



www.phparch.com

December 2022  
Volume 21 - Issue 12

# php[architect]

The Magazine For PHP Professionals

## *Owning The Web*

**Harness The Power of the AST**

**Bring Value To Your Code**

ALSO INSIDE

**The Workshop:**  
Get a Blog!

**Education Station:**  
Refactoring Yourself

**PHP Puzzles:**  
Sticker Swapping

**Security Corner:**  
Debt Management

**DDD Alley:**  
Exception Report

**PSR Pickup:**  
PSR-13: Link Definition

**Finally{ }:**  
Who Controls Your Content?

# 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](https://www.cloudways.com)





# CONTENTS

**DECEMBER 2022**  
**Volume 21 - Issue 12**



- |   |  |
|---|--|
| <b>2 The Joy of the PHP Season</b><br>Eric Van Johnson                  | <b>24 PSR-13: Link Definition Interfaces</b><br>PSR Pickup<br>Frank Wallen |
| <b>3 Harness The Power of the AST</b><br>Tomas Votruba                  | <b>27 Get A Blog</b><br>The Workshop<br>Joe Ferguson                       |
| <b>10 Bring Value To Your Code - Part 2</b><br>Dmitri Goosens           | <b>31 Debt Management</b><br>Security Corner<br>Eric Mann                  |
| <b>17 Refactoring Yourself</b><br>Education Station<br>Chris Tankersley | <b>33 Exception Report</b><br>DDD Alley<br>Edward Barnard                  |
| <b>21 Sticker Swapping</b><br>PHP Puzzles<br>Oscar Merida               | <b>42 Who Controls Your Content?</b><br>finall}<br>Beth Tucker Long        |

Edited in the North Pole

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](mailto:contact@phparch.com) for more information.

#### Advertising

To learn about advertising and receive the full prospectus, contact us at [ads@phparch.com](mailto:ads@phparch.com) today!

#### Contact Information:

General mailbox: [contact@phparch.com](mailto:contact@phparch.com)  
Editorial: [editors@phparch.com](mailto: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.

# The Joy of the PHP Season

The year is quickly coming to an end. Here in the US, we just celebrated one of the more significant holidays in our country and have started the march to the final holidays of the year. As I reflect back on the past year, it feels good to see how I've continued to grow as a PHP developer, in no small part, thanks to php[architect]. I've tackled things like Event Sourcing and started to embrace other principles with Domain Driven Design, such as Value Objects which has changed the way I look at how I handle data coming into and being passed around my application. I continue to fine-tune my development skills thanks in no small part to all the great contributors to this magazine.

Another tool for growing as a developer has always been conferences. With the world also moving forward and learning how to live and function, in-person conferences are starting back up. I would be remised if I didn't mention that php[tek] is also making a return in May of 2023, and you can get all the details about that at <https://tek.phparch.com>. I am not going to lie to you, I obviously have a vested interest in this particular conference as myself and John are on the hook to make this conference as strong and impactful as so many of the Tek's have been in the past.

I talked about my journey to my first php[tek] a couple of months ago, so I won't rehash that now. Still, I also recall over the years all the tweets and mentions from people in User Groups about what an impact attending a php[tek] conference had on them and their development path. Our goal is to keep that tradition of excellent talks and stellar insight into PHP and the PHP Community going in 2023.

As the time draws closer and our excitement here at php[architect] continues to grow, I am sure php[tek] 2023 will be a recurring theme of editorials in the upcoming months, but it's not the reason I brought it up this month. This editorial is more of a reflection of the growth and the gift of continual learning, and with that said, let's see what treats our columnists have for us this final month of 2022.

Speaking of Value Objects, Dimitri Goossens wraps up his Feature contribution, *Bring Value To Your Code*. In this part, Dimitri looks at some of the benefits and perks of using Value Objects in your codebase. Tomáš Votruba brings us his final contribution talking about the abstract syntax tree and discussing the theory behind AST and some fundamental tools in *AST Part 3*.

Oscar Merida uses his passion for the FIFA World Cup in his column, PHP Puzzles, to share the solution to last month's exercise, *Sticker Swapping*. If you have some holiday leave, it's an excellent opportunity to sharpen your developing knives and take a crack at this month's puzzle around Shared Birthday Days. Do you remember your first blog? I do, and they are still a thing. In The Workshop, Joe Ferguson shares with us some of his ideas and approaches to creating, owning, and maintaining a blog in his column *Get A Blog!* Domain Driven Design has been one of the most talked about topics in PHP circles over the past year, and our Edward Barnard has been living and writing about the DDD life for a while. This month he looks at how to capture the "rare and random" failures in this month's DDD Alley, *Exception Report*.

Frank Wallen's column PSR Pickup this month reviews *PSR-13:Link Definition Interface*. Chris Tankersly discusses some soft skills this month in Education Station which his article *Refactoring Yourself*. In Security Corner, Eric Mann writes about something we've all had to deal with at some point in our careers, *Debt Management*, and why not all debt is bad. We often talk about convenience vs. security, but there is another area of our everyday life that this probably impacts, and we don't give it much thought. More and more people depend on services to manage things like emails, music, movies, photos, and documents, but what happens when these services go away or something goes wrong within the service itself? In Finally, Beth Tucker Long wraps things up with *Who Controls Your Content?*

## 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](mailto: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/December2022\\_code](https://phpa.me/December2022_code)

# Harness The Power of the AST

Tomas Votruba

This is the last post of a 3-part series where you learn how you can change the world with AST. In the first post, we learned about abstract syntax tree, AST parser, node traverser, and node visitors. We now know the smallest item of AST is a node. In the second post, we explored three existing tools in the PHP world that can modify the abstract syntax tree of PHP. We also know how to change configs and even a little bit of templates. Everything uses prepared node visitors that can change exactly what we want. We now know the theory behind AST, three fundamental tools, and how to use them. Today we'll take it a step further. Here you'll become a real master of change with the ability to shape the world of code. **We'll create our own tool to parse code into AST.**

## 3 Reasons to Write Your Own Parser

There are three reasons why we want to have the skill to write our own AST parser:

- we have many files in a language with no existing tool to parse it into AST, e.g., .json files
- we want to automate changes, e.g., within our continuous integration workflow
- a simple “find and replace” is not enough for us

## Task: Validate City Names

Let's say we're working with thousands of JSON files from an external and REST service. These files contain data about coffee locations all around the world. But this service is an external provider that we have no control over. It worked very well until it recently provided a coffee location in a city called “New Yerk”, and this information propagated to our production website: (See Listing 1)

Not the best day for advertisers of this coffee, as you can imagine. That's why the business asked us for validation of city names. They must be existing and valid names in one of the states in the USA. If there is a non-existing city or a bad word, it will simply get skipped. We'll log a notification and the coffee will never reach the production website. The world is safe again.

How do we do that?

Listing 1.

```
{
  "coffees": [
    {
      "name": "Coffee Atrium",
      "city": "New Yerk"
    }
  ]
}
```

## Option 1. The Manual Approach:

At first glance, we can simply foreach the data with coffees key:

```
$jsonArray = json_decode($response, true);
foreach ($jsonArray['coffees'] as $coffee) {
    $coffeeCity = $coffee['city'];
    // validate the $coffeeCity here
}
```

It looks good; it's simple and works. We assume every coffee has city item set, and the coffees are only found under this key.

But the external service also provides another item, a group of coffee under a certain tag or category that unites those coffees: (See Listing 2)

*NOTE: The listings have been modified to fit the magazine layout. The json\_decode parameters should include 512 for depth and JSON\_THROW\_ON\_ERROR for flags.*

Listing 2.

```
1. {
2.   "coffees": [
3.     {
4.       "name": "Coffee Atrium",
5.       "city": "New Yerk"
6.     }
7.   ],
8.   "coffee-groups": {
9.     "speciality-coffee": [
10.      {
11.        "name": "Coffee Roastery",
12.        "city": "New Yerk"
13.      }
14.    ]
15.  }
16. }
```

## Listing 3.

```

1. $jsonArray = json_decode($response, true);
2. foreach ($jsonArray['coffees'] as $coffee) {
3.     $coffeeCity = $coffee['city'];
4.     // validate the $coffeeCity here
5. }
6.
7. foreach ($jsonArray['coffee-groups'] as $groups) {
8.     foreach ($groups as $groupName => $coffees) {
9.         foreach ($coffees as $coffee) {
10.             $coffeeCity = $coffee['city'];
11.             // validate the $coffeeCity here
12.         }
13.     }
14. }

```

Now we have extended our script: (See Listing 3)

The script got a bit complicated and less readable.

To add more fuel to the fire, the external service might only return one of those lists. We have to add validation to avoid crashing: (See Listing 4)

Now is a good time to ask a code quality question: what is wrong with this code?

The algorithm works; we can read it and understand the code. Yet the code is very specific and requires the exact structure of the key names and the nested arrays on specific locations, and if we **assume** the structure will never change. When the inevitable change does occur, we'll first get a fatal error report and then try to figure out exactly where it broke and prepare a test with a fix.

## Listing 4.

```

1. $jsonArray = json_decode($response, true);
2. foreach ($jsonArray['coffees'] ?? [] as $coffee) {
3.     $coffeeCity = $coffee['city'];
4.     // validate the $coffeeCity here
5. }
6.
7. $coffeeGroups = $jsonArray['coffee-groups'] ?? [];
8. foreach ($coffeeGroups as $groups) {
9.     foreach ($groups as $groupName => $coffees) {
10.         foreach ($coffees as $coffee) {
11.             $coffeeCity = $coffee['city'];
12.             // validate the $coffeeCity here
13.         }
14.     }
15. }

```

This code is:

- hard to maintain
- specific
- hopes for array format to stay the same

The last item is like assuming the nuclear reactor works because there is no fire in the village 5 miles away.

## Let's Step Back: What is the Goal?

When we get confused and nested like this, it helps to clear our mind and refresh. What is our goal here?

We want to validate the city name. We don't care about the coffee place name, and we don't care about what prices they have or the opening hours. We don't care about the price range, whether the coffee is home-roasted or from an external supplier.

We want to validate the city name. Let's put that into the code:

```

// we get "$city" on input
// then we validate the city name,
// throwing an exception on fail
validate_city_name($city);

```

This is it. This is our final feature that we can reuse on any JSON. How is that useful? Usually, projects are more complex than processing a single JSON array. It's not unlikely we'll want to validate city names on any other input JSON.

**We strive for a universal algorithm that we write once and will never have to touch again.** Unless some bad word becomes a legal city name or "New Yerk" actually becomes a city name in the future.

## Option 2. The Node Visitor Approach

So how can we make our script check general? We'll use the abstract syntax tree, of course.

Now we know the JSON always has an array on the input. How do we get the "city" item? It's an array item that has a string with a "city" key and is followed by a string value. In a PHP code, this would look like this: (See Listing 5)

Just two early returns and then a validation function call. That's it; we can use it anywhere. The final node visitor implementation will be very similar to this code.

How do we get to the node visitor? We have to write a parser and node traverser first.

## Step 1: Let's Write a Parser

The job of the parser is to convert specific input code to a generic tree of node objects we can work with further.

## Listing 5.

```

1. if (! isset($item['city'])) {
2.     return;
3. }
4.
5. $city = $item['city'];
6. if (! is_string($city)) {
7.     return;
8. }
9.
10. // here we know we're dealing with city
11. validate_city_name($city);

```



I picked JSON for a reason; it's one of the simplest formats we can work with. There are no objects, no magic, no magical references or syntax—simple string with nested strings.

The most complex rule of JSON is that it can have two forms:

```
{
  "key": "value"
}
```

and

```
{
  "key": [
    "value",
    "another-value",
  ]
}
```

The item value is a string, or the item value is an array of nested items. That's it.

Let's convert the input string from the external service into arrays. (See Listing 6)

The parse() method now produces an array with various contents and nesting. But what good can we do with an array?

#### Listing 6.

```
1. namespace PhpArch\Ast;
2.
3. final class JsonParser
4. {
5.     public function parse(string $inputResponse): array
6.     {
7.         return json_decode($inputContents, true);
8.
9.         // or including validation
10.        return json_decode($inputContents, true
11.                           512, JSON_THROW_ON_ERROR);
12.    }
```

## Step 2: Convert Into Node Objects

As any PHP developer knows, arrays are pretty leaky to work with. If we modify an array, the value can be missed. Maybe we got used to objects so much that we forgot they didn't return a reference or return the changed array.

That's why the main rule of writing an AST parser and traverser is to use objects. So let's convert arrays to fancy objects: (See Listing 7)

The createNodes() method structure is set. Except for the contents:

```
// @todo parse the $key and $value to objects
```

#### Listing 7.

```
1. namespace PhpArch\Ast;
2.
3. final class JsonParser
4. {
5.     public function parse(string $inputResponse): array
6.     {
7.         $jsonData = json_decode($inputContents, true);
8.
9.         return $this->createNodes($jsonData);
10.    }
11.
12.    private function createNodes(array $jsonArray): array
13.    {
14.        $nodes = [];
15.
16.        foreach ($jsonArray as $key => $value) {
17.            // @todo parse the $key and $value to objects
18.        }
19.
20.        return $nodes;
21.    }
22. }
```

But how do we fill this gap so it returns a tree of connected node objects?

Before we get into thinking about the algorithm, let's take the input and output exercise first:

```
{
  "key": "value"
}
```

What object do we expect here to get in return? Simple

#### Listing 8.

```
1. namespace PhpArch\Ast\Node;
2.
3. final class ItemNode
4. {
5.     public function __construct(
6.         public string|int $name,
7.         public string $value
8.     ) {
9.     }
10. }
```

ItemNode with "key" and "value". It could look like this: (See Listing 8)

Then we can create the node like this:

```
$itemNode = new ItemNode("key", "value");
```

```
var_dump($itemNode->name);
// "key"
```

```
var_dump($itemNode->value);
// "value"
```

Easy, simple, and... public. **Why are the properties public?** In 2022, we're used to writing clean code with constructors and getters, right?

The reason is simple; we use AST not just for reading data but also for modification. By convention, most AST tools written in PHP, e.g., nikic/php-parser and phpstan/phpdoc-parser, use public properties. That way, we can get them with reflection easily and avoid guessing setter and getter method names.

Now that we know, we should also handle the 2nd JSON item—with nested items:

```
{
    "key": [
        "first-value",
        "second-value"
    ]
}
```

What could this node class look like? (See Listing 9)

The usage would be as simple as: (See Listing 10)

Now we can handle both single and many array items. But what happens if we get this code on input?

```
{
    "key": [
        [
            "first-value",
            "second-value"
        ]
    ]
}
```

Now the array is nested in an array. But we require a single `ItemNode` in `$subNodes`, so another class, `ChildAwareItemNode` would crash here. How do we deal with that?

## Step 3: Add Abstract Node

That's why there is a convention of **using a common abstract node class** that shares the type and common properties.

In our case, we introduce `AbstractJsonNode`:

```
namespace PhpArch\Ast\Node;

abstract class AbstractJsonNode
{
    public function __construct(public string $name)
    {
    }
}
```

And let both node classes inherit from it:

```
namespace PhpArch\Ast\Node;

-final class ChildAwareItemNode
+final class ChildAwareItemNode extends AbstractJsonNode
{
    // ...
}
```

and

```
namespace PhpArch\Ast\Node;

-final class ItemNode
+final class ItemNode extends AbstractJsonNode
{
    // ...
}
```

That's it! Now we have all the node classes we need. It's time for the array to nodes factory method `createNodes()` to come true. (See Listing 11)

This is the biggest method we've written so far. It's important to understand this piece of code, so let's look at it closely:

- if we get the item with a single value, we create `ItemNode`
- if we get the item with a nested array, we create `ChildAwareItemNode` with the help of recursion

**The recursion is the trick** that allows us to convert any number of nested input items like a piece of cake.

Alright, we've got the parser, we've got the nodes, and we've just created an abstract syntax tree from a messy array:

### Listing 9.

```
1. namespace PhpArch\Ast\Node;
2.
3. final class ChildAwareItemNode
4. {
5.     /**
6.      * @param ItemNode[] $subNodes
7.      */
8.     public function __construct(
9.         public string|int $name,
10.        public array $subNodes
11.    ) {
12.    }
13. }
```

### Listing 10.

```
1. $childAwareItemNode = new ChildAwareItemNode("key", [
2.     new ItemNode(0, "first-value"),
3.     new ItemNode(1, "second-value"),
4. ]);
5.
6. foreach ($childAwareItemNode->subNodes as $subItemNode)
7. {
8.     var_dump($subItemNode->value);
9.     // first "first-value"
10.    // then "second-value"
11. }
```



Listing 11.

```

1. namespace PhpArch\Ast;
2.
3. use PhpArch\Ast\Node\AbstractJsonNode;
4. use PhpArch\Ast\Node\ChildAwareItemNode;
5. use PhpArch\Ast\Node\ItemNode;
6.
7. final class JsonParser
8. {
9.     // ...
10.
11.     /**
12.      * @return AbstractJsonNode[]
13.      */
14.     private function createNodes(array $jsonArray): array
15.     {
16.         $nodes = [];
17.
18.         foreach ($jsonArray as $key => $value) {
19.             if (is_array($value)) {
20.                 // A. has array children?
21.                 $nestedNodes = $this->createNodes($value);
22.                 $nodes[] = new ChildAwareItemNode(
23.                     $key,
24.                     $nestedNodes
25.                 );
26.             } else {
27.                 // B. is simple node?
28.                 $nodes[] = new ItemNode($key, $value);
29.             }
30.         }
31.
32.         return $nodes;
33.     }
34. }

```

```
namespace PhpArch\Ast;
```

```

$jsonParser = new JsonParser();
$jsonNodes = $jsonParser->parse($inputResponse);
// var_dump($jsonNodes);
// array of `PhpArch\Ast\Node\AbstractJsonNode` objects

```

Have you heard? **We have created an abstract syntax tree with object nodes.**

Now the fun begins!

