFC: StreamWrapper Support for glob() #PHP 8.3

Timmy Almroth proposes implementing StreamWrappers support for glob() function.

https://wiki.php.net/rfc/glob_streamwrapper_support

RFC: Deprecations for PHP 8.3

An umbrella RFC that lists features to be considered for deprecation in PHP 8.3 and removal in PHP 9.

https://wiki.php.net/rfc/deprecations_php_8_3



Making Our Own Web Server: Part 1

Chris Tankersley

Why does PHP not work like a lot of other “modern” web languages, and what happens if we want it to work without a web server? Sure, we have the PHP development server, but it has always been labeled as for development only. Why is PHP the way it is, and can we make PHP process its own requests?

In a weird turn of events, one thing that sets PHP apart from many other languages is the idea that PHP itself is not a server. PHP applications rarely allow a direct connection from the internet, and almost no tutorials tell you to run the PHP you write directly and assume that HTTP connections will be handled. As it turns out, PHP applications cannot handle those HTTP connections by default.

Contrast this with the typical node application tutorial. You will install a framework like Express¹, write a few routes, and then run something like `node server.js`. Suddenly you can go directly to an address like `http://localhost:3000`² and the application is handling the HTTP connection and returning a response.

Why does PHP not work like a lot of other “modern” web languages, and what happens if we want it to work without a web server? Sure, we have the PHP development server, but it has always been labeled as for development only. Why is PHP the way it is, and can we make PHP process its own requests?

PHP Process Lifecycle

PHP still runs pretty much as it has since the early days of the web. The lifecycle of a PHP script is “start, process, die” the engine itself does not understand the concept of handling multiple connections or even a single connection. PHP expects some sort of web server to be running that will take web requests and, if needed, invoke PHP somehow.

The two most common ways are using PHP as a module inside Apache’s httpd web server and as proxied FastCGI processes using PHP’s FPM manager. In the case of httpd, httpd starts a PHP engine in each of its own processes. When a web request comes in, httpd processes the request and then pipes the requested file through the PHP engine. If there is PHP code, the PHP engine executes it and generates a response that httpd returns. PHP itself does not handle the initial request and uses httpd to return the response.

When it comes to FastCGI processes, PHP uses an internal system called FPM to spawn a handful of processes that can take a FastCGI request from an external webserver. The process is generally the same but PHP is decoupled from the web server itself.

A common setup for FastCGI is nginx with PHP-FPM. The request comes in, nginx determines if it should hand it off to PHP, and then waits for the PHP process to return a result. The result is then returned via nginx. Just like with httpd’s built-in processing, PHP does not take the original request, nor technically return the response. The web server does that.

In either of these two cases, PHP still invokes the old “start, process, end” mantra. The request comes in, the engine processes it as a procedural block of code with a distinct startup and shutdown lifecycle, and goes on from there. The httpd threads and PHP-FPM processes might handle several connections throughout their lifetime, but they only ever process one request at a time. If you have ten httpd processes or ten PHP-FPM processes, you can only process ten requests at a time.

PHP does this because socket programming, which deals with handling network connections between the client and the server, can be incredibly complicated. In a bout of true Unix purity, PHP handles dynamic responses and leaves the actual dirty work of handling the TCP connections to software better suited to handle it. We do not, and should not, care about the actual connection handling. We just need PHP to understand the request, do some magic, and return a response.

The PHP Dev Server

PHP did add a development server³ starting with PHP 5.4.0. It provides a very basic, single-threaded web server that can be used to run an application without the need for a web server. For most developers, this works well when dealing with an application that is fully PHP and does not rely on any special web server or other language processing.

Starting with PHP 7.4 it is possible to run a multi-threaded development server. As with the main invocation, this is not production ready and should only be used for testing applications. This does not mean the single-threaded server has graduated to production status, it is still only to be used for development purposes.

Since its inception, the development server has been labeled as not production ready, and there is a good chance

¹ Express: <https://expressjs.com>

² `http://localhost:3000`: <http://localhost:3000/>

³ a development server: <https://phpa.me/manual-features>



it never will be. As explained before, PHP's stance is that it should be web server agnostic. You, as the developer, can choose which server you want to run, and PHP will implement standards-complaint ways to allow you to interface with it (i.e., FastCGI).

If you are not familiar with the development server, you can invoke it with:

```
$ php -S <ip>[:port] [-t /docroot] [/router.php]
```

You specify the IP address and optional port for PHP to listen on. Any requests that come to the address will be handled directly by the PHP engine. By default, it will serve files from the current directory, or you can also specify a document root with `-t`.

The development server can also take a “router” script, allowing you to short-circuit out of having the engine process a file. This is useful for directly serving file types that PHP does not understand. PHP automatically understands a handful of commonly used mime-types to serve up; however, if you have special files or additional logic, you can let the router script return false early just to have the file served up directly.

As an example, if you are working on a Laravel application where the document root is the `public/` folder, you would invoke it with the following and then access it at `http://localhost:8080`:

```
$ cd path/to/application/
```

```
$ php -S localhost:8080 -t public/
```

Some frameworks like Laravel or Symfony may ship additional tools that start a web server. These are generally wrapping the PHP development server but may provide other functionality like additional debugging.

The most significant downside to the development server is that it can only handle a single connection at once. As applications become more demanding in terms of concurrent connections, you can quickly bottleneck something like a single-page application that is constantly making requests to the PHP backend. This may not be noticeable on your local machine, but scale this to just a few users and the requests will start to block each other.

The development server is also not as performant serving static files. PHP understands how to serve about fifty different mime types, but you have to instruct PHP on how to handle anything outside of those mime types. In general, full web servers are better equipped to serve static files than the PHP engine and may even provide caching mechanisms that the development server does not.

The development server is also very bare-bones. Servers like `httpd` and `nginx` allow for more complex logic in handling

requests before they are ever dispatched, like rewriting URIs to new URIs, passing connections off to other sockets and servers, and being able to handle scaling for thousands of connections.

At the end of the day, the development server is just that—a simple way to access PHP applications without the need for a full-service web server, but the handful of downsides keep it from being a way to deploy web applications.

Can PHP be a Web Server?

What if we wanted our application to handle its own HTTP connections and not be dependent upon a web server? Since most of the core of PHP is a wrapper on the C language itself, it turns out that some of those really low-level functions have made their way up to PHP. One set of those functions deals with network sockets.

A network socket is an endpoint for network communication. Since we will be working with HTTP connections, we will want to make a network socket that works with TCP/IP, which is the fundamental protocol that governs how two computers talk to each other over the internet. This is specifically known as a socket address.

A socket address is made up of an IP address and a network port. We can write an application that opens and listens on a socket address. When data comes in on that socket address, we can do some work and return a response. This is what the PHP development server is doing at a very basic level.

Writing a simple socket server only takes a few built-in methods in PHP. These are:

- `socket_create`⁴ - Creates a rudimentary socket that we can configure
- `socket_bind`⁵ - Bind the socket to an IP address and port
- `socket_listen`⁶ - Start listening on the socket
- `socket_accept`⁷ - Start accepting connections via the socket
- `socket_read`⁸ - Read information from the socket
- `socket_write`⁹ - Write information back to the socket
- `socket_close`¹⁰ - Close the socket when we are done

These seven functions allow you to create a basic socket server that will accept a single request at a time and return a response. Putting these all together is pretty close to invoking all these functions in the above order.

