



# php[architect]

The Magazine For PHP Professionals

## THE VALUE OF THE AST

### Your AST Toolkit Bring Value To Your Code

ALSO INSIDE

#### The Workshop:

Local Dev with Lando

#### Education Station:

Making Our Own Web Server: Part 2

#### PHP Puzzles:

World Cup Draws

#### Security Corner:

Direct Object References

#### DDD Alley:

Transactional Boundary

#### PSR Pickup:

PSR-11: Container Interface

#### Drupal Dab:

Security in Drupal 9

#### Finally{}:

Our Responsibility in Learned Helplessness

# 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

**NOVEMBER 2022**  
Volume 21 – Issue 11



**2 Giving Thanks**

**3 Your AST Toolkit**

Tomas Votruba

**11 Bring Value To Your Code**

Dmitri Goosens

**16 Local Dev with Lando**

The Workshop

Joe Ferguson

**21 Direct Object References**

Security Corner

Eric Mann

**24 PSR-11: Container Interface**

PSR Pickup

Frank Wallen

**26 Making Our Own Web**

**Server: Part 2**

Education Station

Chris Tankersley

**31 World Cup Draws**

PHP Puzzles

Oscar Merida

**35 Transactional-Boundary**

DDD Alley

Edward Barnard

**41 Security in Drupal 9**

Drupal Dab

Nicola Pignatelli

**44 Our Responsibility in  
Learned Helplessness**

finally{}

Beth Tucker Long

Edited immutably

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.

# Giving Thanks

November is upon us, and in the United States, it's the month we celebrate Thanksgiving. I want to take this time to thank each and every one of you who reads our magazine. Producing each issue takes more time than I ever knew, and we have had our share of ups and downs over the past year.

In a world full of stressful situations, it's necessary to take a few minutes to appreciate what we already have. I personally have a loving wife, two amazing kids, and a fulfilling career. Take a moment to think about the positives of your life right now, and savor that feeling.

So I want to take a moment and share one of the positives from my life, especially my professional career. Back in 2010, I chose to attend my first ever PHP conference called php[tek]. In my eyes, 12 years ago, it was a magical experience. I've met some fantastic people that I am still friends with today. I've been able to help many of those friends find jobs, and I know that if I were in need, they'd do the same for me. I have a group of people I can lean on when I am struggling mentally and with code. You get to know people and learn who to reach out to when you are struggling to solve a problem. At that conference, I also learned to suppress the imposter syndrome in me and realized that while I had a TON to learn, I wasn't as bad of a developer as I wanted to believe. I am thankful for that experience and hope you get to experience it when you come to php[tek] in 2023.

Diving into this month's magazine, we have a ton of great content for you. Leading off is a follow-up to an article from September's issue about the Abstract Syntax Tree. Tomas Votruba brings us *Your AST Toolkit*, teaching us how we can use the AST to make upgrading less painful. Some examples seem a little simplified, but the process is often used on a much larger scale. For example, my team uses the AST to enforce specific coding standards that apply only to our

codebase and are not already part of another tool, such as PHPStan.

Our second feature article is from Dmitri Goosens, with part 1 of *Bring Value To Your Code*. While value objects themselves are straightforward to understand, their value to our codebase cannot be understated. I used to believe that value objects were unnecessary until I wrapped my head around the power they provide.

Our columnists have been hard at work as well. Joe Ferguson was in The Workshop this month tinkering with *Local Dev With Lando*, which is a fantastic tool, especially if you juggle more than one project. Over in Education Station, Chris Tankersly gives us part 2 of his column from last month, *Making Our Own Web Server*, teaching us how to handle more than one connection at a time.

Oscar Merida is passionate about soccer, and this month's PHP Puzzle is determining *World Cup Draws*. These puzzles are great for helping our minds open up, think about problems, and try different solutions. Over in DDD Alley, Edward Barnard continues his Domain Driven Design series with *Transactional Boundaries*. All of these articles come together to teach us about all of the aspects of DDD. Security is an ever-growing topic and this month Eric Mann shows us the danger of *Direct Object References*. Take a moment to think about how your data could be exposed to the wrong people. Frank Wallen's *PSR-11: Container Interface*, will show us the interfaces that our favorite frameworks are using to automatically provide us the classes we need when we need them.

Nicola Pignatelli introduces us to *Security in Drupal 9* in this month's Drupal Dab column.

And finally{} closes this issue with *Our Responsibility in Learned Helplessness*. How can we help our website visitors be more successful and limit their learned helplessness?