## Step 4: Create a Node Traverser

As we already know, the node traverser is a single generic service that goes through every node and nested node. It traverses the abstract syntax tree from the very top node to the very bottom one.

We already know we have two types of node classes, so we'll have two ways to traverse the json node: (See Listing 12)

Let's look at this piece of code closely.

- the `traverse()` visits every node
- if the `$node` is instanceof `PhpArch\Ast\Node\ChildAwareItemNode` we run `traverse()` all its children

Listing 12.

```

1. namespace PhpArch\Ast;
2.
3. use PhpArch\Ast\Node\AbstractJsonNode;
4. use PhpArch\Ast\Node\ChildAwareItemNode;
5.
6. final class JsonNodeTraverser
7. {
8.     /**
9.      * @param AbstractJsonNode[] $jsonNodes
10.     */
11.     public function traverse(array $jsonNodes): void
12.     {
13.         foreach ($jsonNodes as $jsonNode) {
14.             // @todo add node visitors
15.
16.             // traverse all the children
17.             if ($jsonNode instanceof ChildAwareItemNode) {
18.                 // @todo add node visitors
19.                 $this->traverse($jsonNode->subNodes);
20.             }
21.         }
22.     }
23. }

```

This is the recursion we just did above, just an inverted process.

The last piece of the puzzle is missing. The implementation of node visitors.

## Step 5: Add Visitor Implementation

The node traverser, without node visitors, is like a gun with no bullets:

```

$jsonParser = new PhpArch\Ast\JsonParser();
$jsonNodes = $jsonParser->parse($inputResponse);

```

```

$jsonNodeTraverser = new PhpArch\Ast\JsonNodeTraverser();
$jsonNodeTraverser->traverse($jsonNodes);

```

The gun itself works, the mechanics make sense, and it clicks and clacks. But would you bring such a gun into a fight? I guess not.

The node visitors are the bullets we provide. So how do we load them to the node traverser?

First, we'll add a "bullet interface"... er, the `JsonNodeVisitorInterface`:

```
namespace PhpArch\Ast\Contract;
```

```
use PhpArch\Ast\Node\AbstractJsonNode;
```

```

interface JsonNodeVisitorInterface
{
    public function enterNode(AbstractJsonNode $jsonNode);
}

```

It's very simple; it "visits" or "enters" the node in `enterNode()` method. That's it.

Now, to actually load a bunch of these node visitors into our node traverser, we need to improve `JsonNodeTraverser` for extension—with an `addVisitor()` method: (See Listing 13)

Now we have the full logic ready. Make the change happen.

## Step 6: Make First Node Visitor

Where exactly do we add the node visitor? Right before the `traverse()` method, so we shoot with the bullets:

```
$jsonNodeTraverser = new PhpArch\Ast\JsonNodeTraverser();
$jsonNodeTraverser->addVisitor(...);
$jsonNodeTraverser->traverse($jsonNodes);
```

Now we can finally get back to our goal with validation. This was our original script: (See Listing 14)

Now our job is to write the same algorithm in an AST node syntax with a node visitor. Do you have an idea in your mind? Try it yourself; I know you know it already.

My idea of `ValidateCityJsonNodeVisitor` looks like this: (See Listing 15 on the next page)

Let's look at the `enterNode()` method closely. The early return is the name of the game:

- We only look for simple `ItemNode`, so we return if this is not the case
- We look for items with the key of “city”; otherwise skip
- If the conditions all pass, we have **a city value we can finally validate**

That's it! This script is reusable on any JSON input, with any number of nesting, with cities in coffee shops, restaurants, or pubs, with single or 10-level nesting, grouped or flat.

We add it to the `JsonNodeTraverser` and `traverse`:

```
use ...\JsonNodeTraverser;
use ...\ValidateCityJsonNodeVisitor;
...
$jsonNodeTraverser = new JsonNodeTraverser();

$jsonNodeVisitor = new ValidateCityJsonNodeVisitor();
$jsonNodeTraverser->addVisitor($nodeVisitor);

$jsonNodeTraverser->traverse($jsonNodes);
```

That's it!

## Final Step: Use the Power of Ast

Now you've written your first JSON AST parser + node traverser + node visitor. You know how to add another node visitor that verifies data or modifies them.

You can also easily correct the city names:

```
if ($jsonNode->value === 'New Yerk') {
    $jsonNode->value = 'New York';
}
```

The code we wrote today is fully functional. It has zero dependencies, well... except for PHP 8.0.

### Listing 13.

```
1. namespace PhpArch\Ast;
2.
3. use PhpArch\Ast\Contract\JsonNodeVisitorInterface;
4. use PhpArch\Ast\Node\AbstractJsonNode;
5. use PhpArch\Ast\Node\ChildAwareItemNode;
6.
7. final class JsonNodeTraverser
8. {
9.     /**
10.      * @var JsonNodeVisitorInterface[]
11.      */
12.     private array $jsonNodeVisitors = [];
13.
14.     public function addVisitor(
15.         JsonNodeVisitorInterface $jsonNodeVisitor
16.     ): void {
17.         $this->jsonNodeVisitors[] = $jsonNodeVisitor;
18.     }
19.
20.     /**
21.      * @param AbstractJsonNode[] $jsonNodes
22.      */
23.     public function traverse(array $jsonNodes): void
24.     {
25.         foreach ($jsonNodes as $jsonNode) {
26.             foreach ($this->jsonNodeVisitors as $visitor) {
27.                 $visitor->enterNode($jsonNode);
28.
29.                 // traverse all the children
30.                 if ($jsonNode instanceof ChildAwareItemNode) {
31.                     $this->traverse($jsonNode->subNodes);
32.                 }
33.             }
34.         }
35.     }
36. }
```

### Listing 14.

```
1. if (! isset($item['city'])) {
2.     return;
3. }
4.
5. $city = $item['city'];
6. if (! is_string($city)) {
7.     return;
8. }
9.
10. validate_city_name($city);
```

Now go have fun and play around. Do you need inspiration? Try it today in your CI to validate if your framework is on the latest X version for every package?

- `parse composer.json`
- look for “require” and “require-dev” sections
- there look for “/”
- validate the version to the one you're expecting
- all packages must have the same version or crash

## Listing 15.

```

1. namespace PhpArch\Ast\JsonNodeVisitor;
2.
3. use PhpArch\Ast\Contract\JsonNodeVisitorInterface;
4. use PhpArch\Ast\Node\AbstractJsonNode;
5. use PhpArch\Ast\Node\ItemNode;
6.
7. final class ValidateCityJsonNodeVisitor
8.     implements JsonNodeVisitorInterface
9. {
10.     public function enterNode(AbstractJsonNode $jsonNode)
11.     {
12.         if (!$jsonNode instanceof ItemNode) {
13.             return;
14.         }
15.
16.         if ($jsonNode->name !== 'city') {
17.             return;
18.         }
19.
20.         validate_city_name($jsonNode->value);
21.     }
22. }

```

That's it. I believe you can make it, and next time you face a "thousand files" changes or checks, you know how to deal with it quickly and effectively! You'll be the most favorite person on the team because you have the power to save so much time and boring work for your colleagues.

Happy coding!



*Tomas Votruba is a regular speaker at meetups and conferences and writes regularly. He created the Rector project and founded the PHP community in the Czech Republic in 2015. He loves meeting people from the PHP family so he created Friend-Of-PHP which is updated daily with meetups all over the world. Connect with him on twitter [@votrubaT](https://twitter.com/votrubaT)*

## Related Reading

- *Real World AST* by Tomas Votruba, September 2022. <https://phpa.me/real-world-ast>
- *Your AST Toolkit* by Tomas Votruba, November 2022. <https://phpa.me/ast-part2>

**Using Xdebug to squash bugs, identify bottlenecks, and boost productivity?**



Become a Pro or Business supporter to help ongoing development.

Supporters get help via email and elevated issue priority.

<https://xdebug.org/support>

[support@xdebug.org](mailto:support@xdebug.org)



# Bring Value To Your Code - Part 2

Dmitri Goosens

Last month I introduced you to the basics of Value Objects and demonstrated a series of very simple PHP examples. This month, let's have a closer look at the benefits of using them.

## Value Objects are Valid Objects


In last month's examples, we mostly focused on the properties of the Value Objects. We barely touched on one of their biggest advantages: when implemented properly, Value Objects are **always** valid objects. (See Listing 1)

### Listing 1.

```
1. class PhpArchEmailAddress
2. {
3.     public function __construct(public readonly string $value)
4.     {
5.         $regex = '.*@phparch\.com';
6.         if (preg_match("/$regex/m", $value) !== 1) {
7.             $txt = '<%=> is not a valid email address.';
8.             $message = sprintf($txt, $value);
9.             throw new InvalidPhpArchEmailAddress($message);
10.        }
11.
12.        $regex = 'ben\.ramsey@phparch\.com';
13.        if (preg_match("/$regex/m", $value) > 0) {
14.            $msg = "Ben? Ben Ramsey? Are you kidding me?";
15.            throw new ExasperatedEric($msg);
16.        }
17.
18.        $this->value = $value;
19.    }
20. }
```

One can apply all the required validation and filter rules directly in the constructor or in a factory\*\* before \*\*instantiating the Value Object. As a result, if the object can be initiated with the passed in attributes, it will be valid. Even better, it is auto-validated!

If a basic string is used to deal with such an email address, one must check its validity every time. With immutable Value Objects, one no longer needs to verify anything. Its sole existence proves that the value(s) it contains is valid. This does clean up our code drastically and makes it much easier to maintain and read.

 **Side note:** when the validation rules get a little more complicated, it's recommended to move them to a dedicated class that will be easier to maintain and, more importantly, to test.

So, yes, Value Objects are **ALWAYS** valid!

## Types

Since PHP 7.4, we have been blessed with properly typed properties in PHP. We can now benefit from all the advantages other strongly typed languages offer in terms of performance and security. Value Objects are perfect to extend our favorite programming language with rich and expressive types.

Compare both of these examples:

```
class PhpArchNewEditionNotifier
{
    // with a string
    public string $senderEmailAddress;
    // ...
}
```

and

```
class PhpArchNewEditionNotifier
{
    // with a Value Object
    public PhpArchEmailAddress $senderEmailAddress;
    // ...
}
```

Next to being much more efficient, remember one does not need to write/copy-paste validating lines every time the value is used. The second example is much more expressive as it clearly indicates what the \$senderEmailAddress should be.

## Specialized & Rich Models

Value Objects address one particular concern and, next to the values, can encapsulate rich and context-bound business logic.

This means one will not bundle things together if they don't belong to the same context. Instead of this: (See Listing 2)

### Listing 2.

```
1. class Price
2. {
3.     public function __construct(
4.         public readonly float $price,
5.         public readonly string $currency,
6.         public readonly float $vatRate,
7.         public readonly ?float $discountRate = null,
8.         public readonly ?float $discountAmount = null,
9.     )
10.    {
11.    }
12. }
```

One could move attributes that belong to the same context into dedicated Value Objects. For instance: (See Listing 3)

#### Listing 3.

```
1. class Price
2. {
3.     public function __construct(
4.         public readonly Money $price,
5.         public readonly VatRate $vatRate,
6.         public readonly Discount $discount,
7.     ) {
8.     }
9. }
```

In the example above, we are using the Money for PHP library (see <https://www.moneyphp.org/><sup>1</sup>). Next to dealing with the well-known float issues when dealing with money, it is also a remarkable example of a Value Object.

In short, it more or less functions like this (not in PHP8.1): (See Listing 4)

#### Listing 4.

```
1. class Currency
2. {
3.     public function __construct(
4.         private string $code,
5.     )
6. }
7.
8. class Money
9. {
10.     public function __construct(
11.         private int|string $amount,
12.         private Currency $currency,
13.     ) {
14.     }
15. }
```

\$amounts are in cents and are, in the end, converted to a string to be able to store values that are greater than the system's integer limit (PHP\_INT\_MAX). Check out Money's source code to better understand the \$amount attribute.

The VatRate is a rather simple Value Object:

```
class VatRate
{
    public function __construct(
        public readonly float $rate,
    ) {
    }
    // ...
}
```

Interestingly enough, one could add a nice method to the VatRate Value Object to calculate the VAT and to apply the VAT, making the model richer: (See Listing 5)

#### Listing 5.

```
1. class VatRate
2. {
3.     // ...
4.
5.     public function calculateVat(Money $money): Money
6.     {
7.         return $money->multiply($this->rate);
8.     }
9.
10.    public function applyVat(Money $money): Money
11.    {
12.        return $money->add($this->calculateVat($money));
13.    }
14.
15.    // Money being immutable, the add() & multiply()
16.    // methods will return new instances of Money
17. }
```

Last but not least is the Discount Value Object. One may have wondered why, in the first example, \$discountRate and \$discountAmount were both nullable. That is because the system should be able to deal with rates and fixed amounts for the discount.

This probably is a smarter approach: (See Listing 6 on the next page)

Finally, one can easily figure out the total price:

```
class Price
{
    // ...
    public function total(): Money
    {
        return $this->vatRate->applyVat(
            $this->discount->applyDiscount($this->price)
        );
    }
}
```

This is just a simple example, but it illustrates that Value Objects are very specialized objects and can encapsulate very context-bound business logic.

## Testing

As we are testing the code, we write the code that addresses a written test (see TDD), testing just got much more straightforward.

If we were to use a regular string based email address as an attribute in multiple functions within multiple classes, we would need to write tests with valid and invalid email addresses for each one of them. With an EmailAddress Value Object, we only need to write those tests once, and we'll no longer have to worry about it anywhere else. As mentioned above, if a Value Object can be instantiated, it must be valid so we can ignore the validity checks when testing the rest of our codebase.

Last but not least, there is no need to mock Value Objects in our tests.

<sup>1</sup> <https://www.moneyphp.org/>

## Listing 6.

```

1. interface Discount
2. {
3.     public function asRate(Money $money): float;
4.
5.     public function asMoney(Money $money): Money;
6.
7.     public function applyDiscount(Money $money): Money;
8. }
9.
10. class RateDiscount implements Discount
11. {
12.     public function __construct(
13.         public readonly float $rate,
14.     )
15.
16.     public function asRate(Money $money): float
17.     {
18.         return $this->rate;
19.     }
20.
21.     public function asMoney(Money $money): Money
22.     {
23.         return $money->multiply($this->rate);
24.     }
25.
26.     public function applyDiscount(Money $money): Money
27.     {
28.         return $money->subtract($this->asMoney());
29.     }
30. }
31.
32. class FixedAmountDiscount implements Discount
33. {
34.     public function __construct(
35.         public readonly Money $amount
36.     )
37.     {
38.     }
39.
40.     public function asRate(Money $money): float
41.     {
42.         return $this->amount->getAmount() / $money->getAmount();
43.     }
44.
45.     public function asMoney(Money $money): Money
46.     {
47.         return $this->amount;
48.     }
49.
50.     public function applyDiscount(Money $money): Money
51.     {
52.         return $money->subtract($this->asMoney());
53.     }
54. }

```

## Cognitive Load

As the usage of Value Objects encourages you to encapsulate the context-bound business logic along with it, one will write more classes but much smaller ones in the end.

As a result, just like in DDD, where the business logic is split over multiple insulated contexts, the cognitive load required to understand what a given object does, how it works, and how to interact with it gets reduced... Even more now that one knows it cannot change all of a sudden and that it will *de facto* always be valid.

## Caching

Value Objects become perfect for caching because of their immutability, even for a very long period of time.

This can be done either with an application-wide caching mechanism: (See Listing 7)

## Listing 7.

```

1. $psr6CachePool = new Cache();
2.
3. $cache_key = PhpArchEmailAddress::class . $emailAsString;
4. if(!$psr6CachePool->hasItem($cache_key)) {
5.     $email = new PhpArchEmailAddress($emailAsString);
6.
7.     $psr6CachePool->save(
8.         $psr6CachePool->getItem($cache_key)
9.         ->set($email)
10.    );
11. }
12.
13. $emailAddress = $psr6CachePool->getItem($cache_key)->get();

```

or through memoization, storing the result locally and using when needed:

## Listing 8.

```

1. class ComplicatedMathematicalValueObject
2. {
3.     private ?float $result = null;
4.
5.     public function __construct(
6.         public readonly float $val1,
7.         public readonly float $val2,
8.     )
9.     {
10.    }
11.
12.     public function complicatedOperation(): float
13.     {
14.         if (is_null($this->result)) {
15.             $this->result = // very long operation goes here
16.         }
17.         return $this->result;
18.     }
19. }

```

Bear in mind that when using memoization, one loses the ability to use the == operator to compare two Value Objects.



For the == operator, the Value Object before and after memoization will no longer be equal.

```
$var1 = new ComplicatedMathematicalValueObject(1.2, 1.3);
$var2 = new ComplicatedMathematicalValueObject(1.2, 1.3);

var_dump($var1 == $var2); // true

$var1->complicatedOperation();

var_dump($var1 == $var2); // false after memoization
```

Therefore, it is very common to add an isEqual() method to the Value Objects to deal with this. (See Listing 9)

#### Listing 9.

```
1. class ComplicatedMathematicalValueObject
2. {
3.     // ...
4.
5.     public function isEqual(
6.         ComplicatedMathematicalValueObject $other
7.     ): bool {
8.         return $this->val1 == $other->val1 &&
9.             $this->val2 == $other->val2;
10.    }
11. }
12.
13. $var1 = new ComplicatedMathematicalValueObject(1.2, 1.3);
14. $var2 = new ComplicatedMathematicalValueObject(1.2, 1.3);
15.
16. var_dump($var1->isEqual($var2)); // true
17.
18. $var1->complicatedOperation();
19.
20. var_dump($var1->isEqual($var2)); // true
```

By the way, this isEqual() method will also deal with inheritance issues:

This makes sense; the == operator also expects the objects to be of the exact same type (See Listing 10). Sadly, it is not possible yet, to overload PHP's operators.