4 `socket_create`: <https://phpa.me/manual-function>

5 `socket_bind`: <https://phpa.me/manual-function-socketbind>

6 `socket_listen`: <https://phpa.me/manual-function-socketlisten>

7 `socket_accept`: <https://phpa.me/manual-function-socketaccept>

8 `socket_read`: <https://phpa.me/manual-function-socketread>

9 `socket_write`: <https://phpa.me/manual-function-socketwrite>

10 `socket_close`: <https://phpa.me/manual-function-socketclose>



```
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
socket_bind($socket, '0.0.0.0', 8080);
socket_listen($socket);
```

The first thing we do is create a socket using `create_socket()`. Since sockets are a fundamental building block and can be used for various purposes, we need to specify what kind of socket we need. We want to accept an HTTP connection over TCP/IP, so we configure it by telling the function that we are going to listen for IPv4 connections (via `AF_INET`) and that we want a socket that can be read from and written to (via `SOCK_STREAM`), and that we will be using the TCP/IP protocol (via `SOL_TCP`).

Next, we bind the socket to an IP address and a port using `socket_bind()`. We pass in the socket we configured, an IP address, and a port number. In this case, we will listen on all IP addresses on the computer and on port 8080. This means we should eventually be able to access our site on `http://localhost:8080`, or even `http://<any IP address on the computer>:8080`.

Then we start to listen using `socket_listen()`. Our socket starts to listen on the configured address and port number. Now we need to actually accept and respond to a request. See listing 1.

Listing 1.

```
1. while (true) {
2.   $connection = socket_accept($socket);
3.   $buffer = socket_read($connection, 1024, PHP_NORMAL_READ);
4.   echo $buffer;
5.
6.   socket_write($connection, $buffer);
7.   socket_read($connection, 1024);
8.   socket_close($connection);
9. }
```

Now we just need to wait until a connection is made. We will do this inside of a `while()` loop, so once one connection finishes, we will wait for another.

Since our socket is just listening current, we can use `socket_accept()` to wait for a connection to come in. This will block our script until a connection is made. Our script will sit here until a connection is made to our address and port.

Once the connection is made and accepted, we stop working directly with our socket for a bit and work with the connection we just made. We can read the incoming data using `socket_read()`. This takes the connection and a length of data to read in, and we also specify that we want all of the data to be returned as a string and stop at line endings with `PHP_NORMAL_READ***`. We could also specify `PHP_BINARY_READ` if we knew we were not getting textual data.

We save this data off into a buffer. From here, you can do whatever you want, but we will simply just echo it out to the

console for the time being. You could parse the text as JSON or XML, as an HTTP request, or whatever you think the incoming data is. Since we are doing a simple socket server, we will just accept text.

Now we can write information back to the socket. For the sake of triviality, we will just echo whatever the user typed back to them. We use `socket_write()` and pass the connection and the buffer back as parameters. This will send the data back to the client just as they had typed it.

Now that we have sent data, we do one final read in case the client sends anything back. We do not care what that response is, so we do not save it off like we did the initial read. This is mostly just to clear the socket of any information before we close the connection with `socket_close()`.

The connection is closed, and our `while()` loop starts the process all over again. If we run this script, we can use a utility like netcat to make a basic socket connection and test the response: See listing 2.

Listing 2.

```
$ netcat localhost 8080
test
test
```

With this small amount of code, we can open, accept, and respond to a network request. This is what the PHP development server does at a basic level, but right now we cannot process an HTTP request properly.

What About Http?

As mentioned in April 2020's "Anatomy of a Web Request", we went over that an HTTP request is a string in a specific format. This string contains the HTTP request verb, the URI we are requesting, the HTTP version, an optional list of key-value pairs called Headers, and potentially a request body. See listing 3.

Listing 3.

```
1. GET / HTTP/1.1
2. Host: localhost:8080
3. User-Agent: insomnia/2022.4.2
4. Content-Type: application/json
5. Accept: */*
6. Content-Length: 17
```

If we want to handle our own HTTP requests, we should have a way of parsing this request. If something does not look like an HTTP request, we can return an error, but if the HTTP request looks valid, we can take action on it.

Parsing an HTTP request can be incredibly complicated. A variety of RFCs detail exactly how this request should be formatted, and there are probably more clients that are not standards-compliant than clients that fully follow the



HTTP standard. Luckily someone has already written an HTTP request parser, which is part of the `laminas-diactoros` package¹¹ from the Laminas framework.

```
$ composer require laminas/laminas-diactoros
```

`laminas-diactoros` can be installed via composer, just like all the other Laminas packages. We can then use the `Laminas\Diactoros\Request\Serializer` class to parse a request and turn it into a PSR-7 request. Let's change our code to do that: See listing 4.

Listing 4.

```
1. while (true) {
2.
3.     $connection = socket_accept($socket);
4.
5.     $buffer = socket_read($connection, 1024, PHP_NORMAL_READ);
6.
7.     $request = Serializer::fromString($buffer);
8.
9.     // We'll handle the response in a moment
10. }
```

We can use the `fromString()` method on the serializer to turn the buffer we read from the connection into a PSR-7 object. Now that we have this, we can hand this off to any PHP library or framework that understands how to handle a PSR-7 request!

If we are going to handle an HTTP request properly, we cannot just echo the request back to the client. We need to generate a proper HTTP response. As a follow-up to the April 2020 issue, we discussed what makes up an HTTP response in the May 2020 issue in the article, “Anatomy of Web Response.”

Like a request, a response is a string response letting the client know the HTTP version you are responding with, a numerical code indicating the status of the response, and a textual overview (like 200 OK). There may be additional headers; in almost all cases, there will be a response body.

Like the request, we could build this all ourselves, but `Diactoros` has a serializer that can take a PSR-7 HTTP Response object and convert it back to a string. This serializer takes care of formatting everything to the HTTP standard and is much easier to work with than trying to build this string by hand.

Let's create a response that returns a string that tells the user the URI and HTTP used for the request. See listing 5.

We use `Laminas\Diactoros\Response` to create a new PSR-7 response with a return code of 200. We then write to the body of that request the message telling the user what URI string and HTTP method they passed. At this point, you could add additional headers or any other manipulations you wanted, all while taking advantage of the PSR-7 interfaces.

Listing 5.

```
1. while (true) {
2.     $connection = socket_accept($socket);
3.     $buffer = socket_read($connection, 1024, PHP_NORMAL_READ);
4.
5.     $request = Serializer::fromString($buffer);
6.
7.     $response = (new Response())->withStatus(200);
8.     $message = 'You requested ' . $request->getUriString() .
9.     $message .= ' with verb ' . $request->getMethod();
10.    $response->getBody()->write($message);
11.
12.    $responseString = Serializer::toString($response);
13.
14.    socket_write($client, $responseString);
15.    socket_close($client);
16. }
```

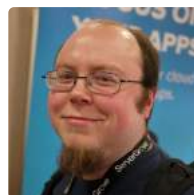
When we are all done, we use `Laminas\Diactoros\Response\Serializer::toString()` to turn the object into an HTTP compliant string. We can then write that back up the socket and close the connection out. Our script should now handle requests from browsers, API tools like Insomnia or Postman, or any other HTTP client and be able to parse and display a response!