## 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: <http://facebook.com/phparch>

## Download the Code Archive:

[https://phpa.me/November2022\\_code](https://phpa.me/November2022_code)



# Your AST Toolkit

Tomas Votruba

Today we look at the three most powerful tools in the AST Toolkit. What is the tool for, how do we use it properly, and where should we use something else or our bare hands?

We looked at *why* AST manipulation should be one of our core skills of 2023 in the previous post called Real World AST<sup>1</sup>, published in the September 2022 issue.

Now we know we have a lot of power and ability to shape the whole project when we look at the code with AST glasses. In the same way we can see much more with X-rays:

Figure 1.



(license: <https://creativecommons.org/licenses/by-nc/2.0/>)

The previous post was published almost a month ago, so let's refresh our memory and take a look back.

## What is an AST?

An AST, or in full wording—**abstract syntax tree**, is not a tool, not a specific script, not a software you can use. **It's an approach to seeing the code.**

To get a better idea, let's look at a few ways we can see PHP code:

- The most obvious way to see code is to **execute the code**. We provide a few input values, the script modifies the data, and then prints out the result.

Also, we have a couple of ways to see the PHP code *between lines* without actually running it. It would not be exact, but it is similar to parsing code:

- We have **reflection**, that can analyze metadata about the code without running it. We can get class name, we can detect what properties the class has. We can see metadata about the properties, whether they're typed, propagated, read-only etc.
- Then we can also look at PHP code in the form of **basic tokens** with `PhpToken::tokenize()`<sup>2</sup>. We can see this single line of code `echo 1` as: `T_ECHO`
  - `T_SPACE`
  - `T_WHITESPACE`
  - `T_LNUMBER`

Check [3v4l.org](https://3v4l.org/example) example<sup>3</sup> and list of tokens<sup>4</sup>. You can get an exact idea about the complexity and cumbersome approach to tokens there.

What is **the abstract syntax tree way** of seeing code? We get the best of all worlds—we see the code, its exact values, and its position in the whole tree, e.g.

The `echo 1;` code is parsed into AST node objects like this:

```
new Echo_(new LNumber(1));
```

The most powerful idea I want to talk about is code modification with patterns. Have you heard of **pattern refactoring**? Let's say we want to change all `echo` with a number to an `echo` with a string.

We define a specific pattern, the `A → B` change, and run the automated refactoring on the whole project.

The biggest advantage of the AST approach is that **we have to write very little code to change a single pattern in the whole codebase**. Imagine doing the same change with tokens, where every space, parentheses, and brackets matters.

The definition of pattern refactoring above might be similar to the following (apologies to php-parser experts):

```
-new Echo_(new LNumber(1));
+new Echo_(new String('1'));
```

With this single line, we change the whole codebase. Isn't that amazing? **We define the exact pattern and we are 100 % sure that only the exactly defined pattern is changed. Everything else is skipped.**

<sup>1</sup> Real World AST: <https://phpa.me/real-world-ast>

<sup>2</sup> `PhpToken::tokenize()`: <https://phpa.me/php-tokenize>

<sup>3</sup> [3v4l.org](https://3v4l.org/example) example: <https://3v4l.org/IIp4v>

<sup>4</sup> list of tokens: <https://www.php.net/manual/en/tokens.php>

Now, you may wonder: “why would anyone want to change echo from integer to string—that’s a pretty weird and dangerous move”? Apart from playing an April fools joke on your colleagues when they go to the toilet and forget to lock their screens, **this change has a highly practical value.**

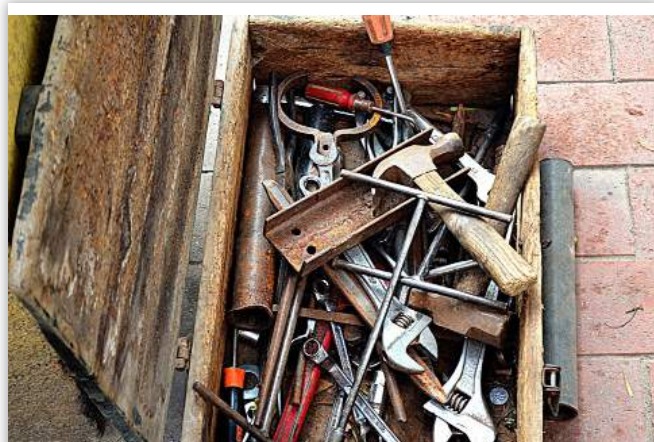
Once you know how to write a PHP script **that can perform a very basic pattern refactoring, you know how to create a pattern refactoring.**

From there on, you can change whole structures of code. For example, add `int` type to a property once you know it works only with integer values and you’re using PHP 7.4.

```
-private $value = 1000;
+private int $value = 1000;
```

You can change anything that you’d otherwise change in the smaller commit. Anything that you can explain to a colleague, that is.