Luckily, the isEqual() method will deal with this: (See Listing 11)

Remember that we care about the values here. It is obvious that a MustacheColor and a Color are not identical, yet their values are.

## Caveats

### And What About Dtos?

As Matthias Noback wrote in a blog post lately<sup>2</sup>, DTOs and Value Objects are not the same.

First of all, they have different purposes. DTO, or Data Transfer Objects, are used on the boundaries of your

#### Listing 10.

```
1. class MustacheColor extends Color
2. {
3.
4. }
5.
6. $colorA = new Color(255, 255, 255, 1);
7. $colorB = new MustacheColor(255, 255, 255, 1);
8.
9. var_dump($colorA == $colorB); // false
```

#### Listing 11.

```
1. class Color
2. {
3.     // ...
4.     public function isEqual(Color $otherColor): bool
5.     {
6.         return $this->red == $otherColor->red
7.             && $this->green == $otherColor->green
8.             && $this->blue == $otherColor->blue
9.             && $this->alpha == $otherColor->alpha;
10.    }
11. }
12.
13. $colorA = new Color(255, 255, 255, 1);
14. $colorB = new MustacheColor(255, 255, 255, 1);
15.
16. var_dump($colorA->isEqual($colorB)); // true
```

application, either when data enters it or leaves it. On the other hand, value objects are mostly used within the application and even more often within its business layer.

Secondly, even though DTOs and Value Objects define a structure for data, DTOs do not need to be complete as required for a Value Object (see Whole Value above). DTO's properties can be nullable, and it is not the DTO's responsibility to validate or only accept valid data or state.

Thirdly, as indicated above, Value Objects ought to be specialized, only deal with a specific concern, and eventually include context-bound business logic. DTOs, on the other hand, do not require limitation to a specific scope. They need to make data interchangeable, able to go in/out of an application, and it should be formatted according to its usage. Also, DTOs will barely ever include any complex business logic.

### Persistence

One of the biggest challenges when working with Value Objects is clearly persistence, *i.e.*, storing it in a database or wherever. The complication mostly comes from the absence of an identifier. This mind shift requires a little effort as we are used to mirroring our database design with our code or the other way around.

This is another principle from DDD, where the code used to express the business layer is to be completely decoupled from the persistence layer, for instance, the database. PHP

2 Matthias NOBACK, *Is it a DTO or a Value Object?*

Value Objects are only useful in a PHP context, not in a database... at least not as such. So it makes perfect sense to break the sacred rule of one model/table and one property/column!

An unpopular point of view, maybe: the database is the place where one stores data, NOT where one consumes it. To use the data outside of the application, either one ought to use the application to fetch it – for instance, through a dedicated API that uses the application's business logic – or build some kind of data warehouse where the data is transformed and can be optimized for that specific task.

However, there are multiple ways to store these Value Objects in the database. There will always be a suitable solution:

### Along with the Encapsulating Entity

Remember, and this is important, your database does not have to be an exact reflection of your code (and *vice versa*).

For example, (See Listing 12)

#### Listing 12.

```
1. // Entity
2. class Book
3. {
4.     public readonly Uuid $uid;
5.     private Author $author;
6.     private Title $title;
7.     private ?Color $coverColor;
8.     // ...
9. }
```

could simply be stored in a table that would look like this:

```
create table book
(
    uid            varchar(36) not null,
    author_uid     varchar(36) not null,
    title          varchar(255) not null,
    cover_color_rgba char(8)    null,
    -- ...
    foreign key (author_uid) references author(uid)
    -- ...
);
```

The Author being another entity, with an identity, the book table contains a foreign key to the author table.

The Title Value Object is directly stored as string.

And in the above example, the Color is stored in its RGBA notation (and this is explicitly indicated) even though the original Color Value Object uses three ints and a float. It is shorter and easier to store in a single field this way.

Depending on the chosen technology for the communication with the database (raw SQL, PDO, ORM, REST...), this communication layer will be responsible for converting the data in the database into valid PHP objects and the other way around. It is highly recommended to look closely at the Repository pattern to deal with this.

### Denormalization

Much has been said about how to avoid repetition in code and why we should do so. Maybe a little too much. We also see this enforced in databases where one uses foreign keys to reference data reused elsewhere.

Yet, denormalization probably is one of the most interesting approaches in the case of Value Objects, especially for small ones. Denormalization is the exact opposite of “avoiding repetition”, except one does it on purpose.

Please note that unlike what one may think, if done properly, this might improve the database's performance as additional joins are no longer required.

Consider this code:

```
// Entity
class Book
{
    // ...
    private ?Color $coverColor;
    private ?Money $price;
    // ...
}
```

If one were to mimic the code's structure and did not want the RGBA Color notation, this would result in database tables that could look more or less like these:

#### Listing 13.

```
1. create table color
2. (
3.     id            int auto_increment primary key,
4.     color_red     int      not null,
5.     color_green   int      not null,
6.     color_blue    int      not null,
7.     color_alpha   float    not null,
8. )
9.
10. create table price
11. (
12.     id            int auto_increment primary key,
13.     amount        char(10)  null,
14.     price_currency char(3)   null,
15. )
16.
17. create table book
18. (
19.     -- ...
20.     cover_color_id int      null,
21.     price_id       int      null,
22.     -- ...
23.     foreign key (cover_color_id) references color(id)
24.     foreign key (price_id) references price(id)
25.     -- ...
26. )
```

Color and Price would have an identity, and foreign keys would be created to reference the \$bookCoverColor and \$price.

If we were to add another concept with a price or a color, we would be able to use the same color and price tables. This would require multiple joins to fetch all the book's attributes.

But it could also be stored in a denormalized way, in one single database table that would look like this:

Listing 14.

```
1. create table book
2. (
3.     -- ...
4.     color_red          int          null,
5.     color_green        int          null,
6.     color_blue         int          null,
7.     color_alpha        float        null,
8.     price_amount       int          null,
9.     price_currency     char(3)      null,
10.    -- ...
11. );
```

In this example, the Color Value Object is stored directly in four fields, one per channel, in the book table. Depending on database usage and personal preferences, one could choose this version or the one in the previous code listing using the RGBA notation.

The Money \$price Value Object is stored in two distinct columns, one for the amount and another for the currency.

**Side-note:** even though the \$amount\* in a Money Value Object really is a **string**, in the example, it is stored as an **int**. The thing is, a book's price will never exceed `PHP_INT_MAX`, and the data department will be happy the data is immediately usable and will not need to be cast.\*

This does mean, however, that other concepts with a color and/or price would also need to implement these columns. But is that really a problem? Not really...

## Serialization

One of the easiest ways is to serialize the data and store it in one single field in the database, preferably in JSON format (not with PHP's `serialize()` function).

Remember that, these days, most of the RDBMS can deal directly with JSON. So if one really needs to access the data in the database directly, even though it will be a little slower, it still is possible.

A Color field would look like this:

```
{"red" : 255, "green" : 0, "blue" : 0, "alpha" : 1}
```

Readable, exploitable, single field... Definitely an option to bear in mind.

## As an Entity

Last but not least, in some cases, it does make sense to consider a Value Object as an entity when storing it (and only when storing it). However, this should only be considered when other options are not feasible.

There are multiple options here; all of them are right and really depend on how much one wants to decouple the persistence layer from the Value Objects.

The first option, the easiest one, is to autogenerate a unique identifier when instantiating the Values Object. This property should be private and not accessible via the object's public API (when using an ORM like Doctrine, you actually don't need any setter or getter methods).

Listing 15.

```
1. class Color
2. {
3.     private int $id;
4.
5.     public function __construct(
6.         public readonly int $red,
7.         public readonly int $green,
8.         public readonly int $blue,
9.         public readonly float $alpha,
10.    ) {
11.    }
12. }
```

The second option is the one used in the DDD-world and makes use of a surrogate identifier. This is achieved by using a parent class with a private or protected property.

Listing 16.

```
1. abstract class ValueObjectWithId
2. {
3.     private int $id;
4. }
5.
6. class Color extends ValueObjectWithId
7. {
8.     public function __construct(
9.         public readonly int $red,
10.        public readonly int $green,
11.        public readonly int $blue,
12.        public readonly float $alpha,
13.    ) {
14.    }
15. }
```

Obviously, this could also be achieved with a trait: (See Listing 17)

The id generation would be delegated to the database in all three cases. ORM systems, like Doctrine, can hydrate the private properties, but this could also be achieved with



## Listing 17.

```

1. trait ValueObjectWithId
2. {
3.     private int $id;
4. }
5.
6. class Color
7. {
8.     use ValueObjectWithId;
9.
10.    // ...
11. }
```

Reflection or by binding a closure to the class (see Marco Pivetta's, aka [@ocramius](#), [blog post here](#)<sup>3</sup>).

This approach requires more SQL queries, though. When saving the Value Object to the database, one will have to check if an entry with the same properties already exists and inject the private id that will be used later on in the encapsulating entities or Value Objects.

## Dedicated Persistence Types

ORMs like Doctrine and Eloquent allow us to create dedicated adapters that take care of the conversion of DataTypes, in this case, the Value Objects, from the application's PHP into the database and the other way around.

One still has to decide which of the above strategies is the most suited... But once implemented, these auto-magic transformers come in quite handy.

## Enums

As PHP 8.1 introduced the long-awaited native enums, adding a little note on those will be beneficial. It seems obvious to store a PHP Enum as an Enum in a database... The thing is, this may not be the best move.

Back in 2011 (yes, that's a long time ago), Chris Komlenic wrote a little [blog post](#) listing a series of issues with the MySQL Enum type<sup>4</sup>. As MySQL still is one of the most popular RDBMS, this list of objections remains pretty much valid. In PostgreSQL, Enums are real, reusable data types, so part of the objections are less valid for that RDBMS.

While database Enums are easy to use, they come at a cost and, in the end, offer only negligible benefits on optimization (an in-depth analysis of this would be out of the scope of the current article).

In the end, and especially when working with an ORM, it is really not complicated to use one of the strategies mentioned above and avoid the "evil" native database enum type.

## Some Interesting Value Objects

We already mentioned the `DateTimeImmutable` object, looked at the `Money` Value Object in the examples above,

and also mentioned its benefits over using float to deal with monetary values.

Here are some more Value Objects that might be of interest:

### Uuid

Ben Ramsey created PHP's best implementation of Uuid (<https://uuid.ramsey.dev/en/stable/index.html>). By the way, Ben just added version 7 of the Uuid specification. Make sure to check it out.

### Not Null

In his brilliant blog post "Much ado about null", Larry Garfield shows a series of very interesting alternatives to using null. Check out the Maybe [Monad] Value Object and, even more, his Result proposal at the end of the post (<https://phpa.me/peakd-hive-null>).

### Symfony/string

The Symfony/String component is another very nice implementation of a "big" Value Object that handles strings as immutables and provides rich functionality (<https://phpa.me/symfony-string>).

## Conclusion

We now know what a Value Object is, that it is one fundamental tool when working in Domain Driven Design, but it definitely is a very interesting approach anywhere. We've briefly discussed Value Object's characteristics, benefits, and ways to use and store them.

Domain Driven Design is gaining a lot of traction within the PHP community. Showing that our favorite programming language has matured a lot over the last years, and chances are Value Objects and other DDD concepts will become more and more used, even when not working on a DDD project.

Another building block to bring value to our code...



*Dmitri is a Belgian backend PHP Dev and Architect, working at Ardennes-étape, who is very passionate about DDD, OpenSource and everything Digital. When he's not working, playing with code or reading about code, he spends time with family and cats and enjoys Belgian beers (especially Orval) and going for long walks in the forest.*  
@<https://twitter.com/dgoosens>

## Related Reading

- *Bring Value To Your Code* by Dmitri Goosens, November 2022.  
<https://phpa.me/2022-11-value>
- *Security Corner: Self-obfuscating Value Objects-A Design Pattern for PII* by Eric Mann, November 2020.  
<https://phpa.me/security-nov-2020>

<sup>3</sup> <https://phpa.me/access-private-class>

<sup>4</sup> <https://phpa.me/mysqls-enum-data-evil>



# Refactoring Yourself

Chris Tankersley

Ever since I started working on the Education Station column, one of my goals has been to ensure that the column's content revolves around the things I think all developers should know. This means that useful coding topics like design patterns and dependency injection can sit alongside useful theory articles like how HTTP works or suggestions and descriptions of how APIs work. These technical topics are ones that I think developers can find useful in a web-centric language.

Some of the softer skills are one set of topics that don't get as much time. While there have been some lighter topics like licensing or code reviews, one topic I've avoided for a while is advancing in your career. After a recent trip to Longhorn PHP, I think it is finally time to tackle the subject. I will also wade into the eternal argument of Craftsperson versus Engineer. If you have skimmed ahead and read the titles, I'm sure you can tell which way I lean. If you have not, I will go ahead and tell you. We are Craftspeople. We make things, sometimes beautiful, complex, and wonderful, but we are creators. We do not currently have the discipline as an industry to consider ourselves engineers. Yes, some of the things we build are life-saving, and we should be striving to make secure, safe, structurally sound applications. We should be building things with an engineering mindset. At the end of the day, however, we act, work, and deliver more like a craftsperson than an engineer. With that in mind, we should look at ourselves as craftspeople. That separates us until Apprentices, Journeymen, and Master Craftsmen. For clarity and to ease additional research, if you want to dig deeper into these ideas, I will use the gendered form of these titles. It is not my intention to diminish the work of people of other genders.

## Titles are Meaningless

One thing I want to get out of the way is that you are not defined by the title that your company gives you. One company's junior developer is another company's mid-level developer. As you

enter the later stages of your career, what is the difference between an Architect and a Staff Engineer? Is there one?

Titles are a way for companies to categorize employees in terms of hierarchy and reporting. Many also use them as loose guides for pay, but the pay can, unfortunately, range drastically from employee to employee in the same title. This gets worse as companies become more remote and start to institute regional pay banding, or scaling pay, with what they assume is a standard cost of living (for example, a developer in San Francisco will get paid more than a developer doing the same job in the Midwest, because the San Francisco employee has a higher cost of living).

I want you to think less about what title you have right now, and I want to focus on what I feel describes the various levels you can be in terms of skill. If you are honest with yourself about your skill level, you can better plan to grow your skills and experience and be happier with the jobs that you select.

I think it is also important to note that your skill level is, and should not be, directly tied to a length of time. Just because you have been working as a developer for three years does not necessarily mean that you are at a certain level. Conversely, just because you finished a boot camp does not confer any additional advancement.

You are at the skill level you are at, and that is fine! The goal this month is to learn where you are at, and what I think it takes to grow as a developer. No matter where you are currently in your career there is always room to grow.

## Your Brain Lies to You

If I am going to ask you to be honest with yourself about your skill level, I need to go ahead and give a quick talk about the two ways that your brain will lie to you about your career. Our brains are great at filling in information and making things up, and our brains will do the same regarding skill level.

All people in the tech industry will suffer from two things—the Dunning-Kruger effect<sup>1</sup> and Imposter Syndrome<sup>2</sup>. I say 'all' because we all have both over and underestimated our abilities when it comes to projects. There is nothing wrong with us when either effect happens, but we must be aware of them.

The Dunning-Kruger effect is a bias of someone overestimating their ability when, in reality, they are not very good at something. This does not mean someone is "stupid" or "incompetent", but it manifests when someone thinks they are better at something than their abilities demonstrate because they do not understand their deficiencies.

I am sure we have all talked to someone else who touts their abilities, but then the examples they give fall a bit flat. A common one I hear is that a developer understands what it takes to have a highly available site that can withstand heavy traffic surges. And then you find out that the person did this by asking their shared hosting provider for a larger package and has no applicable knowledge of load balancers, clustering, geographical separation, or caching.

<sup>1</sup> Dunning-Kruger effect:  
<https://phpa.me/wikipedia-kruger-effect>

<sup>2</sup> Imposter Syndrome:  
<https://phpa.me/wikipedia-impostor-syndrome>



They have a bias toward thinking they can handle the same problem elsewhere.

Imposter Syndrome is the opposite. Imposter Syndrome leads people to think they lack useful, applicable skills when it turns out they are more than capable of handling a situation. I see this happen when someone goes to pick up a new language or framework. They see people online pushing a new awesome technology, such as Vite<sup>3</sup>, that purports to make development easier. When they try to use it, they might hit a roadblock and think they are not as good of developers as they thought. If others can make this work, why can't I?

These are both somewhat simplifications of Dunning-Kruger and Imposter Syndrome, but when evaluating yourself, keep both of these things in mind. You may think that you are not as good as other developers just because you have been doing only WordPress for five years. That may not be true; you may have built up a lot of useful skills in the WordPress ecosystem that few acknowledge, and those skills may be transferrable as you learn something new. Or conversely, you may be a Laravel developer who sneers at that WordPress developer, thinking that WordPress is not that hard. Eight hours later, you cannot figure out how to properly return a JSON response.