We are Almost There

We can now accept an HTTP request and return an HTTP response. One thing that we can do that the normal PHP development server cannot do is directly manipulate the request before handling it. All of this code is also user-land code without using any extensions, so we can take the PSR-7 request and just deal with it directly.

We do not need to read a file and parse it; we can throw the request into anything that can parse a PSR-7 request, like a PSR-15 middleware stack, and just execute the code path directly. We could do things like pass static files through streaming static files through a PSR-7 response.

One thing we did not do any better is multiple connections. There are a few ways we can handle this, and next month we will look at taking this basic HTTP server and allowing it to handle concurrent connections. We will not be at the “production ready” stage yet, but there is still a lot more we can do to expand on this basic socket server.



*Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. 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](https://twitter.com/dragonmantank)*

¹¹ `laminas-diactoros` package:

<https://github.com/laminas/laminas-diactoros>

Application Event Walkthrough

Edward Barnard

This month presents a relatively quick code walkthrough even though this feature is more complex than the previous chapter. We'll see the structure is similar to the Domain Event feature. We'll focus on those points important to developing your instinct with Strategic Domain-Driven Design.

Figure 1.

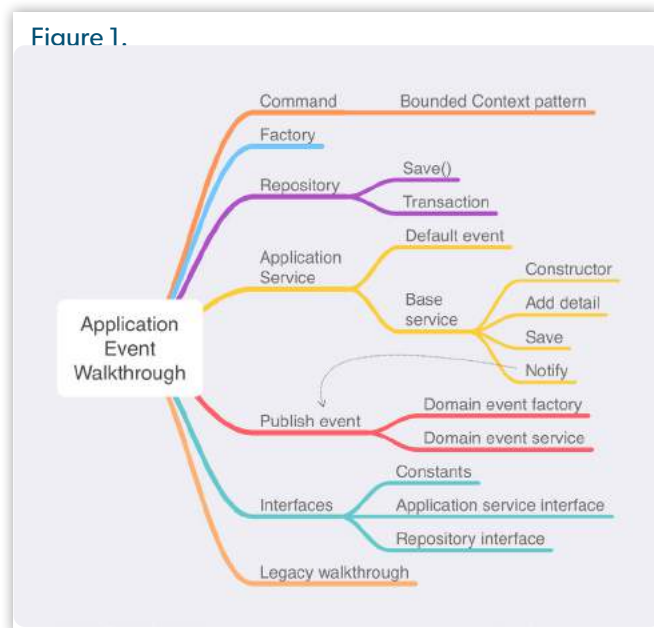
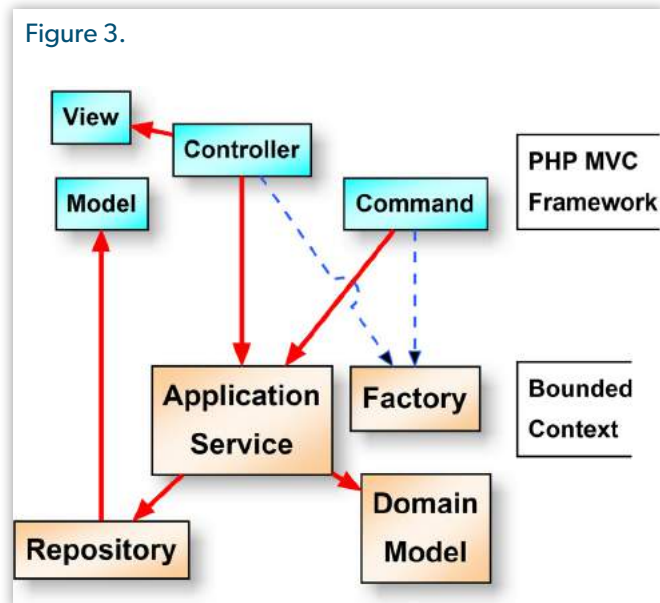


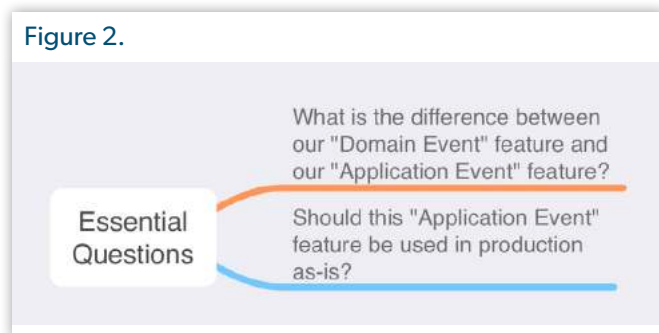
Figure 3.



Essential Questions

Upon surviving this article, you should be able to answer

Figure 2.



(see Figure 2.):

- What is the difference between our “Domain Event” feature and our “Application Event” feature?
- Should this “Application Event” feature be used in production as-is?

Application Event Command

This month's code walkthrough is quite similar to the Domain Event feature we saw last month. We'll focus on the differences. We'll continue to conform to the Bounded Context design shown in Figure 3.

Things will get more interesting as soon as we get past the event factory in the next section. We'll have to do some negotiating with our database administrator.

Please note that there is a massive amount of code examples in this article. We've had to adjust most of them to fit the layout. Please download the code archive¹ to get the full listings.

Listing 1 on the next page shows our test harness. The Application Event Factory creates our application service. The next line, the call to `save()`, would normally be within a database transaction. We'll see its full production usage next month. The next line, calling `notify()`, publishes the Application Event if and only if the database transaction completed

¹ code archive: https://phpa.me/October2022_code



Listing 1.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command;
6.
7. use ...\AppEventFactory;
8. use Cake\Command\Command;
9. use Cake\Console\Arguments;
10. use Cake\Console\ConsoleIo;
11.
12. final class AppEventCommand extends Command
13. {
14.     /**
15.      * @throws \JsonException
16.      */
17.     public function execute(Arguments $args, ConsoleIo $io):
18.     {
19.         $action = 'Command-line test';
20.         $description = 'app_event command';
21.         $appEvent = AppEventFactory::defaultAppEvent(
22.             $action,
23.             $description
24.         );
25.         $appEvent->save();
26.         $appEvent->notify();
27.
28.         return 0;
29.     }
30. }
```

successfully. Again, we'll see the full production workflow next month.

Application Event Factory

Listing 2 shows two factory methods. `defaultAppEvent()` instantiates the repository and passes it into the Application Service constructor. The second method, `dbStateChangeAppEvent()` simply delegates to `defaultAppEvent()` with a hard-coded event action.

Our full feature, coming next month, will record every database update as a “database state change” application event.

Don't blindly put a feature like this, recording every database state change, into production! It will generate a very large number of table rows and possibly impact database performance. However, it can be useful in a development environment, allowing you to get a comprehensive picture of database activity.

Default Application Event Repository

Finally! Things get a bit more interesting with listing 3 on the next page. The `save()` method shows our general pattern for a manual database transaction within the CakePHP

Listing 2.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace ...\AppEvent\Factory;
6.
7. use ...\ApplicationServices\DefaultAppEvent;
8. use ...\DomainModel\Constants\CAppEventOriginatingContexts;
9. use ...\DomainModel\Interfaces\IAppEvent;
10. use ...\Repository\RAppEventDefault;
11.
12.
13. class AppEventFactory implements CAppEventOriginatingContexts
14. {
15.     private function __construct()
16.     {
17.     }
18.
19.     /**
20.      * @throws \JsonException
21.      */
22.     public static function defaultAppEvent(
23.         string $action,
24.         string $description,
25.         ?array $detail = null
26.     ): IAppEvent {
27.         $repository = new RAppEventDefault();
28.         return new DefaultAppEvent(
29.             $repository, $action, $description, $detail
30.         );
31.     }
32.
33.     /**
34.      * @throws \JsonException
35.      */
36.     public static function dbStateChangeAppEvent(
37.         string $description,
38.         ?array $detail = null
39.     ): IAppEvent {
40.         return self::defaultAppEvent(
41.             self::ACTION_DB_STATE_CHANGE,
42.             $description, $detail
43.         );
44.     }
45. }
```

framework. We'll be passing in raw MySQL strings as `$insert` and `$read`. The `$parms` array is equally dangerous with bare values that must match the order specified in `$insert`.