Try to think about your skill level objectively, and ask yourself if you are selling yourself short or overestimating your abilities. Be honest with yourself, and remember that just because you can or cannot do something, it's OK either way. You can always capitalize on an existing skill or learn something new, but you will only advance if you know where you are and what to grow.

## Apprentice

Apprentices are entry-level craftspeople. An apprentice learns the trade via hands-on work and can also include in-classroom training. In many cases, an apprentice is also paired with a trained professional to help them learn the ins and outs of the job. The

expectation of an apprentice is for them to learn and grow.

One of the key defining factors of an apprentice is the need to learn. I would expect anyone at this career level to constantly ask questions about how things work and to need a lot of direction in performing tasks. An apprentice needs to be shown how to navigate the problem, and in many cases, the solution.

An apprentice's key skill at this stage is learning. It is not problem-solving, it is not code quality, and it is not the amount of technical knowledge they come into the role with. This is not a bad thing, as the entire point of being an apprentice is to learn. A good apprentice is someone who wants to learn. They are okay asking questions, and they are fine taking directions.

An apprentice-level developer should be given tasks that are fleshed out and have a clear goal. An apprentice should ask questions along the way about anything that is not clear and should ask for help when they are stuck. They need a support structure to allow them the freedom to make simple mistakes, but that support structure should guide them along the way.

An apprentice should not be required to come up with a solution on their own as they need more job knowledge to properly implement a solution efficiently. While the tasks may involve some level of general problem-solving, the solutions that an apprentice comes up with are not expected to be novel. Apprentices are best served by showing them the way and allowing them to reproduce what they have learned.

If we put this in terms of developers, an apprentice developer should be given clear, concise, and easy-to-implement tickets. They should have access to more senior developers to ask questions and learn, and in the best cases, be able to participate in things like pair programming to see how others work. Code reviews with them should be as insightful as possible to the corrections needed.

Any company hiring an apprentice-level developer needs to have a

good way for these apprentices to be mentored; otherwise, it will be incredibly hard to wait for someone to build themselves up. If you expect an apprentice to learn on their own with no direction, there is a good chance they may waste time learning things that are not beneficial. They may stagnate as they struggle to learn on their own, or worse, burn out, unable to do their job adequately.

I am a big fan of user groups and building a good network of people to help out with questions, and there is no reason a company should not provide the same for apprentices. If you are hiring an apprentice developer, have that support structure in place to help them grow. In the long run, you and they will be better off for it.

Unlike traditional apprenticeships, there is no clear way to know when you have crossed over to the next level, Journeyman. It will happen over time as you grow more confident in your ability to come up with solutions and you are more confident in your ability to perform the work. You begin to ask fewer and fewer basic questions. Questions themselves are okay, but over time you should find yourself asking fewer questions about common tasks and questioning more about intent and implementations.

Spend your time learning the core tasks that your job entails. If that is doing WordPress work, find out the best ways to work in WordPress. Join a local user group. Participate in Stack-Overflow. If it is Laravel, absorb what you can from LaraCasts, go to Laracasts, and find out what people are doing. Ask your teammates for help and guidance. It does not matter what the core competency of your job is; find a way to learn. Your job is to learn.

## Journeyman

The next level up is called Journeyman. The name comes from the French "journée", meaning "day." It meant that the person was competent enough to do a day's worth of work and be able to charge for said work. They had completed all the work to earn

3 Vite: <https://vitejs.dev>





their license and had the skills needed to work, but they were not yet considered on the same level as someone who could be self-employed.

That last sentence is one of the defining factors between a journeyman and a master craftsman. Journeymen still require some oversight, but their experience allows them to work on more complicated tasks with less oversight. They can even be good enough to start mentoring apprentices, as journeymen have enough institutional knowledge to start showing others the ropes.

For developers, this means that when given a task, questions will tend to be more about the specific task at hand or goals, and will move away from more basic questions. For example, adding a new feature to a site may garner questions like “What’s the expected output?” or “Does this require new restrictions?” and less “Does this go in a controller?” or “How should I set up this model?”

Journeymen will also be able to draw on more experience than an apprentice as they have touched more projects. They begin to notice patterns in problem-solving, and can more easily equate a new problem to an older one to infer solutions. They are more confident in the tools that they learned during the apprenticeship and using them for new solutions.

If we go back to the WordPress example, a journeyman will begin to form their preference for structuring plugins in a way that works for them, informed by the work they have done before. Using the best practices they learned from others, they start to develop a familiarity with how things work to the point where boilerplate slowly becomes second nature. Sure, they might still look things up, but they begin to learn what is easy to find versus what is easier to remember.

The work that a journeyman developer does will start to get more complex, and there will be greater freedom to come up with solutions. That does mean that there will be a greater responsibility for the outcome of a project, but there will be greater freedom to influence the

final product. You will still be learning new things, but alongside learning new things, you are finding new ways to use existing skills.

The Journeyman programmer will still be taking queues on what is considered important work from more senior developers, even though they have more latitude to work freely. They may be called into some technical discussions, but Journeyman will be utilized for their ability to program and find a solution much more than someone who is called to come up with longer-term designs or plans.

Moving on from a Journeyman to a Master Craftsman is a very blurry line. As your career advances toward being a Master Craftsman, you will start to notice that not only will your programming and problem-solving skills be sought out, but you will also be brought into more design or product decisions. You may spend more time with non-coding tasks not because of your raw programming ability, but because your breadth of experience is considered valuable to others to help inform their own decisions.

## Master Craftsman

Master Craftsman was considered the highest level that a tradesperson could achieve. Originally, they would have needed to show that they could produce a large amount of money and create a suitable masterpiece. The masterpiece would need to be approved by other masters before an aspiring journeyman would be granted the title of Master.

While we no longer have to create a masterpiece, a modern master craftsman is someone that is respected amongst their peers. Few people question the technical ability of a master craftsman and instead seek them out for their guidance. From a technical perspective, master craftsmen will start to spend much more of their time on non-programming tasks as more people utilize them for their knowledge.

Oddly enough, a master craftsman may find it harder to continue to grow in skills, not because of a lack of talent

but because of a lack of time. Programming tasks tend to be much larger in scope when the need arises, but many find themselves falling into specific niches. The book “Staff Engineer” by Will Larson breaks these niches down into four types:

- The Tech Lead - You are responsible for leading a particular team and working closely with managers to understand the needs of the business. Much of the work is delegated to members of the team, while you spend time organizing and making sure the team moves forward.
- The Architect - You are responsible for designing, working on, and maintaining critical areas. Your knowledge allows you to make key decisions across many areas of a project.
- The Solver - You spend your time working on and solving complex problems. You may jump from project to project as needed, or spend a lot of time on one complex project.
- The Right Hand - You help superiors and executives by being able to take on part of their load as well as being able to have additional authority by adding additional bandwidth to leadership.

“Staff Engineer”, also references an article by Tanya Reilly, where developers at this level act as the glue for an organization. Much of the work you do will be to help move things forward and may be unglamorous, but it is work that is needed to ensure a successful project. Additional meetings, preparing slide decks, and working closely with more people across the organization is key.

You will need to grow your organizational and people skills as you move toward being a master and as you continue at this level of your career. These skills will be just as important as the technical skills that you have spent time learning and refining. Many will find this to be the hardest part of the



job as sometimes people will be much harder to handle than a good technical problem.

## Some Things Never Change

You will always be learning, no matter where you are in your career. Apprentices, journeymen, and Master craftsmen will spend a large amount of time learning new skills, both technical and soft. You should always be a part of the mentoring loop, starting off learning as a mentee and later helping transfer knowledge as a mentor. You will constantly be adding new tools to your toolbox to help you solve problems.

If you want to grow in your career, you will need to overcome your biases and your self-doubt. You will want to decide if you need to work on your technical skills, increase your experience, or find better ways to work with people. The

transitions will not be quick, and you may spend much of your career floating at the journeyman level. That is okay.

Keep your personal goals in mind, and find the areas where you can grow. Find a good mentor either at work or in the wider PHP community. Do what works for you to meet those goals, and be happy as your career grows.



*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](#)*



### You're the Team Lead—Now What?

Whether you're a seasoned lead developer or have just been “promoted” to the role, this collection can help you nurture an expert programming team within your organization.

After reading this book, you'll understand what processes work for managing the tasks needed to turn a new feature or bug into deployable code.

**Order Your Copy**  
<https://phpa.me/devlead-book>



# Sticker Swapping

Oscar Merida

The FIFA World Cup wraps up this month. One thing many fans worldwide do in the months leading up to the tournament is to fill out sticker albums with each team's roster. Completing an album, short of spending a ton of excess money, involves trading with other collectors. A common technical interview question is to find the common characters in a string or the common elements in multiple arrays. Let's take a look at how to approach such a problem.

## Recap

Stickers come in packs of 5; eventually, you should be trading stickers with others to get the ones you need. To trade, everyone can generate a list of needed stickers and duplicates. Each sticker has a three-letter code and a number between 1 and 20. One person may list the ones they need in any format like this:

```
FWC 2, 3, 7 9, 13, qat 5, qat 6 10 qat 11, 13, 14, qat 15,
qat 18, 19, ECU 1 2 3, 5, 6, 10, 13 19, SEN 7, 8, sen 11,
sen 19, NED 2, 3, 13, ENG 7, ENG 10, ENG 19,
IRN 2, 3, 5, 7, 10 12 17 19, USA 2, USA 12, USA 14 15 16,
USA 17, USA 19, WAL 2, WAL 5, WAL 16, ARG 4 8 9 10
Arg 11, Arg 14, Arg 18 , ECU 14 15, ECU 18 Ned 17
```

Similarly, a list of duplicates to trade might look like:

```
00, FWC 1,4, FWC 12, FWC 17, QAT 1, QAT 3,4 ECU 18 19
GER 2,8, 10, 13,14, 15, JPN 3, JPN 10, JPN 14 15 19,
JPN 20, BEL 3 9 12 14 15 16 19, Can 5,7,17,19, USA 4,
MAR 4, MAR 14, MAR 16, SEN 11, SEN 19, NED 2,3,13,ger 20,
CRO 6, CRO 18, BRA 1, BRA 8, BRA 13, SRB 5,10 14, SRB 16,
SRB 18, 20, SUI 4, SUI 6, SUI 9, SUI 11, SUI 15, SUI 20,
CMR 7 9,11,13,15,16, POR 3, 5, 8, POR 10,11,12, WAL 16
por 15,16, 18, Wal 2,5, USA 15,16,17
```

*Given two separate user-provided text lists like the two above, write a program that will compare both sets of text and return the stickers in the duplicates list that are also in the list of needed ones.*

## Planning an Approach

My immediate instinct is to get the data into arrays we can compare. PHP has plenty of functions we can use to do so. But the input data for either a list of wants or a list of needed stickers is not presented in a standard format. When working with input, particularly if it's user-generated and out of our control, we should clean it up as much as possible. Doing so will help us later by reducing complexity and checking for correctness.

I won't do so in this exercise, but we could check that the country names submitted are valid against an acceptance list of teams participating in the tournament. Likewise, we could check that the numbers in each list are in our expected range of 1-20.

To get a working solution, I was initially tempted to explode on commas and maybe spaces, but neither is used consistently. Time to take a step back. Let's look at what we have to deal with:

1. Country abbreviations may be in upper or lowercase
2. Sometimes the country is listed once with multiple numbers following it.
3. Items are separated by spaces, commas, or comma+space.

Luckily, it looks like we can count on lines starting with a country name and not worry about numbers wrapping. I think our first step is to clean up the incoming data. A straightforward `cleanInput()` function converts everything to uppercase, gets rid of meaningless commas, and eliminates sequential spaces.

```
function cleanInput(string $in): string
{
    $out = strtoupper($in);
    $out = str_replace(',', ' ', $out);
    return preg_replace('/\s+/', ' ', $out);
}
```

I discovered an unexpected benefit of using `preg_replace()` to replace one or more whitespace characters with a space. Since newlines and line breaks are whitespace characters, replacing them joins multiple lines of input into a single line. I think that'll be useful later, so I will keep it. Using it, we get a string that looks like the following:

```
FWC 2 3 7 9 13 QAT 5 QAT 6 10 QAT 11 13 14 QAT 15 QAT 18 19
```



## Building Arrays with Regex

Next, we need to take our cleaned-up string and turn it into an array where each element is a string of three-letter country code followed by a number. We could explode our string on spaces and then walk through each element, figure out if we have a country or number, and handle it appropriately. But I think I can use a regular expression to chunk the string into smaller strings with country + all subsequent numbers. A couple of minutes on <https://regex101.com><sup>1</sup>, yields the following:

```
/([A-Z]+\s([0-9 ]+))/
```

The expression above returns a country match in the first capture group, eg FWC, and then all the numbers that follow it, like 2 3 7 9 13. We can use `preg_match_all()` to grab all such “matching pairs” to build our arrays. (See Listing 1)

### Listing 1.

```
1. <?php
2.
3. function buildArray(string $input): array
4. {
5.     preg_match_all('/([A-Z]+\s([0-9 ]+))/', $input, $match);
6.     $result = [];
7.     foreach ($match[0] as $i => $m) {
8.         $country = $match[1][$i];
9.         $numbers = explode(' ', trim($match[2][$i]));
10.         foreach ($numbers as $number) {
11.             $result[] = $country . ' ' . $number;
12.         }
13.     }
14.     sort($result, SORT_NATURAL);
15.     return $result;
16. }
```

### Listing 2.

```
1. [1] => Array
2. (
3.     [0] => FWC
4.     [1] => QAT
5.     [2] => QAT
6.     [3] => QAT
7.     [4] => QAT
8.     [5] => QAT
9.     ...
10. [2] => Array
11. (
12.     [0] => 2 3 7 9 13
13.     [1] => 5
14.     [2] => 6 10
15.     [3] => 11 13 14
16.     [4] => 15
17.     [5] => 18 19
```

The function above uses the regular expression to give two capture groups via the arrays in `$match`. `$match[1]` holds the country abbreviations and the sticker numbers are in the same index of `$match[2]`. Each looks like the output below: (See Listing 2)

I used a nested loop to then combine both and sort them to get output like:

```
Array
(
    [0] => ARG 4
    [1] => ARG 8
    [2] => ARG 9
    [3] => ARG 10
    [4] => ARG 11
    [5] => ARG 14
)
```

Since we don't need two separate functions to clean and process our input, we can replace it with the function shown in Listing 3.

### Listing 3.

```
1. <?php
2.
3. /**
4.  * @return string[]
5.  */
6. function toStickerList(string $input): array
7. {
8.     // clean input
9.     $input = strtoupper($input);
10.    $input = str_replace(
11.        ',', ' ', $input
12.    );
13.    $input = preg_replace(
14.        '/\s+/', ' ', $input
15.    );
16.
17.    // make country+number array
18.    preg_match_all(
19.        '/([A-Z]+\s([0-9 ]+))/',
20.        $input,
21.        $match
22.    );
23.    $result = [];
24.    foreach ($match[0] as $i => $m) {
25.        $country = $match[1][$i];
26.        $numbers = explode(
27.            ' ',
28.            trim($match[2][$i])
29.        );
30.        foreach ($numbers as $number) {
31.            $result[] = $country . ' ' . $number;
32.        }
33.    }
34.    sort($result, SORT_NATURAL);
35.    return $result;
36. }
```

<sup>1</sup> <https://regex101.com>: <https://regex101.com/>





## What Can We Swap?

The bulk of our work is in preparing our inputs so that we can compare them. PHP has a wealth of functions for working with arrays, and we can use `array_intersect()`<sup>2</sup> to find the elements in common. It gives us the stickers in one collector's "needs" list that are in someone else's "duplicates" list. I'm constantly learning about new functions in PHP's standard library, and exploring the ones available for sorting and comparing arrays is worth the time. (See Listing 4)

*It turns out, PHP also has an `array_uintersect()` function that allows you to define a callback for determining if two items are "equal." With it, you can inject custom logic for deciding if two items should be part of the intersection of two sets. It's not something I've used before, but it's good to have in your back pocket. Perhaps you want to use soundex to find words in two arrays that sound alike. Or you might have arrays that hold objects and have complex business logic that determines if any two are "the same" based on properties they have in common.*

### Listing 4.

```
3. $need = <<<EOM
4. FWC 2, 3, 7 9, 13, qat 5, qat 6 10 qat 11, 13, 14, qat 15,
5. qat 18, 19, ECU 1 2 3, 5, 6, 10, 13 19, SEN 7, 8, sen 11,
6. sen 19, NED 2, 3, 13, ENG 7, ENG 10, ENG 19,
7. IRN 2, 3, 5, 7, 10 12 17 19, USA 2, USA 12, USA 14 15 16,
8. USA 17, USA 19, WAL 2, WAL 5, WAL 16, ARG 4 8 9 10
9. Arg 11, Arg 14, Arg 18, ECU 14 15, ECU 18 Ned 17
10. EOM;
11.
12. $dups = <<<EOM
13. 00, FWC 1,4, FWC 12, FWC 17, QAT 1, QAT 3,4 ECU 18 19
14. GER 2,8, 10, 13,14, 15, JPN 3, JPN 10, JPN 14 15 19,
15. JPN 20, BEL 3 9 12 14 15 16 19, Can 5,7,17,19, USA 4,
16. MAR 4, MAR 14, MAR 16, SEN 11, SEN 19, NED 2,3,13,ger 20,
17. CRO 6, CRO 18, BRA 1, BRA 8, BRA 13, SRB 5,10 14, SRB 16,
18. SRB 18, 20, SUI 4, SUI 6, SUI 9, SUI 11, SUI 15, SUI 20,
19. CMR 7 9,11,13,15,16, POR 3, 5, 8, POR 10,11,12, WAL 16
20. por 15,16, 18, Wal 2,5, USA 15,16,17
21. EOM;
22.
23. $need = toStickerList($need);
24. $dups = toStickerList($dups);
25.
26. $swap = array_intersect($need, $dups);
27. echo "You can swap the following: " . implode(' ', $swap);
```

We can output the values of `$swap` so they can trade and get one step closer to completing their albums. In this case, they can swap these 12 stickers:

You can swap the following: ECU 18, ECU 19, NED 2, NED 3, NED 13, SEN 11, SEN 19, USA 15, USA 16, USA 17, WAL 2, WAL 5, WAL 16

## Shared Birthday Day

*Write a program that takes two people's birthdays and outputs the following information for the two dates:*

*Were they born on the same day of the week?*

- *Are they born in the same meteorological season?*
- *Were they both born on an even or odd day?*
- *What is the difference in ages in years, months, and days?*

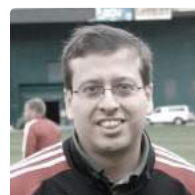
*In a given calendar year, how many days are between their birthdays?*

### 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`<sup>3</sup> 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: World Cup Draws* by Oscar Merida, November 2022.  
<https://phpa.me/puzzles-nov-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)

<sup>2</sup> `array_intersect()`: [https://php.net/array\\_intersect](https://php.net/array_intersect)

<sup>3</sup> `PsySH`: <https://psysh.org/>



# PSR-13: Link Definition Interfaces

Frank Wallen

There are two basic types of links: link and anchor tag. This article will take a more in-depth look at both and discuss the decision developers face to use one over the other in web applications.