I don't generally recommend flinging low-level instructions at the database engine like this. We do use `prepare()` and `execute()` to prevent SQL injections. Why bother?

I wrote it this way to remove a dependency. Our database administrator agrees that there's a strong possibility that the current application we're developing will begin to use multiple databases. Assuming that we don't allow MySQL transactions to span across databases, we'll need to have a separate local



Listing 3.

```

1. <?php
2.
...
31. public function save
    (string $insert, string $read, array $parms)
32. {
33.     $connection=$this->localAppEventsTable->getConnection();
34.     try {
35.         $connection->transactional(
36.             function ($conn) use ($insert, $read, $parms) {
37.                 $statement = $conn->prepare($insert);
38.                 $statement->execute($parms);
39.                 $statement = $conn->prepare($read);
40.                 $statement->execute([$statement->lastInsertId()]);
41.                 $readback = $statement->fetchAll('assoc');
42.                 if (!(is_array($readback) &&
43.                     array_key_exists(0, $readback))) {
44.                     throw new DatabaseException
45.                         ('Event readback failed');
46.                 }
47.                 $this->readback = $readback[0];
48.             }
49.         );
50.     } catch (Exception) {
51.         return [];
52.     }
53.
54.     return $this->readback;
55. }
...

```

Listing 4.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace ...\ApplicationServices;
6.
7. final class DefaultAppEvent extends BaseAppEvent
8. {
9.     protected static string $insert = <<<QUERY
10.         insert into `local_app_events`
11.         (action, subsystem, description, detail,
12.          event_uuid, when_occurred, created, modified)
13.         values (?, ?, ?, ?, ?, now(6), now(), now())
14.     QUERY;
15.
16.     protected static string $read =
17.         'select * from `local_app_events` where id = ? limit 1';
18. }

```

store in each database, with the global Domain Event store elsewhere.

One approach would be to write separate repositories for each local store. Another approach is to use low-level `prepare()` and `execute()`. As we'll see next month, the latter approach is simpler because we need to pass in the database

connection to stay inside the transaction. If you prefer the other approach, that's okay!

There's another reason for going "low level" with raw SQL queries using `prepare()` and `execute()`. As we turn to Strategic Domain-Driven Design in our own projects, we are also upgrading MySQL server versions, PHP compiler versions, framework versions, and so on.

In the legacy codebase, we are also considering which ORM (database layer) to use while at the same time upgrading the existing database support. The raw SQL strings have proven to be the most framework-independent approach because of PHP's built-in support for PHP Data Objects² (PDO).

I am *not* advocating that you build your entire PHP application using raw hand-built SQL strings! But I'll continue to show the advantages as they impact our exploration of Strategic Domain-Driven Design. Once you understand those principles, you'll know how to design your own database-related software layers.

Default Application Event

Listing 4 shows the raw MySQL strings for inserting and reading the new application event. This arrangement allows each child class to provide slight variations on a theme, such as the type of application event being captured. Subsystem and source table can also be adjusted in the child class, as we'll see in the base class.

Base Application Event

Listing 5, on the next page, provides a useful example showing that most use case logic may reside in the Application Service. There might not be a Domain Model in use at all. The repository had a careful focus on a single database transaction.

Most of this feature's logic and processing flow are right here in the Application Service. That will be the normal situation for many of our production use cases, perhaps most. When the use case is not particularly complex, this is all we need.

The constructor is more complex than we've seen to this point. We're capturing everything that we'll need for storing the event. The constructor calls `validateSubclass()` for a sanity check to make sure we didn't miss anything when setting up a new subclass for a new type of application event.

The public method `addDetail()` can be useful inside a long and complex transaction. For example, when persisting an invoice with line items, all part of the same database transaction, we could accumulate detail after each row insert.

The public method `save()` is designed to be called within a database transaction. We're designing this feature to include manual transactions, with the transaction processing inside a closure. Thus, rather than returning the readback array (i.e., a

2 PHP Data Objects: <https://www.php.net/manual/en/book.pdo.php>



Listing 5.

```
1. <?php
...
39. public function __construct(
40.     IAppEvent $repository,
41.     string $action,
42.     string $description,
43.     ?array $detail = null
44. ) {
45.     $this->repository = $repository;
46.     $this->action = $action;
47.     $this->description = $description;
48.     $this->detail = is_array($detail) ?
49.         json_encode($detail, JSON_THROW_ON_ERROR) :
50.         null;
51.     $this->uuid = Uuid::uuid4()->toString();
52.
53.     $this->validateSubclass();
54. }
55.
56. private function validateSubclass(): void
57. {
58.     ...
59. }
60.
61. /**
62.  * @throws \JsonException
63.  */
64. public function addDetail(array $detail): void
65. {
66.     $prior = (null == $this->detail) ?
67.         [] :
68.         json_decode((string)$this->detail, true);
69.     $new = array_merge($prior, $detail);
70.     $this->detail = json_encode($new, JSON_THROW_ON_ERROR);
71. }
72.
73. public function save(): void
74. {
75.     $parms = [
76.         $this->action,
77.         static::$ subsystem,
78.         $this->description,
79.         $this->detail,
80.         $this->uuid,
81.     ];
82.     $this->readback = $this->repository->save(...);
83. }
84.
85. public function notify(): void
86. {
87.     if (empty($this->readback)) {
88.         return;
89.     }
90.     $domainEvent = DomainEventFactory::domainEvent();
91.     $domainEvent->notifyDomainEvent(...);
92. }
```

copy of the row that was just inserted), we store it within the Application Service.

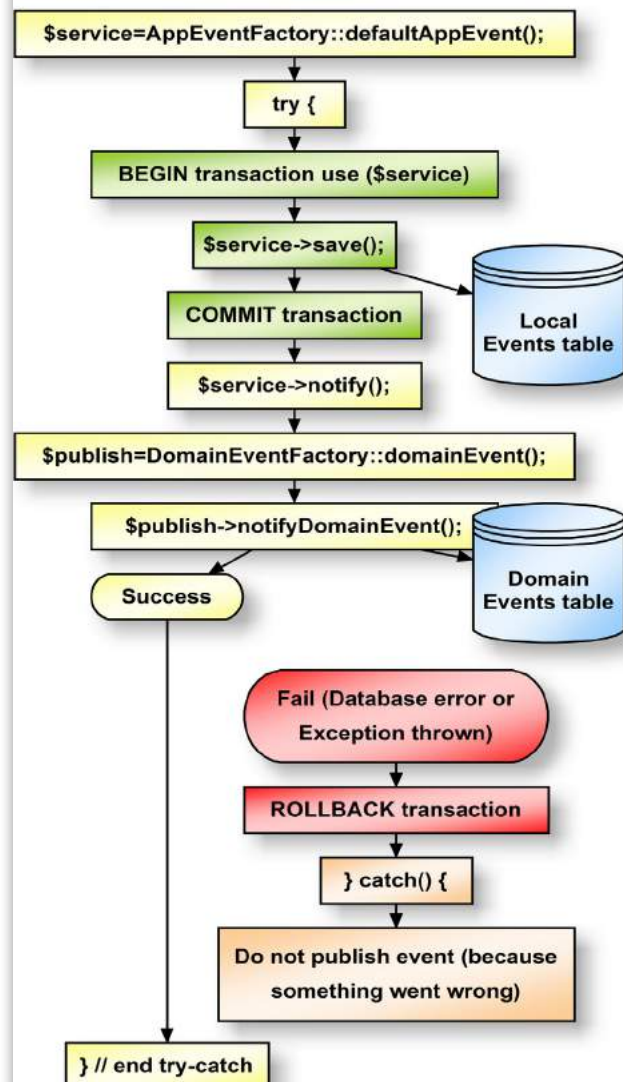
Calling this class an Application Service is a bit of a misnomer. It's the best terminology I've found, but it's still a problem. The real problem is that the word "service" takes on too many meanings. A "real" service generally needs to be immutable so that its behavior remains predictable over time. Here we are dealing with a transient object, a use case handler that embodies both behavior and current state.

Notify

The key feature of our Application Event service is `notify()`. It is completely separate from `save()`. That's because `save()` runs inside the database transaction, but `notify()` runs after

Figure 4.

Publish Application Event





the transaction successfully commits. `save()` saves the event to the local event store; `notify()` publishes the event.

How do we publish the event? That is the “publish domain event” use case. We’re connecting two use case handlers together. The Domain Event Factory provides the Application Service. We then invoke its `notifyDomainEvent()` entry point. See Figure 4.

Have we made this more difficult than necessary? Do we really need this two-step notification, connecting the Application Event feature to the Domain Event feature? Could we just fold them into one?

No. Production code would normally use some sort of messaging mechanism such as RabbitMQ³. There’s simply no need to clutter up this example with messaging infrastructure.

How would that work? The Application Event service `notify()` packages up the input parameters (`static::$sourceTable`, `$this->readback`) and forwards it to RabbitMQ. The Domain Event service gets invoked when the message arrives, and the Domain Event service stores the event in the global Domain Event store.

Listing 6.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\,\,\DomainModel\Constants;
6.
7. interface CAppEventOriginatingContexts
8. {
9.     public const SOURCE_TABLE_PRIMARY = 'local_app_events';
10.    public const SUBSYSTEM_DEFAULT = 'Default';
11.    public const SUBSYSTEM_ROLE_BASED_LOOKUP = '...';
12.    public const ACTION_DB_STATE_CHANGE = 'db_state_change';
13. }
```

Originating Event Context

Listing 6 shows another separation. The PHP interface construct allows both public constants and public methods. Here we separated the constants from the methods. Why?

Any class that implements the interface (which only contains class constants) inherits the constants. For example, the class could then reference `self::SOURCE_TABLE_PRIMARY`. I find this fact extremely useful for PHP projects that use a lot of magic numbers⁴ (Unnamed_numerical_constants) or strings.

If we combine constants and function methods in the same file, those constants are only available by specifying the class name or by the class implementing those methods. But when “implementing” only the constants, an IDE such as PhpStorm can auto-complete the values.

³ RabbitMQ: <https://www.rabbitmq.com>

⁴ magic numbers: <https://phpa.me/wiki-magic-number>

Listing 7.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\...\Interfaces;
6.
7. interface IAppEvent
8. {
9.     public function __construct(
10.         IAppEvent $repository,
11.         string $action,
12.         string $description,
13.         ?array $detail = null
14.     );
15.
16.     public function addDetail(array $detail): void;
17.
18.     public function save(): void;
19.
20.     public function notify(): void;
21. }
```

Listing 8.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\...\DomainModel\Interfaces;
6.
7. interface IAppEvent
8. {
9.     public function save(
10.         string $insert,
11.         string $read,
12.         array $parms
13.     ): array;
14. }
```

Application Event Interface

Listing 7 and listing 8 allow our Factory to mix-and-match as needed.

We have a bit of a tricky situation in that the repository is designed to handle writing to any of several different tables. We pass a raw MySQL string to the repository, and that string includes the table name. Type-hinting the interface, rather than the actual repository class, allows us to connect things up correctly. Only the Factory needs to know how those connections actually work.

There’s another very important reason for using a separate interface for the repository, and that is the fact that it can simplify. Most of this feature’s complexity is in the application service rather than the repository. It’s a fairly simple matter for the test suite to include a test fixture that implements this `IAppEvent` interface.



Legacy Application Event Walkthrough

Porting to the legacy codebase gets easier and easier with practice. Once again, I found it easiest to type up the code from the above listings in reverse order:

- Application Event repository interface
- Application Event service interface
- Originating context constants
- Base Application Event service
- Default Application Event service
- Default Application Event repository
- Application Event factory
- Test harness

Listings 9-16 are available in the downloadable code archive⁵.

Listing 9 shows the repository interface is identical except for namespace.

Listing 10 shows the Application Service interface is also identical. Note that when passing the repository into the constructor, we are type-hinting the repository interface.

Listing 11 shows the constants are also identical.

Listing 12 shows the base Application Service is essentially identical, as it should be. It has no framework dependencies. The only difference is the property annotations due to being compatible with PHP 7.3.

Listing 13 shows the child class is essentially the same.

Listing 14 shows the repository has the same processing flow but uses a different database access layer.

Listing 15 shows the factory is essentially identical.

Listing 16. shows the test harness is a bare PHP script but follows the same processing flow.

Below shows the output is concise. One row did get inserted into both the `local_app_events` and `domain_events` tables (not shown).

Figure 5.



```
~# php test/exercise_app_event.php
Application event complete
```

Essential Questions Answered

See Figure 5.

- *What is the difference between our "Domain Event" feature and our "Application Event" feature?* The Application Event, once published, is captured as the Domain Event.
- *Should this "Application Event" feature be used in production as-is?* No.

Summary

This was a quick walkthrough. We saw a repetition of the same structure as with Domain Event. We now have the pieces in place to finish out our feature, namely dealing with random and rare failures.



Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others.
[@ewbarnard](https://twitter.com/ewbarnard)

⁵ code archive: https://phpa.me/October2022_code

study says skype for linux sucks.
THE study:



Daniel Stori {turnoff.us}

turnoff.us | Daniel Stori
Shared with permission from the artist

Converting Float Strings

Oscar Merida

Converting data is never as straightforward as we'd initially expect. Users can enter data incorrectly or in the wrong format. On the other hand, computers may not have trouble working with some data. For this article, we look again at floating-point values.