These days nearly everyone with access to a browser, by computer or smartphone, understands what a link is: Something on the page the user clicks on, or touches, that takes them to another page or begins the download of assets like documents, PDFs, videos, and images (which, in the background, is really directing the browser to another resource or page where the asset is delivered to the browser). Links, however, are not just for navigating through the internet; they are also references to resources that direct the browser to load (like CSS) or are identified as related to the document one is viewing. Links may be invisible to the viewer, but the system loading the document (like a browser or feedreader) can aggregate them for the user, either pre-loading resources for viewing and downloading or organizing and presenting them *for* viewing at another time. PSR-13<sup>1</sup> aims to define interfaces for links, manage links, and offer methods for serializing and providing links (for reference and definition of links, see RFC 5988<sup>2</sup>).

There are two basic types of links:

- Link tag like `<link rel="stylesheet" href="mystyles.css"/>` placed in the `<header>`.
- Anchor tag like `<a href="https://duckduckgo.com">Search at Duck Duck Go</a>`

The big difference is that an anchor tag *redirects* the user to another resource, while the link tag *references* a resource. A linked resource, in the case of stylesheets, identifies an asset that should be loaded for the page so that

it's rendered correctly and completely. Another form of the link can reference a resource that an Atom RSS reader would recognize as the URI to the feed it should be subscribing to or loading. For example: `<link rel="alternate" type="application/rss+xml" href="...xml"/>`, which would allow search engine robots to identify and find the associated RSS feed.

At a minimum, a Link must consist of a URI and a relationship defining how it relates to the source. Attributes are not really standardized, so PSR-13 does not place any restrictions on them.

The interfaces provided by PSR-13 can be found on GitHub at `php-fig/link`<sup>3</sup>. The primary interface is `Psr\Link\LinkInterface`:

## Listing 1.

```
1. namespace Psr\Link;
2.
3. interface LinkInterface
4. {
5.     public function getHref();
6.     public function isTemplated();
7.     public function getRel();
8.     public function getAttributes();
9. }
```

This is very straightforward; an object implementing this interface would provide the URI through `getHref()`. The `getRel()` method would provide the list of resource relations that apply to the link, like 'alternate', 'stylesheet', etc. (an extensive list can be found here<sup>4</sup>, and the `getAttributes()` method provides a list of all the attributes that should be applied to the link (see RFC 5988 for a comprehensive definition of attributes). The `isTemplated()`

method is a boolean result indicating whether the URI is templated or not. A templated URI could be in the form of `https://mylink.org/{org:id}/{user:id}`, where the string would be interpolated to provide the requested data (the organization ID and the user ID). For more information about URI templating, see RFC 6570<sup>5</sup>.

The `EvolvableLinkInterface` was defined with the purpose of Link objects being implemented similarly to PSR-7 Response Objects<sup>6</sup>, with Link objects considered primarily as value objects since any data stored in them should be validated for use by the application. Therefore, they should be considered immutable, and any changes should be done with a new instance of the Link object. Each method in the following interface **MUST** return a new instance: See Listing 2 (next page)

In implementing this interface, the Link object can modify and validate its properties based on the application's requirements. It should be noted that the `$attribute` parameter in `withAttribute()` and `withoutAttribute()` is a mixed argument of the types `string|\Stringable|int|float|bool|array`. In the first version of PSR-13, the `$attribute` argument was expected to be a string. Multiple calls to `withAttribute()` with the same name should replace previously provided values.

The `LinkProviderInterface` is a recommended interface for a Link provider object that should return a list of Link objects to the requestor. It's a simple interface offering two methods:

1 PSR-13:  
<https://www.php-fig.org/psr/psr-13/>

2 RFC 5988:  
<https://www.rfc-editor.org/rfc/rfc5988>

3 `php-fig/link`:  
<https://github.com/php-fig/link>  
4 here: <https://phpa.me/iana-link-relations>

5 RFC 6570:  
<https://www.rfc-editor.org/rfc/rfc6570>

6 PSR-7 Response Objects:  
<https://www.php-fig.org/psr/psr-7/>



```
namespace Psr\Link;

interface LinkProviderInterface
{
    public function getLinks();
    public function getLinksByRel($rel);
}
```

Both methods should return an iterable list of Link objects or an empty list if there are no links. In the case of the `getLinksByRel()` method, this should return an iterable list of Link objects with the specified relationship only.

As a list of Link objects should be read-only, the `EvolvableLinkProviderInterface` is recommended, similarly to the `EvolvableLinkInterface`, to allow modifications to the list:

```
namespace Psr\Link;

interface EvolvableLinkProviderInterface
    extends LinkProviderInterface
{
    public function withLink(LinkInterface $link);
    public function withoutLink(LinkInterface $link);
}
```

#### Listing 2.

```
1. namespace Psr\Link;
2.
3. interface EvolvableLinkInterface extends LinkInterface
4. {
5.     public function withHref($href);
6.     public function withRel($rel);
7.     public function withoutRel($rel);
8.     public function withAttribute($attribute, $value);
9.     public function withoutAttribute($attribute);
10. }
```


Here, the methods must provide a new instance of the list. The `withLink()` method will provide the Link list but with the added Link object argument. If the Link object already exists in the list, then only one instance must exist. The `withoutLink()` method returns the list *without* the Link object argument.

## Conclusion

How links are handled will vary by context and application, and the decision is up to the architects. The recommendation of PSR-13 is a standardization of the handling of links for dependable results and presentation to the user, including better management of those links. The importance of supporting screen readers and even mobile layouts is improved by having a similar structure or system in place, which can help avoid typos and errors when building the HTML of a page.



Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He is from Southern and Baja California and dreams of having his own Rancherito where he can live with his family, have a dog, and, of course, a cat. [@frank\\_wallen](#)



# 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>



php[tek]

# Early Bird Tickets Now on Sale



<https://tek.phparch.com>

May 16-18, 2023

*Sheraton Suites Chicago O'Hare*

Brought to you by the good people at







# Get A Blog

Joe Ferguson

This month we're going back to the early days before Discord and Slack, to when we shared our knowledge through ancient tomes known as Blogs. Most developers have a blog, even if they never published it.

Blogs are the quintessential example of building application tutorials when learning a new language or framework, and I have certainly done my share of test blogs. Building your blog is a rite of passage, and when I discovered the PHP community in the early 2010s, blogs were how you shared ideas and solutions. My favorite part about blogs is the ability to go back and update the content when you resolve a problem! I'm not ashamed to admit I've sometimes found my own blog post when searching for a solution to a problem. We used to *write down* our problems and solutions to share with each other; what a concept!

Your blog doesn't have to be perfect. You're *going* to make mistakes and you're going to fix them and it's important to focus on sharing your knowledge. The quote "Perfection is the enemy of progress" fits accurately here as you will be tempted not to publish out of fear of failure. Publish anyway because you can always fix it later. Share your ideas and your work with the world. Learn to embrace and quickly fix your typos and grammatical errors.

Now that you're convinced you need a blog, where do you start? WordPress? Medium? I strongly urge you to own your content in some fundamental way. Owning your content means you control where your content is and how it's displayed. My personal example is my blog, which is stored in a Git repository that I explicitly control. I can put that content in any framework, such as Laravel, Symfony, or a content management system, such as WordPress, but I still *own* the content because its source of truth is the Git repository. I publish to my own website hosted on my own domain name so the content and related ways to access it are all directly under my control. If I don't like my host, I can easily pick up my repository and move to a new host. If I don't like my domain registrar, I can transfer it to a more agreeable option and I'm not at the mercy of any one service or company. This completely decouples my content from any hosting provider or service.

## Starting from Scratch

If you have never purchased a domain name before, welcome to the world of registrars, companies who hold domain name registrations. There are plenty of options, and you should look for a good deal while making sure the deal isn't *too good*. Pay attention to not only the registration fee that might be enticing but also the renewal rates. Also, we want to ensure the domain name registration includes DNS

(how your domain name gets routed to your host), which most *should* include. You should be able to find this for around \$20-30 a year, depending on your top-level domain (TLD) of choice. If you're like me, you have at least one registrar with a list of domains you are *totally* going to build one day... Maybe you already have a domain name for your blog? Let's use `phparch.com` as our example domain name for our blog; we have registered our domain at our registrar of choice, we have DNS, and next up, we need a place to *host* our blog.

Web hosting is big business, as you can probably imagine, and the type of blog you build will mostly dictate what you need. If you're going the WordPress route, you'll need a host established in the WordPress ecosystem and specializes in that hosting. I urge you to follow the path of least resistance and make your blog **boring**. It will be easy to update and keep up to date if it's boring. Don't use the latest snazzy widget creator framework for your blog unless that tool is boring to you. I've abandoned several blogs in frustration because I could never get `packages.json` to work after the initial build. Removing the blockers to creating your content is key to sharing that content with the world. My personal blog is the most boring static site generator setup in the world. I use a static site host, which is wired directly to my Git repository and automatically publishes changes when I push to the `main` branch. I don't have to remember any fancy commands to create a new blog post. I create a new file in my repository and fill in the files and assets I may need; it's committed to the repository, and automation takes over from there. When I'm feeling extra lazy, I can directly commit and wait for the automation to take the change live rather than running a full development environment. It's rare, but you don't always need to fire up the development environment to fix a typo or add an image.

Once you have your hosting account, you need to connect your domain name to the hosting account. This is typically done by changing the DNS servers your domain name is using at the Registrar. For example, if you have an account at `reallygreathost.tld`, they may tell you to use `ns1.reallygreathost.tld` and `ns2.reallygreathost.tld` and give you an IP of `1.2.3.4`. Because our domain name comes with DNS, we can instead apply the host's DNS settings to our domain name, so we retain complete control of where our domain name is pointing. We can take the IP from the host and assign A records to our given IP. Once this change propagates across the internet, when you look up the IP for `phparch.com`, it will



resolve to 1.2.3.4. DNS propagation time can range from a few minutes to several days. Most modern registrars will often have relatively quick DNS refresh times, which can be seconds or minutes.

Boring is the name of the game, and what could be more boring than generating HTML from static files? We will cover two great PHP-based static site generators, Sculpin.io<sup>1</sup> and Tighten Jigsaw<sup>2</sup>.

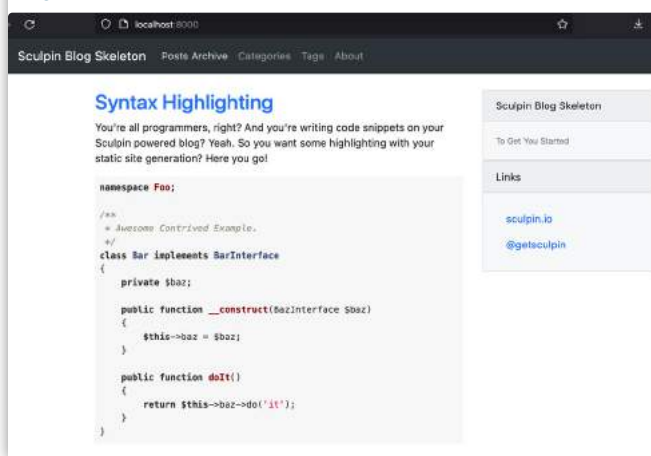
## Sculpin Static Site Generator

Getting started with Sculpin to create a new blog is one composer command away: `composer create-project sculpin/blog-skeleton phparch-sculpin`. We can immediately run the development command to generate our site and start the server.

```
$ vendor/bin/sculpin generate --watch --server
100% (73 sources / 0.01 seconds)
Processing completed in 0.13 seconds
Starting Sculpin server for the dev environment with debug true
Development server is running at <http://localhost:8000>
Quit the server with CONTROL-C
```

After running the generate command, we can browse our site at `localhost:8000`. (See Figure 1)

Figure 1.



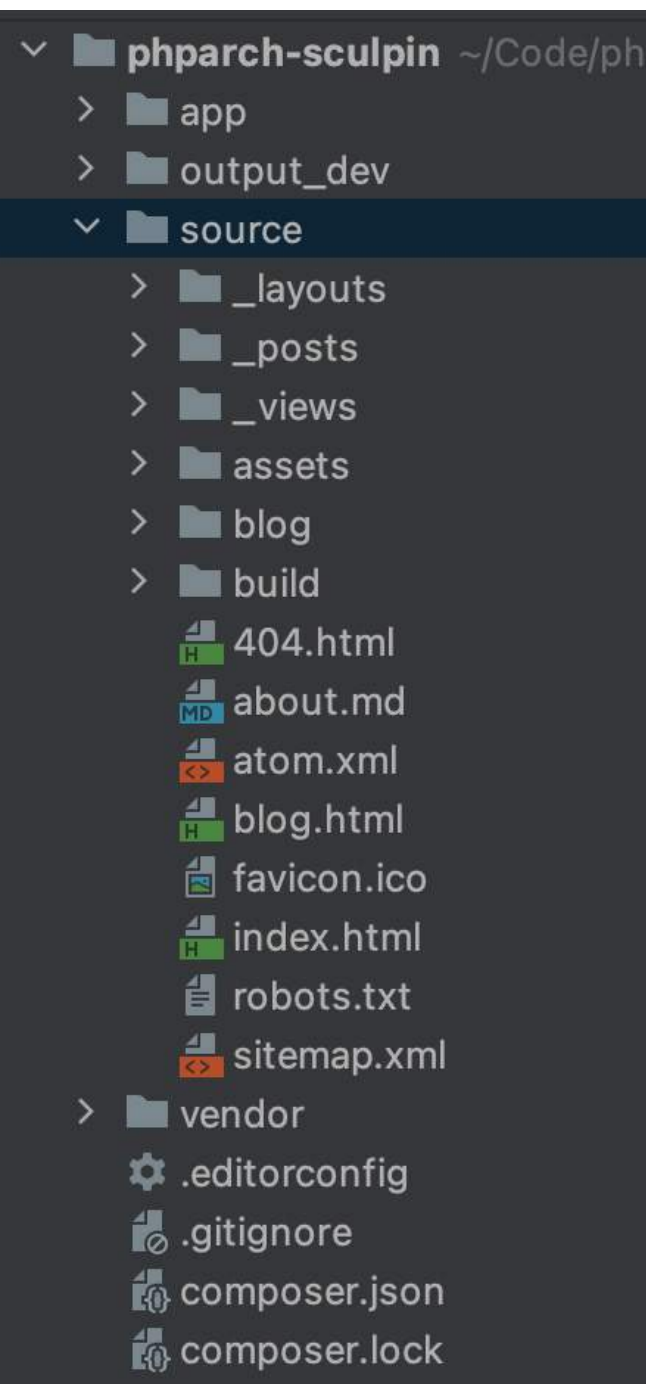
Viewing our Sculpin blog site running locally

The file structure of Sculpin breaks things out into its own `app` folder, `output_*` folders for the compiled site, `source`, which is where our files are stored, and finally, the traditional `vendor` folder. (See Figure 2)

Sculpin directory structure

The directory we're mostly focused on will be the `source` folder containing our content. Here we'll add our blog posts to `_posts/`, folder taking our Markdown content and converting it to HTML. Out of the box, Sculpin uses Twig

Figure 2.



template syntax, which should be familiar to PHP developers. The `_layouts/` folder is where we can define our page theme and layout options that we can extend for the rest of the sites. My personal blog has two layouts, one for highlighting my articles at [phparch.com](http://phparch.com)<sup>3</sup> and another for my traditional blog posts. You may want to use layouts to highlight specific types of content on your blog. One feature on my backlog is book recommendations; I could focus on short updates

<sup>1</sup> Sculpin.io: <http://sculpin.io/>

<sup>2</sup> Tighten Jigsaw: <https://github.com/tighten/jigsaw>

<sup>3</sup> [phparch.com](http://phparch.com): <http://phparch.com>

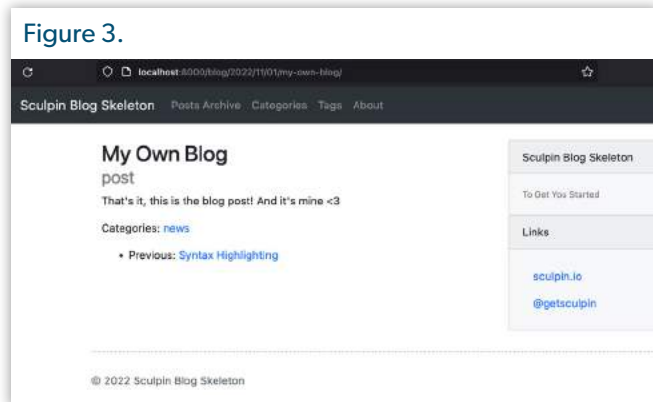


recommending a specific book, video, or another piece of media that would feature nicely alongside articles and blogs. I also keep a running speaking and writing resume on my blog if you need more examples of things you might feature on your site.