Why do I keep returning to this problem? Frequently, we're overconfident in trusting the machines or underestimate the frequency at which we might encounter them. When you're dealing with values that represent currency, a bug can literally cost you. At work, we use a backend system that stores prices for tours. We read prices from the API, which returns them as formatted strings, like \$1,234.56. Our PHP app stores that value as an integer representing the amount of pennies for a given price. Converting decimal values to integers should be easy with `round()`, no?

Recap

Write a function that takes a currency string for dollars and converts it to the equivalent amount in pennies. The String may be formatted like "\$100" or "\$100.00". The dollar sign is optional. Confirm it works with all the following values "105.00", "\$ 128.76", "\$2,487.47", '11,135.65', '\$71,135.71'; '11,135.65', and '\$71,135.71'.

A Naive Approach

Let's ignore dollar signs and commas to begin with. If we have a string that's a floating point value, we can cast that to a float, multiply by 100, cast that to an int, and we'd be done. See Listing 1.

Listing 1.

```
$ php -a
php > $price = (float) "105.00";
php > $price = $price * 100;
php > var_dump($price);
float(10500)
php > $price = (int) $price;
php > var_dump($price);
int(10500)
```

Normalizing the Input

That should work; now, let's write a function to clean up incoming strings and return a float. I decided to use `str_replace()` to remove unwanted characters, but doing so treats an input like \$1,,452\$\$.12 as valid. I'm assuming the

incoming formatting is somewhat consistent, but if needed, we could write code to validate the format before trying to convert it.

```
function priceToFloat(string $input) : float
{
    $output = str_replace(['$', ',', ' '], '', $input);
    return (float) $output;
}
```

We can set up an array to hold our incoming data for processing.

```
$raw = ["105.00", "$ 128.76", "$2,487.47", '11,135.65', '$71,135.71'];
$floats = array_map('priceToFloat', $raw);
```

This gives us a `$floats` array that looks like the following.

```
array(5) {
    [0]=> float(105)
    [1]=> float(128.76)
    [2]=> float(2487.47)
    [3]=> float(11135.65)
    [4]=> float(71135.71)
}
```

We've standardized our incoming data. Now, all we need is to convert it to pennies.

```
function floatToPennies(float $input) : int
{
    return (int) ($input * 100);
}
```

`array_map()`¹ makes quick work of converting our floats using this new function. Let's see what we get. I'm using `array_combine()`² to line up our floats and integers as the keys and values of a single array to inspect the output.

```
$pennies = array_map('floatToPennies', $floats);
$result = array_combine($floats, $pennies);
var_dump($result);
```

1 `array_map()`: https://php.net/array_map

2 `array_combine()`: https://php.net/array_combine



Are we all done?

```
array(5) {
  [105]=> int(10500)
  ["128.76"]=> int(12876)
  ["2487.47"]=> int(248746)
  ["11135.65"]=> int(1113565)
  ["71135.71"]=> int(7113571)
}
```

Did you spot it? We lost a penny when converting 2487.47 as it became 248746. Sure, it's just one penny. However, if someone has to reconcile transactions, they'll have to chase down why some invoices are incorrect. The difference adds up quickly if you do a large enough volume of transactions with the wrong amounts.

What's going on? Let's look at multiplying by 100:

```
$x = 100 * 2487.47;
var_dump($x); // float(248746.99999999997)
```

Remember, computers work in bits and bytes and powers of 2. Floating point values are based on powers of 10. There's a mismatch in how accurately the former can represent the latter. In practice, this can lead to weird rounding errors like the one we're seeing. In this case, we need to round the conversion to a whole number before casting to an int.

```
function floatToPennies(float $input) : int
{
    return (int) (round($input * 100));
}
```

If we update the function and run our earlier code, we'll see that we don't lose a penny in any conversion.

Use Bcmath

In this case, `round()`³ was sufficient for our problem. It may not always work though, especially if you are multiplying one float by another. To ensure you don't lose precision with floating point math, use the BCMath extension⁴.

```
function floatToPennies(float $input) : int
{
    return (int) bcml($input, 100);
}
```

This extension should be installed and enabled with most PHP distributions. However, if you're missing it—or can't install it easily on your local development environment, there is a user land shim⁵ you can install.

```
$ composer require phpseclib/bcmath_compat
```

³ `round()`: <https://php.net/round>

⁴ BCMath extension: <https://php.net/book.bc>

⁵ a user land shim: https://github.com/phpseclib/bcmath_compat

Under the hood, it loads automatically if the extension is not available. You don't need to do anything special to use the `bc*` functions.

World Cup Draws

Since the World Cup is (finally) next month in sunny Qatar, let's use it as our puzzle.

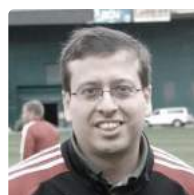
Write a program to simulate a FIFA World Cup draw with participants from this year's tournament. Use the same pots to group teams as in the actual draw and assign every team to one of the eight tournament groups (A-G). Teams from the same pot can not be in the same group. Except for European teams, a team can also not be in a group with another team from their region (confederation). No group can have more than two European teams.

Some Guidelines And Tips

- The puzzles can be solved with pure PHP. No frameworks or libraries are required.
- Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.
- You're encouraged to make your first attempt at solving the problem without using the web for clues.
- Refactoring is encouraged.
- I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.
- Go ahead and try many solutions if you like.
- PHP's interactive shell (`php -a` at the command line) or 3rd party tools like `PsySH`⁶ can be helpful when working on your solution.
- To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.

Related Reading

- *PHP Puzzles: Fractional Math* by Oscar Merida, September 2022. <https://phpa.me/puzzles-sep-2022>
- *PHP Puzzles: Decimals to Fractions* by Oscar Merida, August 2022. <https://phpa.me/puzzles-aug-2022>



Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. [@omerida](https://twitter.com/omerida)

⁶ `PsySH`: <https://psysh.org>



Laracon Online 2022

Eric Van Johnson

During this time of the virtual lifestyle, Laracon Online has been one of the better and most engaging online conferences out. I mean, in my opinion, it doesn't even come close to a good in-person conference, like *hmm, I don't know, maybe php[tek] 2023 in Chicago*. But as for virtual, Laracon Online is really well organized, and last month's broadcast was no exception.

The Year of Livewire?

I've been using the Laravel Framework since the release of Laravel 4, and there has been no other package that had me so quickly and drastically changing my development toolchain. I went from trying to learn various javascript frameworks to handle my front end to feeling like I could accomplish any of that UI sugar clients like to see in their application, using mainly PHP with Livewire. And I've been beating the Livewire drum from the first day I got my hands on it.

Caleb Porzio (@calebporzio) showed no signs of slowing down with his ideas for Livewire in his talk "The future of Livewire," where he discusses his complete rewrite of Livewire 3. One of Caleb's most significant changes was that AlpineJS would now be baked into Livewire. If you've used Livewire in the past, you know there is a lot you can get away with just using the package out of the box. But there would always be those little quality-of-life things javascript would bring to your page. That is why Caleb created a companion javascript framework for Livewire called AlpineJS. AlpineJS was minimalistic and simple to integrate into your application with Livewire. Well, now it's even easier because it's all one package in Livewire 3.