We're going to drop a new file in `source/_posts/` named `2022-11-01-my-own-blog.md`, which will be our brand new blog post on our fresh new Sculpin blog. The file contents can be as simple as this:

```
---
title: My Own Blog
categories:
  - news
---
That's it, this is the blog post! And it's mine <3
```

This will create a new post on our blog with our content:



Our “My Own Blog” post as HTML output

Ready to jump into the deep end, run `vendor/bin/sculpin generate --env=prod` and then upload the contents of `output_prod` to the public folder of your web host and your content will be live.

## Jigsaw Static Site Generator

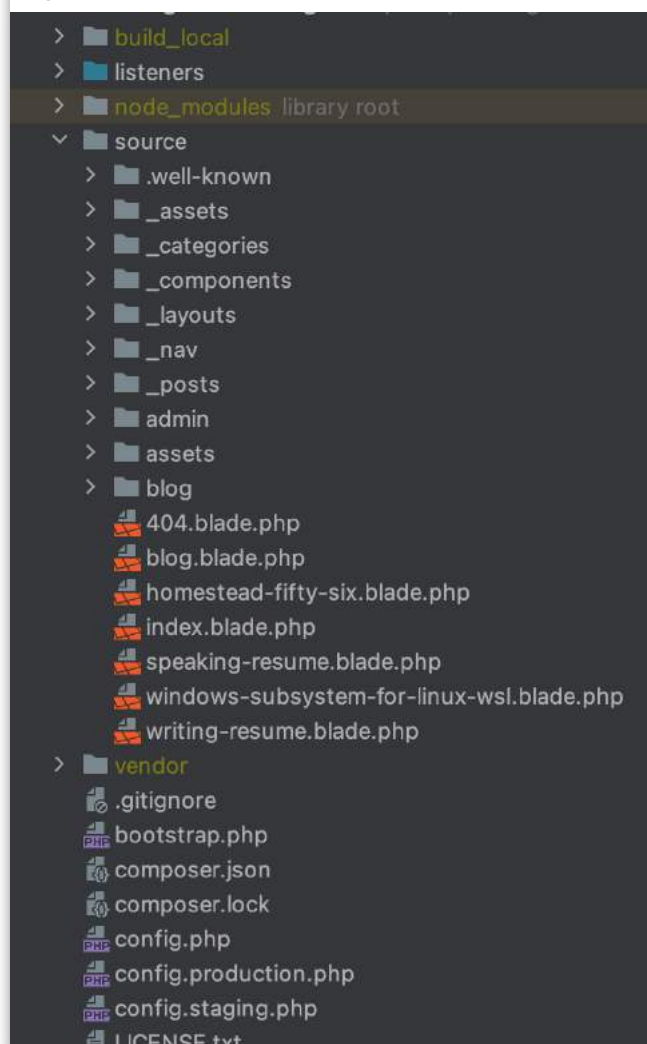
Jigsaw is a project by the fine humans at Tighten<sup>4</sup>, who built their own static site generator, which uses Laravel's Blade template engine by default. Getting started with Jigsaw, we can create a new folder, require the jigsaw project, then initialize a new site in our folder.

```
$ cd phparch-jigsaw
$ composer require tightenco/jigsaw
$ vendor/bin/jigsaw init blog
```

From here, we can build the site with `vendor/bin/jigsaw build` and run `npm run watch` to start the file watcher and refresh when changes are made. I'll give you a tour of my

blog's file structure to better understand what a “real” static site project might look like. (See Figure 4)

Figure 4.



Just like we saw with Sculpin, our content is going to live in the `source` folder, and we have a similar folder structure for `_posts` and `_layouts`; by default, everything uses Blade and Tailwind CSS. This project was my first real exposure to Tailwind CSS and I learned a lot. Remember, you should try to limit the number of new things you're learning at any one time. We're going after boring workflows to make it easier on us to come back and contribute more. We want to avoid fighting with NPM or other tools as much as possible, and Sculpin and Jigsaw both get out of my way and let me focus on my content, which is the only way I can productively post anything at all.

My site has an index/home page that is my about me information; then, I have the top-level blog index, which gives a title/preview of each blog post and contains category information for filtering. I also have pages for my writing and

4 Tighten: <https://github.com/tighten>



speaking resumes, and one featuring my book on Windows Subsystem for Linux and PHP development. Each of these pages is its own Blade template in `source/` and extends one of the files in `_layouts` to structure each page. The default `blade.php` is what the home page uses while `_posts` extend the `_layouts/posts.blade.php` file, and I have `_layouts/phparch.blade.php` to lay out the unedited copies of my past articles.

We can lean heavily on Blade templates to include files and yield content into sections, such as this snippet from my `_layouts/default.blade.php`.

#### Listing 1.

```

1. <div id="vue-search"
2.     class="flex flex-1 justify-end items-center">
3.     <search></search>
4.
5.     @include('_nav.menu')
6.
7.     @include('_nav.menu-toggle')
8. </div>
9. </div>
10. </header>
11.
12. @include('_nav.menu-responsive')
13.
14. <main role="main" class="flex w-full container
15.     max-w-12xl mx-auto py-8 px-6">
16. <div class="flex flex-wrap w-full">
17. <div class="md:w-9/12 xl:w-9/12 px-5 overflow-y-auto">
18.     @yield('body')
19. </div>
20. <div class="md:w-3/12 xl:w-3/12 px-5">
21.     @include('_components.sidebar')
22. </div>
23. </div>
24. </main>

```

We can include our navigation menus with `@include('_nav.menu')` and yield content via `@yield('body')` which allows our `index.blade.php` homepage template to only contain the unique page's content

```
@extends('_layouts.default')
```

```
@section('body')
```

```
    My Awesome content is here! Everything else gets inherited from default!
@endsection()
```

This allows me to easily take the established site layouts and focus on the specific content I want to share. And front matter<sup>5</sup> is supported to denote what categories a post should be associated with and the date and publish time information. For example, a blog post from January talking about building Homestead base boxes looks like this: (See Listing 2)

#### Listing 2.

```

1. ---
2. id: 975
3. title: "Building Homestead in 2022"
4. date: 2022-01-09T09:09:09-05:00
5. author: Joe
6. extends: _layouts.post
7. categories:
8. - vagrant
9. - homestead
10. - opensource
11. - applesilicon
12. ---
13.
14. There are two parts to the Homestead ...

```

This front matter allows us to set variables that can be used in the URLs and other places on our site. We're setting a post ID, title, date, and more. We're also labeling the post with categories that Jigsaw will build for us automatically. We'll have links to categories and posts that share categories to see related content on our site. Adding image assets is trivial, as anything placed in `source/assets/images` will be available at the `/images` URL of the site. We don't have to worry about weird permissions or uploading images with an FTP client; we just commit the assets next to the rest of our content and the robots take over building our site and deploying it.

This has been a whirlwind tour of two fantastic static site generators readily available and easy for PHP developers to pick up and start working with. Automation is key, and remember, we're intentionally boring. If you use a host that offers git repository integration, you can easily connect to the service and configure the site to auto-publish. We've successfully removed as many blockers to sharing our ideas and content by using automated tools and integrations. Some hosts may not offer such integrations, and in that event, we'd have to fall back to our GitHub Actions tools to automatically build our site and deploy the resulting assets to the host. That host could be an S3 storage bucket distributed worldwide or a Raspberry Pi running on your desk.

Get a blog and post your ideas somewhere they'll stick around longer than the next billionaire buys and wrecks on your favorite social media site. Show me your blog; I'd love to see it. I've had my blog longer than I've had any other social media—it's not perfect, but it's mine.



*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](#)*

<sup>5</sup> front matter: <https://jekyllrb.com/docs/front-matter/>



# Debt Management

Eric Mann

Every successful development team has two things in common: They've shipped a product and accepted compromises to make that shipment possible.

*Developers are inevitably forced to compromise. Commit a quick project fix without tests. Ship an experimental module without proper documentation. Add a feature to production to satisfy an urgent client request at 4:55 pm on a Friday.*

Each one of these compromises is debt. They are the direct shift (intentional or not) of some development from *now* to the context of a future development team.

Just as *financial* debt is shifting a monetary burden from now to the future, these kinds of engineering compromises are *technical* debt.

*Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite.—Ward Cunningham*

Every team and every project has technical debt. It comes with the territory when you start building software. Luckily, many see the term “technical debt” as negative; however, that is not always the case.

## Not All Debt is Bad

One of the more nuanced topics discussed in business schools is debt—specifically, how debt can sometimes be a good thing. In personal finances, debt is usually negative; in corporate finances, it's often a necessary tool to help grow the business. To that end, many businesses are rated on their effective use of debt.

Similarly, *technical* debt is not always a bad thing. Instead, it's a powerful tool

in your team's arsenal that helps you ship faster, iterate more quickly, and launch products to your customers' delight.

Cutting corners and making intentional compromises are often a *requirement* for shipping a product. No release will ever be perfect, and a laundry list of items will usually be left on the team's to-do list for the next iteration.

The negative aspects of technical debt occur when this list of to-do items grows and affects the overall stability and security of the application itself.

## Security Implications of Technical Debt

Complexity within a large codebase can hide security issues. It's easy to pass data around through the lifecycle of an application, and the longer it lives in memory, the easier to lose track of where it came from. Determining whether or not a specific value was properly sanitized earlier in a program's execution can become difficult.

More often than not, though, the security impact of technical debt is far more subtle.

Every project has external dependencies. Libraries we import to save time writing code. External systems with which we integrate to flesh out a platform—the underlying operating systems and languages atop which we build. Every moving part added to an application is a compromise—it's trusting that *some other* development team cares as much about our application's security as we do.

And trusting that *our* development team will keep these dependencies up to date.

The well-documented SolarWinds breach of late 2020<sup>1</sup> is a perfect example. SolarWinds' products were used by over 300,000 customers, including key members of the Fortune 500. It enabled remote attackers to inject code into software updates distributed by SolarWinds itself. Organizations that were dependent on and trusting of SolarWinds then inadvertently granted those same attackers full access to their systems.

The entire breach has drastically eroded trust in upstream dependencies—a conversation still bouncing around among infosec leaders now in 2022.

It's critical that engineers work rapidly to ship product value. When that rapid turnaround time necessitates a reliance on third-party engineers, we intentionally weaken the security stance of our own applications. Using that library, vendor, or other dependency is a convenient shortcut that also increases the debt presented by our application.

## Getting Out of Debt

Technical debt is unavoidable. One day, your team will make a necessary compromise to get the code for a feature out the door. Remember, this is *not a bad thing*, but it is still something to keep track of.

Like financial debt, if technical debt is not paid off over time, it will accrue interest. This interest often comes in the form of problematic legacy code and even longer lead times for refactoring or bug fixing.

Instead, keep track of these compromises when they happen and *immediately* make plans for revisiting

<sup>1</sup> SolarWinds breach of late 2020:  
<https://phpa.me/security-supply-chain>



your code in the future to reduce the impact of the debt you've just incurred. Devote a future sprint to refactoring a feature implementation from a different angle. Intentionally schedule downtime to catch up on upgrades to a critical system. *Make room in your development schedule* to implement long-term solutions to the near-term shortcuts you're taking as you take them.

Additionally, take time to budget for writing tests to increase the overall code coverage of your application. Just like you'd budget to pay off a personal loan, commit a set percentage of development time *each cycle* to paying down your technical debt. Add new documentation, refine integration tests, refactor complex methods, or upgrade library dependencies. Each of these changes will help your team cultivate a stronger, more maintainable codebase.

## Intentional Coding

Technical debt should never be incurred by mistake. Discuss priorities with the team and work together to find agreement on the best way to get your product out the door. Commit to following up on any previously deprioritized work so it doesn't disappear into a perpetual backlog. *Intentional* technical debt can get your product in front of your

customers before the competition. Intentionally *paying off* that same debt can keep your product stable and secure long into the future.

### Related Reading

- *Security Corner: A Reintroduction to TLS* by Eric Mann, February 2020.  
<https://phpa.me/security-corner-feb-2020>
- *Security Corner: Cross Site Request Forgery* by Eric Mann, June 2020.  
<https://phpa.me/security-june-20>
- *Security Corner: Getting Started with Cybersecurity* by Eric Mann, February 2022.  
<http://phpa.me/2022-02-security>

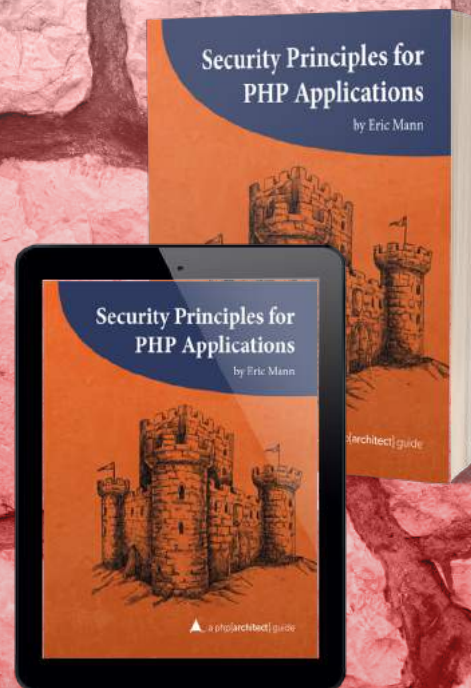


*Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: [@EricMann](https://twitter.com/EricMann)*

## Secure your applications against vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

**Order Your Copy**  
<https://phpa.me/security-principles>

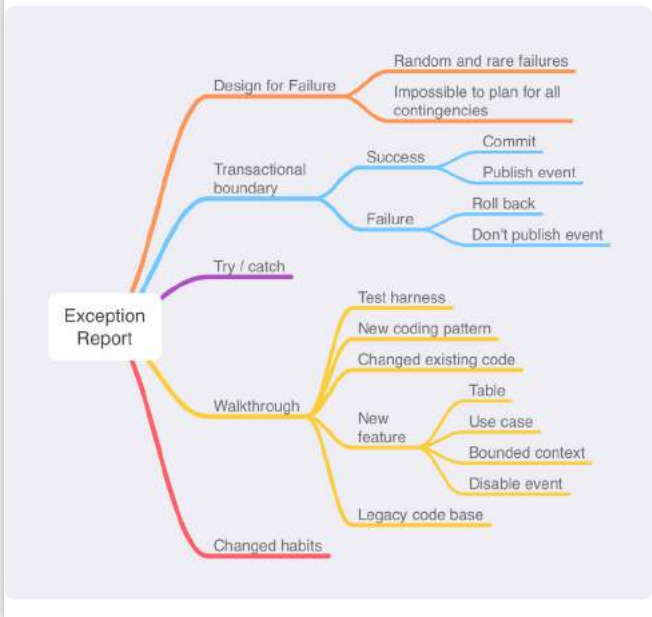


# Exception Report

Edward Barnard

Planning for failure is difficult because it's usually not practical to predict every possible thing that could go wrong. This month we're creating a mechanism for capturing those "rare and random" failures. As we do so we'll gain important insight regarding implementing invariants with a transactional boundary.

Figure 1.



All source code is available on GitHub<sup>1</sup>.

Some listings in this article are abbreviated so we can more easily focus on what changed. All code remains available in full in the GitHub repository.

## Essential Questions

Upon surviving this article you should be able to answer the following:

- How do we implement transactional consistency in our Bounded Context pattern?
- How do we enforce an invariant in our Bounded Context pattern?
- How do we record "random and rare" failures?

## Designing for Failure

The June 2022 "DDD Alley" introduced the concept of "random and rare" failures.

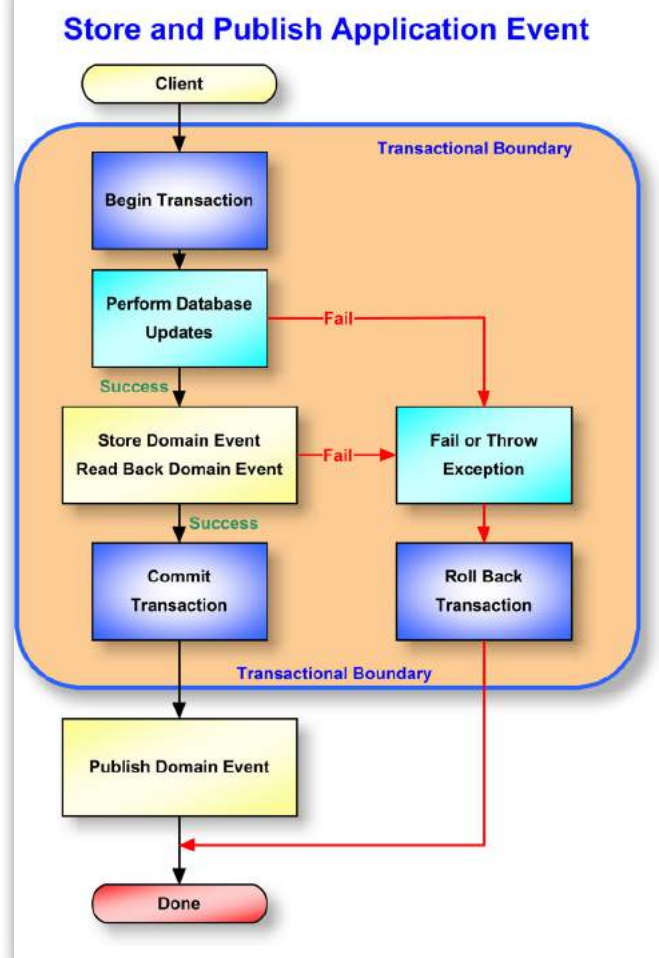
<sup>1</sup> GitHub: <https://github.com/ewbarnard/strategic-ddd>

PHP projects generally have lots of points where something could fail, in theory, but that would be a rare happening. What if it does fail? If we created a separate contingency plan for every single line of code that could, someday, fail for some unknown reason, we'd never get anything done.