One of the past complaints about using Livewire was the number of network calls it made. Livewire could constantly be pinging back to the server to check and retrieve information. The more components you had on your page, the more network chatter it would cause. Coming in Livewire 3, it will handle that better by batching network calls.

Livewire 3 has also implemented an annotation practice that lets you get even more out of your Livewire components. Honestly, I could probably write this entire column on Livewire, but there were so many other presentations to mention.

Community

One of the more refreshing talks I watched was from Caneco (@caneco) called "The Hitchhiker's Guide to the Laravel Community." We pride ourselves here at php[architect] on being a community-driven group, and community is of enormous importance to me. For that reason, it was great to listen to Caneco talk about all the different aspects of the Laravel Community and how someone new to Laravel can quickly and easily get plugged in.

The Darkhorse

One talk I didn't have a lot of high expectations for was Aaron Francis's (@aarondfrancis) "Database Performance for Application Developers." Don't get me wrong, I can appreciate a good DBA, but my eyes would quickly glaze over when trying to watch a talk on database performance.

Aaron discusses things to keep in mind when designing your schemas, keeping them as small as possible but as big as you need. He talks about (B-tree) indexes and how to best go about creating them—left to right, no skipping. Watch his talk to understand that better. He wraps up his talk by discussing queries. You could be making some suboptimal queries without knowing it.

Laravel, No Holding Back

Any talks during Laracon Online had enough substance to write an entire column. I am honestly brushing over the few I enjoyed to make sure I can fit this article into the magazine, but none of this would be possible if I didn't cover the person who made this happen, Taylor Otwell (@taylorotwell). If you've seen Taylor present, you know he is pretty reserved. He gets more excited when he talks about code than anything else. I always appreciated that about Taylor. He's not some salesperson who knows how to do a couple of tricks in a given solution they were trying to get you to buy into, but Taylor is a real developer, one of us, who genuinely enjoys development.

He spent much of his time just talking about many new features and improvements they had released over the past year in the current version of Laravel. Taylor explained that with the new yearly release cycle of Laravel, they don't hold on to new features until the next release, as they would in the past.

There were several "quality of life" improvements to existing artisan command outputs and some new ones, such as the new artisan about command, which gives you one of the best high-level views of your application and its environment. Then there was the artisan db:show command that will provide you with stats about your configured database connections in Laravel. A favorite of mine is the artisan model:show <model name> command that shows you all the attributes for the model you request.



As for new things coming to Laravel, he demoed attachable objects for Mailable, which cleans up your code when adding attachments to emails. New invokable rules allow for a much more intuitive way of writing custom rules, especially when they have multiple conditions where they can possibly fail. Finally, one of the things I am excited about is including a way to easily use UUIDs and ULIDs on Laravel models without having to write your own custom trait to handle it, which is something I've been doing for the past few years. Using UUIDs as primary keys and other column variables has never been easier.

And that isn't everything. You will want to watch the video yourself to ensure you get all the juicy new and shiny things released in Laravel.

video's description to get a complete list of timestamps to all the talks, and hop around to your heart's content. Quickly bounce between talks that interest you, but I must stress that all the talks were good and as time permits, I recommend giving them all a watch. You can catch Laracon Online 2022 here <https://www.youtube.com/watch?v=f4QShF42c6E>.



Eric Van Johnson is one of the minds behind PHP Architect, LLC. Described as "loud and passionate" about the PHP Programming language, he has been known to record a podcast or two or three (PHPUGly, PHP Roundtable, and php[podcast]). Powered by scotch, and hope, you can follow Eric on Twitter as [@shocm](https://twitter.com/shocm).

It's Not Too Late

Laracon Online was around 10 hours long, but if you missed it or any part of it, there is nothing to worry about because it's on YouTube for the attractive price of FREE! Expand the



LONGHORN PHP CONFERENCE

Nov 3-5, 2022

Austin, TX

**A 3-day conference to help PHP
developers level up their craft and
connect with the larger PHP
community.**

Keynote Speakers



Boyd Hemphill



Christina Aldan



Mike Lehan



Rissa Jackson

<https://www.longhornphp.com>





The Web Developer's

SECRET WEAPON!

Exception, uptime, and cron monitoring, all in one place and easily installed in your web app. Deploy with confidence and be your team's devops hero.

Are exceptions *all* that keep you up at night?

Honeybadger gives you full confidence in the health of your production systems.

DevOps monitoring, for developers. *gasp!*

Deploying web applications at scale is easier than it has ever been, but monitoring them is hard, and it's easy to lose sight of your users. Honeybadger simplifies your production stack by combining three of the most common types of monitoring into a single, easy to use platform.

Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring

Know when your external services go down or have other problems.

Check-In Monitoring

Know when your background jobs and services go missing or silently fail.



Start Your Free Trial Today
<https://www.honeybadger.io/>



Beware: Tek is Disruptive

Beth Tucker Long

Chicago May 16-21 2023

In between where we start and how we end up, lies the journey. It's usually filled with uncertainty along the way; however, we're often grateful for the path we took in the end.

Once upon a time (back in the dark ages of PHP 3), there was a young PHP programmer working for a non-profit. Let's call her...choosing a totally random name...how about... Beth. (Yup, it's me.) Self-taught and siloed as the sole IT staff member, I learned as much as possible from scattered blog posts and message boards. The non-profit continued to grow, and after about seven years, they could finally afford a second IT person. Another young PHP programmer, newly graduated, was hired. This new programmer wanted to attend a conference in Chicago she had heard about called php[tek] (back then it was php:tek). She campaigned hard and convinced management to set up a yearly education budget for the IT department. She got her ticket, and when she returned, she was overflowing with excitement about programming, ideas for improving our systems, and suggestions for new tools and frameworks to research. The conference had not only invigorated her, but it had taught her so many things.

We started implementing all kinds of new ideas and technologies. We updated our workflow. We became more efficient and created more stable, easier-to-maintain code. She instantly began encouraging me to attend a conference for my portion of the education budget. The same group of people were running another conference in the fall called php:works in Atlanta, and she was determined to convince me to go. I was skeptical that I would get anything out of attending a conference, but she won me over, and I boarded a plane to Atlanta.

php:works was unlike anything I had ever attended before. I met the people building the tools I was using. I learned so much about new ways of handling data and security. I didn't just learn what I should be doing, but why I should be doing it. Fantastic talks in the conference rooms led to fascinating discussions in the hallways. I started building a network of connections to programmers in all different types of fields and all different areas of the world.

This conference changed my life in the most amazing ways. Not only did this improve my coding skills, but it also cemented how important the community is, so I helped revive my local user group. My new network got me a job when I wanted to switch to working remotely before my children were born. That new job gave me the push I needed to start speaking at conferences. Speaking at conferences helped me grow my own business and start working for myself full-time. This independence gave me the flexibility to start speaking at international conferences. At one of these international conferences, I heard a talk from a woman working with an organization called Open Sourcing Mental Illness (OSMI). She talked about how her diagnosis helped her better understand herself and what she needed. This helped me seek my own therapy and diagnosis, which have radically improved my quality of life.

Looking back, that PHP 3 programmer would never have imagined what I have become today - a happy and healthy woman, an internationally-known speaker, a regular columnist in the longest-running PHP magazine, a business owner, a user group leader—all while doing the work I love, coding.