Let's start our "design for failure" by reviewing how and when we're publishing Application Events. Figure 2 places careful emphasis on identifying the transactional boundary.

After we begin the transaction, we will complete the transaction with either COMMIT or ROLLBACK. That's our transactional

Figure 2.



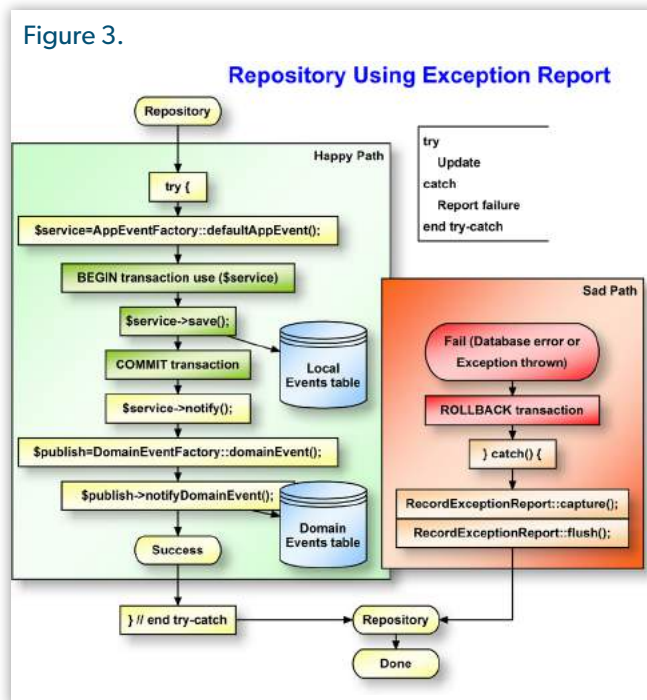




boundary. Now comes the chance to exercise our “nicety of awareness”. We see a transactional boundary. Does this boundary express a business rule that must remain invariant? Indeed, it does! Both the happy path (commit) and the sad path (rollback) express the invariant.

Oops... wait! Have we fully expressed the invariant? No, we have not! The event can only be published upon success (commit). The flowchart is correct, but this part of the invariant is outside the transactional boundary.

Figure 3 shows the solution. Place the transaction inside a try / catch block. When an error happens within the transaction, the PHP code rolls back the transaction and then throws (or re-throws) an exception. Processing continues inside the catch block. In other words, we fulfill our invariant (i.e., business rule, which must always be consistent) with the combination of transaction and try / catch.



We’ll use this structure to catch and record those “random and rare” failures. Note that we cannot save an error record to the database within the transaction. When the transaction rolls back, we lose the error record as well! However, we *can* capture the error in memory somewhere, and once outside the transaction, flush the errors (if any) to the database.

[Millett] relates this concept to the Aggregate pattern (page 419):

*Another take on the domain events pattern is to decouple the publishing and handling of events, so that side effects are isolated. This approach is implemented by storing a collection of events on the aggregate root and publishing them once the aggregate root has completed its task.*

*Significantly, a dispatcher is called from the [Application Services] layer to publish the events, keeping the technical concern out of the domain model.*

Let’s walk through the code to see how everything works.

## Test Harness

Listing 1 shows how we will trigger the failure path.

### Listing 1.

```
1. final class CountEventsCommand extends Command
2. {
3.     public function execute(
4.         Arguments $args,
5.         ConsoleIo $io
6.     ): ?int {
7.         $service = CountEventsFactory::countEvents();
8.         $service->insertCurrentCount();
9.         $service->insertCurrentCount();
10.        $io->out('Count complete');
11.
12.        return 0;
13.    }
14. }
```

The table described in listing 2 contains the datetime column when\_counted, but that column also has a unique key. If we insert two rows with the same date and time (i.e., during the same wall-clock second), the second insert should fail due to the unique-key constraint. That’s our failure mechanism.

### Listing 2.

```
1. CREATE TABLE `event_counts`
2. (
3.     `id`          int unsigned NOT NULL AUTO_INCREMENT,
4.     `when_counted` datetime
5.                 NOT NULL COMMENT 'Intentional poor design',
6.     `event_count` bigint unsigned NOT NULL DEFAULT '0',
7.     `created`     datetime NOT NULL,
8.     `modified`    datetime NOT NULL,
9.     PRIMARY KEY (`id`),
10.    UNIQUE KEY `when` (`when_counted`)
11. ) ENGINE = InnoDB;
```

The Count Events Factory (see listing 3) instantiates the Count Events repository and passes that repository object into the Application Service. The factory passes the application service to our test harness, the Count Events Command.

The Count Events Service (see listing 4) orchestrates our use case with insertCurrentCount(). The first line asks the repository for the row count. The second line instructs the repository to insert the count in our database.





## Listing 3.

```

1. final class CountEventsFactory
2. {
3.     #[Pure]
4.     public static function countEvents(): CountEvents
5.     {
6.         $repository = new RCountEvents();
7.         return new CountEvents($repository);
8.     }
9. }

```

## Listing 4.

```

1. <?php
2.
3. final class CountEvents
4. {
5.     private RCountEvents $repository;
6.
7.     public function __construct(RCountEvents $repository)
8.     {
9.         $this->repository = $repository;
10.    }
11.
12.    public function insertCurrentCount(): void
13.    {
14.        $count = $this->repository->collectCount();
15.        $this->repository->storeCount($count);
16.    }
17. }

```

## The New Pattern

The Count Events Repository shows our new pattern. See Listing 5. We have a transaction inside a try / catch block. The transaction inserts both the event count and the application event. They will both succeed or both fail.

Our new feature, class RecordExceptionReport, sits within the catch block. But first, note that the Application Event service has a slight change. Its save() method now accepts a database connection. All parts of the transaction need to run on the same connection. This slight change caused several files to change.

The application service interface (see listing 6 on the next page) shows save() now requires the database connection as a parameter.

The repository interface shows save() now requires the database connection as a parameter.

```

interface IAppEvent
{
    public function save(
        string $insert, string $read, array $parms,
        Connection $conn
    ): array;
}

```

The base application service (see listing 7) save() accepts the connection object and passes it to the repository save().

## Listing 5.

```

1. final class RCountEvents
2. {
3.     use AppEventsPrimaryTrait;
4.     use EventCountsTrait;
5.
6.     public function __construct() { $this->loadModels(); }
7.
8.     private function loadModels(): void
9.     {
10.        $this->loadLocalAppEventsTable();
11.        $this->loadEventCountsTable();
12.    }
13.
14.    public function collectCount(): int
15.    {
16.        return $this->localAppEventsTable->find()->count();
17.    }
18.
19.    public function storeCount(int $count): void
20.    {
21.        $connection = $this->eventCountsTable->getConnection();
22.        $data = [
23.            EventCount::FIELD_WHEN_COUNTED => FrozenTime::now(),
24.            EventCount::FIELD_EVENT_COUNT => $count,
25.        ];
26.        $entity = $this->eventCountsTable->newEntity($data);
27.        try {
28.            $appEvent = $this->appEvent();
29.            $connection->transactional(
30.                function ($conn) use ($entity, $appEvent) {
31.                    $this->eventCountsTable->saveOrFail($entity);
32.                    $appEvent->save($conn);
33.                });
34.            $appEvent->notify();
35.        } catch (JsonException|Exception $e) {
36.            $message = $e->getMessage();
37.            $detail = [
38.                'trace' => $e->getTraceAsString(),
39.                'entity' => $entity->toArray(),
40.                'entity errors' => $entity->getErrors(),
41.            ];
42.            RecordExceptionReport::capture($message, $detail);
43.            RecordExceptionReport::flush();
44.        }
45.    }
46.
47.    private function appEvent(): IAppEvent
48.    {
49.        return AppEventFactory::dbStateChangeAppEvent(
50.            'Spike for event count'
51.        );
52.    }
53. }

```

The repository (see listing 8) got simplified because it no longer needs to orchestrate its own transaction.

The Application Event Factory, the Application Event service (the child class), and the constants class remain unchanged.



## Listing 6.

```

1. namespace ...\Events\AppEvent\DomainModel\Interfaces;
2.
3. use Cake\Database\Connection;
4.
5. interface IAppEvent
6. {
7.     public function __construct(
8.         IAppEvent $repository,
9.         string $action,
10.        string $description,
11.        ?array $detail = null
12.    );
13.
14.    public function addDetail(array $detail): void;
15.    public function save(Connection $conn): void;
16.    public function notify(): void;
17. }

```

## Listing 7.

```

1. abstract class BaseAppEvent
2.     implements CAppEventOriginatingContexts, IAppEvent
3. {
4.     public function save(Connection $conn): void
5.     {
6.         $parms = [
7.             $this->action,
8.             static::$ subsystem,
9.             $this->description,
10.            $this->detail,
11.            $this->uuid,
12.        ];
13.        $this->readback = $this->repository
14.        ->save(static::$insert, static::$read, $parms, $conn);
15.    }
16. }

```

## The New Feature

Listing 9 shows the exception reports table. It should receive records of any “rare and random” failures we encounter. Each row, therefore, should be examined by a human.

I have found this table to be quite useful for errors that are not so rare and not so random. During new feature development, in my development environment, I often cause database errors that are less than obvious. I might have the wrong column name, validation, or a plain old coding error. Something failed, but I don’t know what or why.

When I follow this pattern during new feature development, the exception\_reports table tells me what happened. This can also be helpful when problems with the new feature sneak into production. For example, suppose the new feature got out of sync with the database table change. The exception report can inform you right away.

## Record Exception

Listing 10 shows how we capture information independently of the database transaction. Both available methods, capture() and flush(), are static, so this can be easily called from anywhere in the codebase.

Method capture() gathers the information and stores it inside an ExceptionReportEntity object. capture() then appends the object to its internal static array. In other words, we’re capturing the information in memory.

Method flush() writes each of those entities to the database. We only call flush when *outside* a database transaction. We

## Listing 8.

```

1. final class RAppEventDefault implements IAppEvent
2. {
3.     /**
4.      * Should be called while within a transaction
5.      */
6.     public function save(
7.         string $insert,
8.         string $read,
9.         array $parms,
10.        Connection $conn
11.    ): array
12.    {
13.        $statement = $conn->prepare($insert);
14.        $statement->execute($parms);
15.        $statement = $conn->prepare($read);
16.        $statement->execute([$statement->lastInsertId()]);
17.        $readback = $statement->fetchAll('assoc');
18.        if ( ! (
19.            is_array($readback) &&
20.            array_key_exists(0, $readback)
21.        )) {
22.            throw new DatabaseException('Readback failed');
23.        }
24.        return $readback[0];
25.    }
26. }

```

## Listing 9.

```

1. CREATE TABLE `exception_reports`
2. (
3.     `id`          int unsigned NOT NULL AUTO_INCREMENT,
4.     `description` varchar(255) NOT NULL DEFAULT '',
5.     `detail`      json          DEFAULT NULL,
6.     `created`     datetime      NOT NULL,
7.     `modified`   datetime NOT NULL ON UPDATE CURRENT_TIMESTAMP,
8.     PRIMARY KEY (`id`)
9. ) ENGINE = InnoDB;

```

do an immediate return if nothing has been captured. Thus, we can call flush() any number of times without harm. Upon successful flush, we reset the internal static array to be empty.

As shown in listing 11 (see next page), the entity class ExceptionReportEntity became more lines of code than it should

because it includes the PHP 8 array shapes and is immutable. It simply captures the provided information and then, during the `flush()` operation, returns that information formatted for the database record insertion.

Our repository (see listing 12 on the next page) follows the same pattern we've been using. We can't record an exception in the midst of flushing exceptions, so instead, we return `false` to indicate the `flush()` failed.

## Disable Application Event

Now that we have the full pattern in place, our final step is to disable storing and publishing Application Events. There's no need to risk degrading production database performance without further planning.

We're going to implement the "disable" feature in the legacy codebase. I'm showing the configuration as a PHP class constant; you would likely make this a site configuration item of some sort.

```
namespace LegacyBoundedContexts\...\DomainModel\Constants;
interface CDisableAppEvent
{
    // Set true for production
    public const DISABLE_APP_EVENT = false;
}
```

Listing 13 (see next page) shows our implementation. In both `save()` and `notify()`, check to see if the feature is disabled, and if so, return without further processing. This approach allows us to design for both Application Events and recording "random and rare" exceptions. We'll still be reporting "random and rare" exceptions as they happen, regardless of whether we are publishing Application Events or not.

## Legacy Exception Report

Here are the changes made to our legacy codebase.

The legacy exception report entity (see listing 14) is new.

The legacy exception report repository (see listing 15) is also new.

The legacy exception report service (see listing 16) is likewise new.

The legacy application event repository interface now includes the connection object being passed in.

```
interface IAppEvent
{
    public function save(
        string $insert,
        string $read,
        array $parms,
        Connection $connection
    ): array;
}
```

### Listing 10.

```
1. final class RecordExceptionReport
2. {
3.     private static array $captures = [];
4.
5.     private function __construct()
6.     {
7.     }
8.
9.     public static function capture(
10.         string $description,
11.         array $detail = []
12.     ): void {
13.         self::$captures[] =
14.             new ExceptionReportEntity($description, $detail);
15.     }
16.
17.     public static function flush(bool $isTest = false): void
18.     {
19.         if ( ! count(self::$captures)) {
20.             return;
21.         }
22.         if ($isTest) {
23.             self::reset();
24.         } else {
25.             $repository = new RExceptionReport();
26.             if ($repository->flush(self::$captures)) {
27.                 self::reset();
28.             }
29.         }
30.     }
31.
32.     public static function reset(): void
33.     {
34.         self::$captures = [];
35.     }
36.
37.     /** Unit test support */
38.     public static function errorCount(): int
39.     {
40.         return count(self::$captures);
41.     }
42. }
```

The legacy application event repository (see listing 17) is a complete update because we no longer need to manage the transaction within this repository.

The legacy application event service interface (see listing 18) now includes the connection object being passed in. This service will be passing it into the repository.

The legacy test harness inserts the current count twice. This should trigger the exception report.

```
require_once(__DIR__ . '/bootstrap.php');

$service = CountEventsFactory::countEvents();
$service->insertCurrentCount();
$service->insertCurrentCount();
echo 'Count complete' . PHP_EOL;
```



Oops! Something went wrong! Below shows that we successfully triggered the “rare and random” error, but we produced no exception report.

```
/usr/bin/php test/exercise_count_events.php
Query failed: An exception occurred while executing
'insert into event_counts(when_counted, event_count...)
VALUES (now(), ?, now(), now())' with params [1]:
SQLSTATE[23000]: Integrity constraint violation:
1062 Duplicate entry '2022-03-29 09:12:23'
      for key 'event_counts.when'
Count complete
```

We changed the pattern but forgot to update the repository to show the new pattern! Listing 19 shows the new pattern. Place the transaction inside a try / catch block, and record the exception report inside the catch block.

Finally, note that since we disabled application events, we can freely incorporate the new pattern within our legacy codebase.

## Changed Habits

The drawback with this approach to capturing “random and rare” failures is that every change to the database needs to be placed inside a try / catch block. If including the Application Event pattern, each database change requires a manual transaction within the try block.

Designing the try / catch block forces us to consider what to do in the case of a failure. That in itself is a good thing! We now have a standard way of reporting failures. If they rarely happen, perhaps one in a million updates, so much the better.

The domain events shown here are *not* proper Domain Events in the DDD sense. These events report the fact that the database changed with a brief description as to why. They’re intended to help developers avoid data corruption. That’s useful but not the same consideration as notifying interested parties that something “important” happened.

This changed habit feels like a hassle. It’s more work and more code to write to accomplish a given task. However, as a result, I got in the habit of recording every request and response from a third-party API. We knew it was intermittently failing and identified precisely what was failing with this record.