And it all started with a trip to tek.



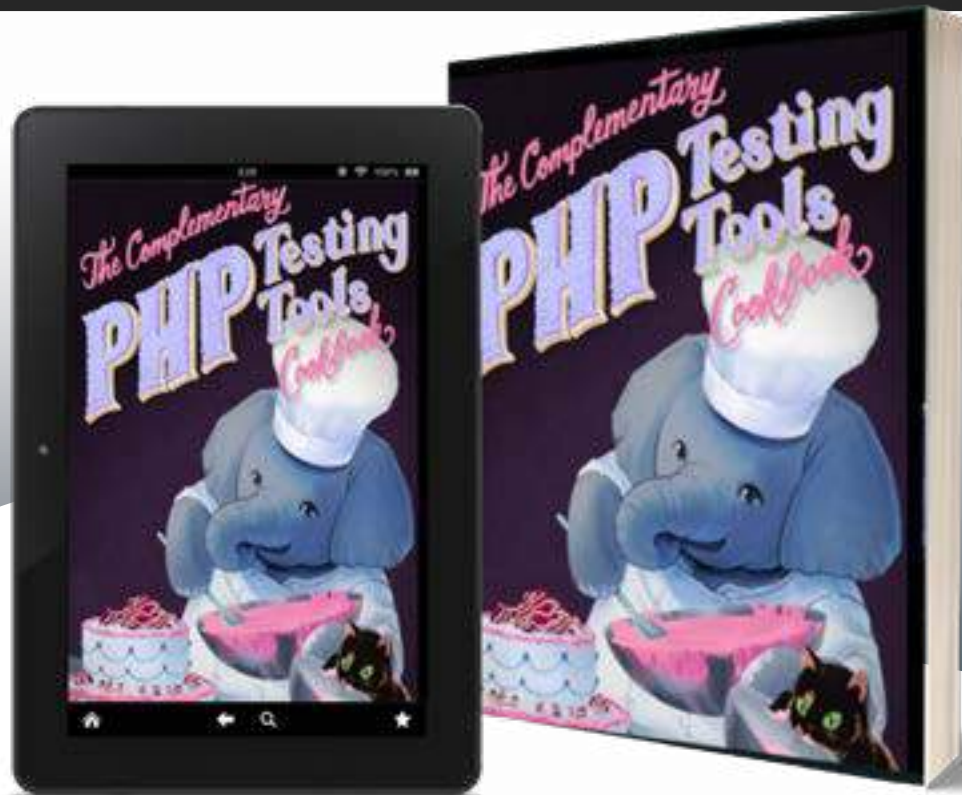
Beth Tucker Long is a developer and owner at Treeline Design¹, a web development company, and runs Exploricon², a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development³ and Full Stack Madison⁴ user groups. You can find her on her blog (<http://www.alittleofboth.com>) or on Twitter @e3BethT

1 Treeline Design: <http://www.treelinedesign.com>

2 Exploricon: <http://www.exploricon.com>

3 Madison Web Design & Development: <http://madwebdev.com>

4 Full Stack Madison: <http://www.fullstackmadison.com>

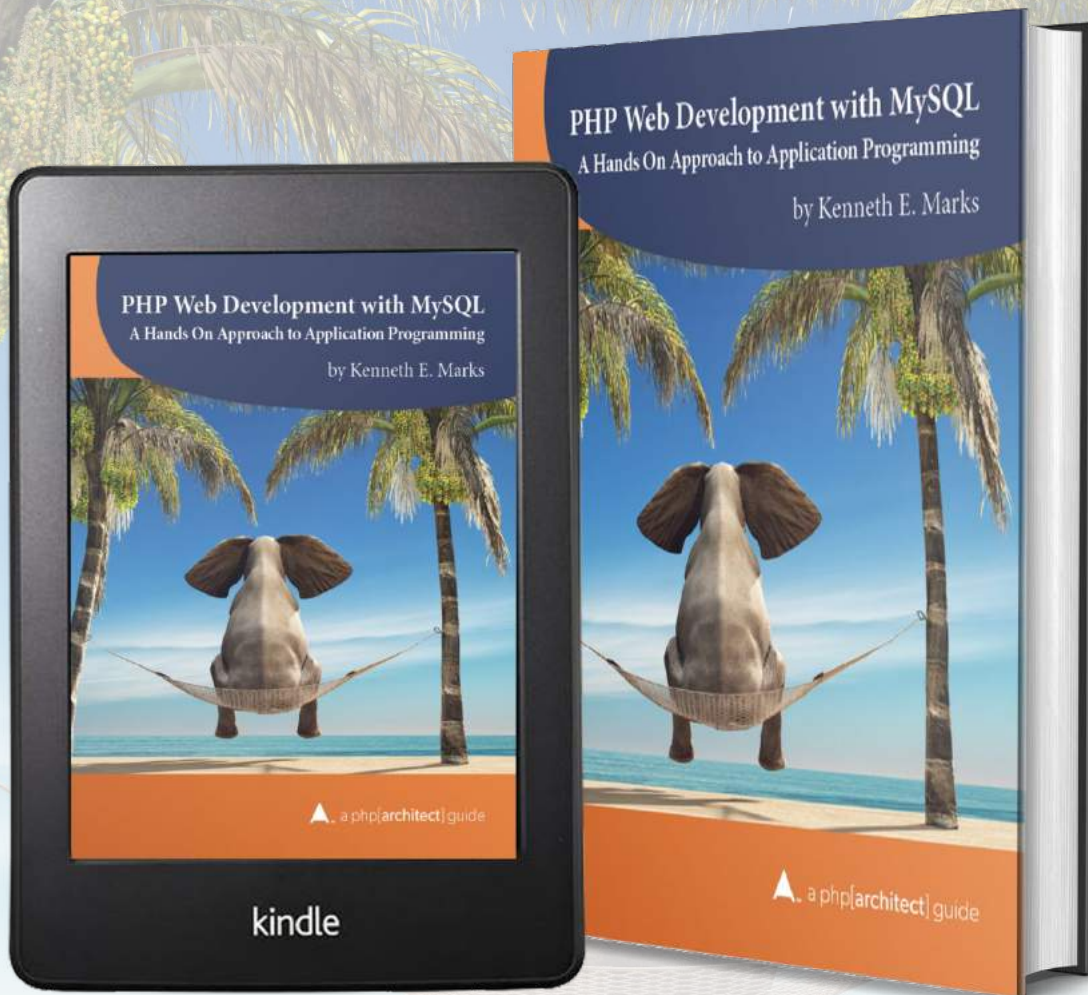


Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

The Complementary PHP Testing Tools Cookbook is Chris Hartjes' way to try and provide additional tools to PHP programmers who already have experience writing tests but want to improve. He believes that by learning the skills (both technical and core) surrounding testing you will be able to write tests using almost any testing framework and almost any PHP application.

Available in Print+Digital and Digital Editions.

Order Your Copy
phpa.me/grumpy-cookbook



Learn how to build dynamic and secure websites.

The book also walks you through building a typical Create-Read-Update-Delete (CRUD) application. Along the way, you'll get solid, practical advice on how to add authentication, handle file uploads, safely store passwords, application security, and more.

Available in Print+Digital and Digital Editions.

Purchase Your Copy
<https://phpa.me/php-development-book>