Listing 11.

```
1. #[Immutable]
2. final class ExceptionReportEntity
3. {
4.     #[Immutable(Immutable::CONSTRUCTOR_WRITE_SCOPE)]
5.     private string $description;
6.
7.     #[Immutable(Immutable::CONSTRUCTOR_WRITE_SCOPE)]
8.     private ?array $detail;
9.
10.    public function __construct(
11.        string $description,
12.        ?array $detail = null)
13.    {
14.        $this->description = $description;
15.        $this->detail = $detail;
16.    }
17.
18.    /**
19.     * Create array in correct format for
20.     * ExceptionReportsTable::newEntity()
21.     */
22.    #[ArrayShape([
23.        ExceptionReport::FIELD_DESCRIPTION => "string",
24.        ExceptionReport::FIELD_DETAIL => "array|null",
25.    ])]
26.    public function data(): array
27.    {
28.        return [
29.            ExceptionReport::FIELD_DESC => $this->description,
30.            ExceptionReport::FIELD_DETAIL => $this->detail,
31.        ];
32.    }
33. }
```

Listing 12.

```
1. final class RExceptionReport
2. {
3.     use ExceptionReportTrait;
4.
5.     /**
6.      * @param ExceptionReportEntity[] $captures
7.      */
8.     public function flush(array $captures): bool
9.     {
10.        if (empty($captures)) {
11.            return true;
12.        }
13.        $this->loadExceptionReportsTable();
14.        $conn = $this->exceptionReportsTable->getConnection();
15.
16.        try {
17.            $conn->transactional(function () use ($captures) {
18.                foreach ($captures as $capture) {
19.                    $data = $capture->data();
20.                    $entity = $this->exceptionReportsTable
21.                        ->newEntity($data);
22.                    $this->exceptionReportsTable
23.                        ->saveOrFail($entity);
24.                }
25.            });
26.        } catch (Exception) {
27.            return false;
28.        }
29.        return true;
30.    }
31. }
```





## Listing 13.

```
//See full listing in this articles code download
1. abstract class BaseAppEvent
2.     implements CAppEventOriginatingContexts, IAppEvent
3. {
4.     public function save(Connection $connection): void
5.     {
6.         if (self::DISABLE_APP_EVENT) {
7.             return;
8.         }
9.
10.        $parms = [
11.            $this->action,
12.            static::$ subsystem,
13.            $this->description,
14.            $this->detail,
15.            $this->uuid,
16.        ];
17.        $this->readback = $this->repository
18.            ->save(
19.                static::$insert,
20.                static::$read,
21.                $parms,
22.                $connection
23.            );
24.    }
25.
26.    public function notify(): void
27.    {
28.        if (self::DISABLE_APP_EVENT) { return; }
29.        if (empty($this->readback)) { return; }
30.        $domainEvent = DomainEventFactory::domainEvent();
31.        $domainEvent->notifyDomainEvent(
32.            static::$sourceTable,
33.            $this->readback
34.        );
35.    }
36. }
```

## Listing 14.

```
1. final class ExceptionReportEntity
2. {
3.     private $description;
4.
5.     private $detail;
6.
7.     public function __construct(
8.         string $description,
9.         ?array $detail = null
10.    ) {
11.        $this->description = $description;
12.        $this->detail = $detail;
13.    }
14.
15.    /** Create array in correct format for table insert */
16.    public function data(): array
17.    {
18.        try {
19.            $detail = is_array($this->detail) ?
20.                json_encode($this->detail)
21.                : null;
22.        } catch (JsonException $e) {
23.            $detail = null;
24.        }
25.        return [$this->description, $detail];
26.    }
27. }
```

## Listing 15.

```
1. final class RExceptionReport extends BaseModel
2. {
3.     /** @param ExceptionReportEntity[] $captures */
4.     public function flush(array $captures): bool
5.     {
6.         if (empty($captures)) {
7.             return true;
8.         }
9.
10.        $connection = $this->db->getConnection();
11.        $sql = 'insert into exception_reports
12.            (description, detail, created, modified)
13.            VALUES (?, ?, now(), now())';
14.
15.        try {
16.            $connection->transactional(
17.                function ($conn) use ($sql, $captures) {
18.                    foreach ($captures as $capture) {
19.                        $parms = $capture->data();
20.                        $conn->executeUpdate($sql, $parms);
21.                    }
22.                }
23.            );
24.        } catch (DBALException|Exception $e) {
25.            return false;
26.        }
27.
28.        return true;
29.    }
30. }
```



Listing 16.

```

3. final class RecordExceptionReport
4. {
5.     /** @var array */
6.     private static $captures = [];
7.
8.     private function __construct() { }
9.
10.    public static function capture(
11.        string $description,
12.        array $detail = []
13.    ): void
14.    {
15.        self::$captures[] = new ExceptionReportEntity(
16.            $description,
17.            $detail
18.        );
19.    }
20.
21.    public static function flush(bool $isTest = false): void
22.    {
23.        if ( ! count(self::$captures)) {
24.            return;
25.        }
26.
27.        if ($isTest) {
28.            self::reset();
29.        } else {
30.            $repository = new RExceptionReport();
31.            if ($repository->flush(self::$captures)) {
32.                self::reset();
33.            }
34.        }
35.    }
36.
37.    public static function reset(): void
38.    {
39.        self::$captures = [];
40.    }
41.
42.    /**
43.     * Unit test support
44.     */
45.    public static function errorCount(): int
46.    {
47.        return count(self::$captures);
48.    }
49. }

```

Listing 17.

```

1. <?php
2.
3. namespace LegacyBoundedContexts\...\AppEvent\Repository;
4.
5. final class RAppEventDefault
6.     extends BaseModel implements IAppEvent
7. {
8.     /**
9.      * Should be called while within a transaction
10.     */
11.     * @throws \Doctrine\DBAL\DBALException
12.     */
13.     public function save(
14.         string $insert,
15.         string $read,
16.         array $parms,
17.         Connection $connection
18.     ): array
19.     {
20.         $connection->executeUpdate($insert, $parms);
21.         $id = (int)$connection->lastInsertId();
22.         $statement = $connection->executeQuery($read, [$id]);
23.         $rows = $statement->fetchAll();
24.         if (empty($rows)) {
25.             throw new InvalidArgumentException('Readback failed');
26.         }
27.         return $rows[0];
28.     }
29. }

```

Listing 18.

```

1. <?php
2. namespace LegacyBoundedContexts\...\DomainModel\Interfaces;
3.
4. interface IAppEvent extends CDisableAppEvent
5. {
6.     public function __construct(
7.         IAppEvent $repository,
8.         string $action,
9.         string $description,
10.         ?array $detail = null
11.     );
12.
13.     public function addDetail(array $detail): void;
14.
15.     public function save(Connection $connection): void;
16.
17.     public function notify(): void;
18. }

```



## Listing 19.

```

1. final class RCountEvents extends BaseModel
2. {
3.     public function collectCount(): int
4.     {
5.         $sql = 'select count(*) row_count
6.             from local_app_events
7.             limit 1';
8.         try {
9.             $statement = $this->sql->executeQuery($sql, []);
10.            $rows = $statement->fetchAll();
11.        } catch (DBALException $e) {
12.            return 0;
13.        }
14.        if (is_array($rows) &&
15.            (1 == count($rows))) {
16.            return $rows[0]['row_count'];
17.        }
18.        return 0;
19.    }
20.
21.    public function storeCount(int $count): void
22.    {
23.        $sql = 'insert into event_counts
24.            (when_counted, event_count, created, modified)
25.            VALUES (now(), ?, now(), now())';
26.        $parms = [$count];
27.        try {
28.            $connection = $this->db->getConnection();
29.            $appEvent = $this->appEvent();
30.
31.            $connection->transactional(
32.                function ($conn)
33.                use ($sql, $parms, $appEvent) {
34.                    $conn->executeUpdate($sql, $parms);
35.                    $appEvent->save($conn);
36.                }
37.            );
38.        } catch (JsonException|DBALException|Exception $e) {
39.            $detail = [
40.                'message' => $e->getMessage(),
41.                'trace' => $e->getTraceAsString(),
42.                'count' => $count,
43.            ];
44.
45.            RecordExceptionReport::capture(
46.                'Legacy store count failed', $detail);
47.            RecordExceptionReport::flush();
48.        }
49.    }
50.
51.    private function appEvent(): IAppEvent
52.    {
53.        return AppEventFactory::dbStateChangeAppEvent(
54.            'Spike for legacy event count'
55.        );
56.    }
57. }

```

## Essential Questions Answered

- How do we implement transactional consistency in our Bounded Context pattern? With a database transaction.
- How do we enforce an invariant in our Bounded Context pattern? With a database transaction inside a try / catch block.
- How do we record “random and rare” failures? By gathering information during the failure and persisting that information after the problem transaction has rolled back.

## Summary

We recognized that “random and rare” failures are often ignored. If we devised a contingency plan for every possible failure with every possible line of code, we’d never get anything done!

We looked at the idea of domain events, specifically the advice that any domain event be persisted in the same transaction as the item of interest. Then, if and only if that whole transaction was successful, publish the domain event.

PHP frameworks understand the need to save or update related data in the same transaction. All must succeed or fail as a unit; otherwise, we have an inconsistent state. However, the framework will not recognize the domain event as needing to be within the same transaction. We began managing database transactions explicitly in our PHP code.

While this appears to be more work, this allowed us to capture information when things go wrong. We can now catch and record those “rare and random” happenings that can cause problems down the line.

## Related Reading

- *DDD Alley: Domain Event Walkthrough* by Edward Barnard, September 2022.  
<https://phpa.me/ddd-sept-2022>
- *DDD Alley: Application Event Walkthrough* by Edward Barnard, October 2022.  
<https://phpa.me/ddd-oct-2022>
- *DDD Alley: Transactional-Boundary* by Edward Barnard, November 2022.  
<https://phpa.me/ddd-nov-2022>



*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)



# Who Controls Your Content?

Beth Tucker Long

We are becoming a world increasingly reliant upon “free” services—free document sharing, free email, free video meetings, and free chat programs. The list goes on and on. Would your organization survive if these free services disappeared? Do you truly understand the risks you are taking?

Them: We are not sure what happened, but your account was canceled instead of converted.

Me: Can you reinstate it?

Them: We cannot reinstate your account until the cancellation process completes.

Me: How long will that take?

Them: Cancellation takes 86 days. After 86 days, you can contact us to reinstate your account, and then you can reissue your request to have your account converted.

This was a conversation I just had with Google. Google recently converted all GSuite accounts over to their new Google Workspace. I had a Legacy GSuite account from way back in the day when Docs first became part of GSuite. As a part of their conversion to Workspace, Legacy account holders got a free two-week preview of Workspace and then would have to decide: purchase a Workspace subscription or request conversion to a personal Google account.

I only use my Legacy GSuite account to access files that people and organizations share with me through Docs (like the schools my children attend), so I requested conversion to a personal Google account. This is where the troubles began. Instantly, everything disappeared in my Google Drive—all the files I created as well as all files shared with me. The above-quoted conversation happened when I asked why I had lost my files. One wrong button click on their end, and I lost access to everything for 86 days.

It wouldn't be so bad since I don't use this for work, but this isn't just affecting me. This means that organizations that I work with, who have full, paid accounts, cannot share documents with me for 86 days. This mistake on my free account is hindering their paid accounts.

This got me thinking—what if I had been using this for work? What if my livelihood depended on these documents, and I lost access to them for 86 days because someone pushed the wrong button?

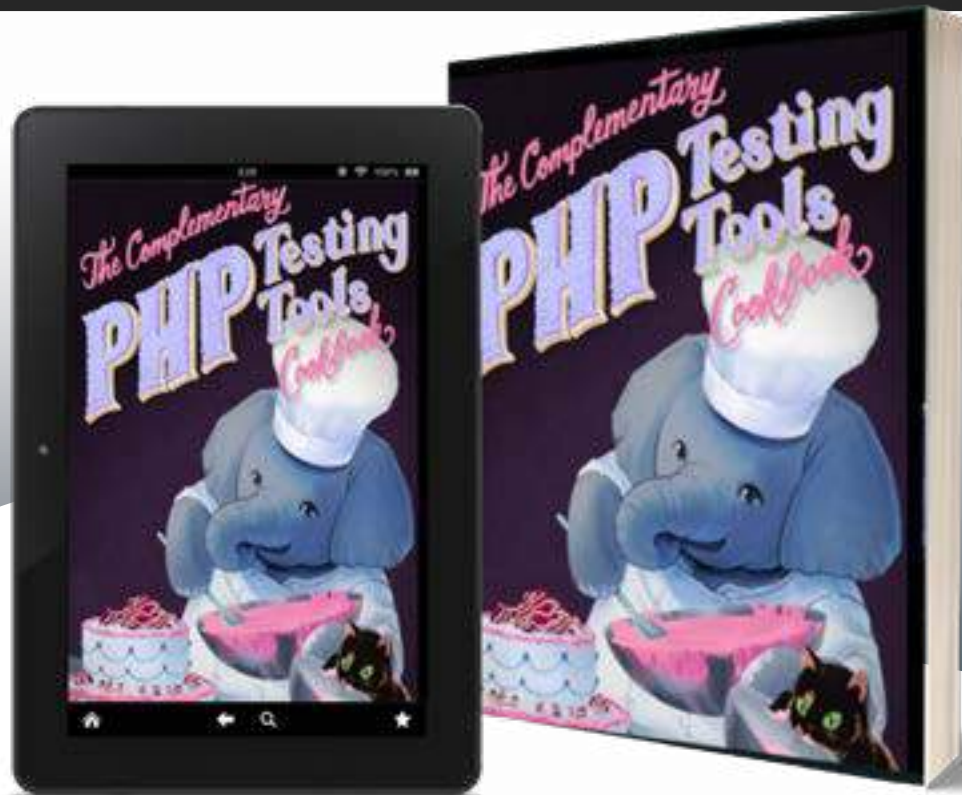
So much of our world relies on other companies properly handling our data and content. This is incredibly risky! Owning the content doesn't help if you can't get to it. Besides access issues, there are also major privacy risks with this setup. When you give a company access to your content, you don't know who else will also have access to it. I'm not just talking about server hacks and data breaches (which major platforms like Google, Meta, and Yahoo have all suffered). I'm talking about who these services are willingly giving your data to. Meta has been criticized for giving advertisers access to users' data, most notably Cambridge Analytica, for disclosing Facebook Messenger messages to law enforcement, and for not fully notifying users of how thorough their user tracking software is. Google is in a similar situation, with controversies over giving Gmail users' emails to third-party developers, tracking users while location tracking was turned off, and storing and sharing information about private browsing sessions that aren't supposed to be stored.

These companies are focused on profiting off of our content, not on being responsible with our trust. As you look at what services you will work with and rely upon in the upcoming year, I urge you to consider the dangers. What would happen if a service shared your content with advertisers (who could be your competitors)? What would you lose if you got locked out of the service? Is the convenience of these services worth the risks?



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



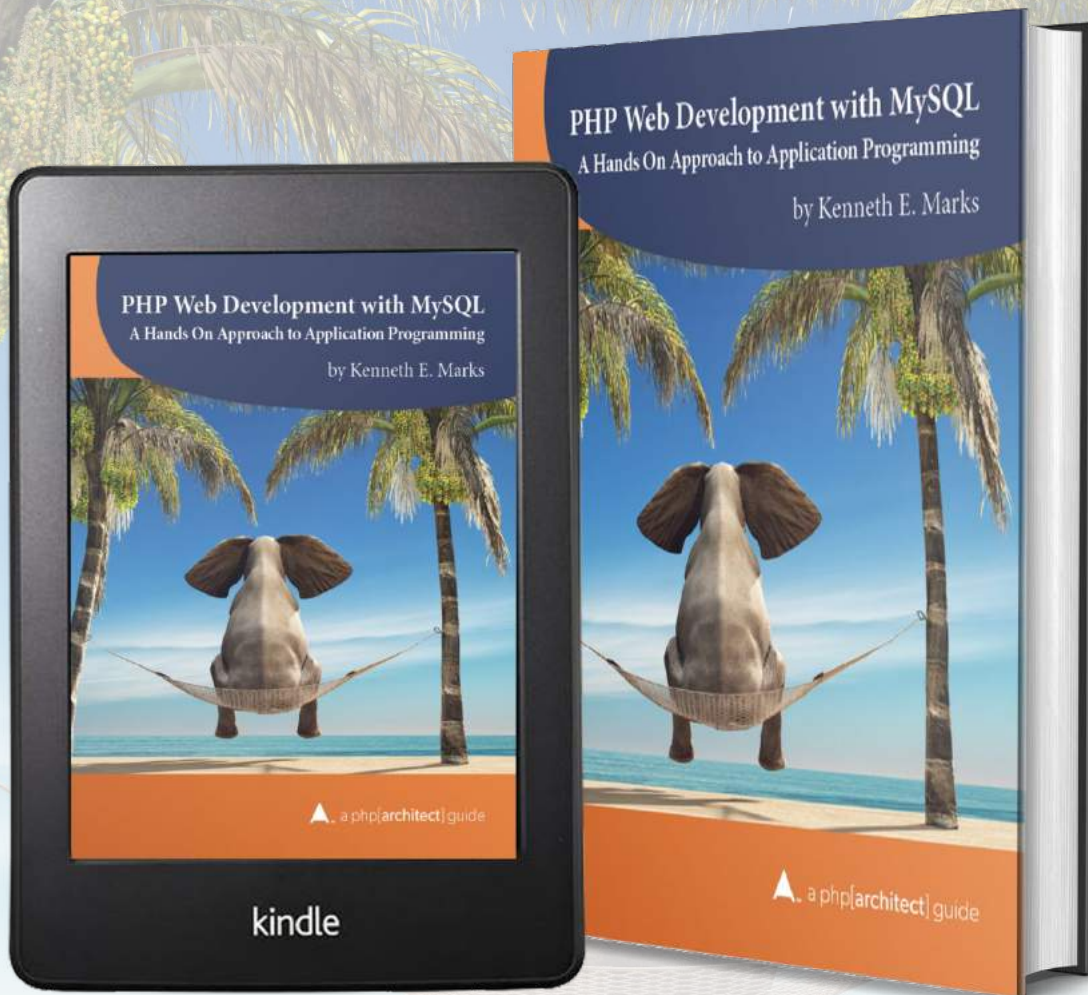


Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

*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](http://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>