**The only rule is**—the change has to be  $A \rightarrow B$ . There must be exactly one way to change code in every situation. In philosophical terms, we would say the algorithm must be *deterministic*.



(license <https://creativecommons.org/publicdomain/zero/1.0/>)

## AST Toolkit for the PHP Developer

Now that we have refreshed our memory with pattern refactoring, let’s dive into real tools that can make the change happen. Imagine that you’ve bought a new flat and want to make a couple of changes. Your father lends you his lifetime toolkit, and now you’re about to discover the content of this toolkit.

Today we look at the\*\* three most powerful tools in the AST Toolkit\*\*.

What is the tool for, how do we use it properly, and where should we use something else or our bare hands?

5 OpenSource Pledge: <https://phpa.me/opensource-pledge>

6 php-src core: <https://github.com/php/php-src>

7 PhpParser 5-alpha1: <https://phpa.me/php-parser>

Let’s start with a tool that can solve most of our problems—a tool to parse and modify PHP.

## AST Parser for PHP

Fortunately, we’re blessed as a PHP community because we already have a package written in PHP to parse PHP—it’s called `nikic/php-parser`.

What I’m about to say might sound like a bold statement, but it’s true: Nikita Popov has been the greatest PHP-core developer of all time. He contributed many great features that we now use daily<sup>5</sup> and cannot imagine our life without them.

One of those features is AST in the `php-src` core<sup>6</sup> itself. The C parses PHP into AST and then works with it internally. This topic is outside our scope, but it’s worth mentioning because it’s why we have a `nikic/php-parser` package.

We’re now interested in the package that parses PHP with PHP—the `nikic/php-parser` package. Without further ado, let’s dive into coding:

```
composer require nikic/php-parser:dev-master --dev
```

Note: The current stable version is `PhpParser 4`, but to keep this post a bit more updated, we’ll work with the latest `PhpParser 5-alpha1`<sup>7</sup>. Who knows, maybe during the publishing of this post, a stable version of the `PhpParser 5` will be published.

Now, I have a very atypical suggestion for a written article: what if we code along together? If you do, you will get a real experience and a wow-moment with this post. Everything clicks once you see you are in control of changing one tiny piece of code to another without ever touching the code.

Are you ready? Let’s dive in.

- First, we create a `refactor-php-pattern.php` file where we will write the custom logic to parse, traverse, and print PHP code.
- We will also create `SomeCode.php`, as a sample PHP file from our application, with `echo 1` in it: (See Listing 1)

Now we have the parser ready, and we get the exact AST nodes for our code.

### Listing 1.

```
1. // refactor-php-pattern.php
2. require_once __DIR__ . '/vendor/autoload.php';
3.
4. use PhpParser\ParserFactory;
5.
6. $factory = new ParserFactory();
7. $parser = $factory->createForNewestSupportedVersion();
8.
9. $file = __DIR__ . '/SomeCode.php';
10. $fileContents = file_get_contents($file);
11. $astNodes = $parser->parse($fileContents);
```

## Adding Node Visitor

How do we add **the node visitor** that changes `echo 1` to `echo "1"`? Thanks to the previous post about AST<sup>8</sup>, we know that node visitors are how we define changes we want to see in the code.

In the future, we'll implement many node visitors, one for every change we need in our code.

With PhpParser each node visitor must extend `PhpParser\NodeVisitorAbstract`. Our job is to implement the `enterNode()` method, that is called for each node in the AST. (See Listing 2)

That's it!

### Listing 2.

```
1. use PhpParser\Node;
2. use PhpParser\Node\Name;
3. use PhpParser\Node\Stmt\Echo_;
4. use PhpParser\Node\Scalar\LNumber;
5. use PhpParser\NodeVisitorAbstract;
6. final class ChangeEchoNumberToStringNodeVisitor
7.     extends NodeVisitorAbstract
8. {
9.     public function enterNode(Node $node): ?Echo_
10.    {
11.        // is it echo node?
12.        if (! $node instanceof Echo_) {
13.            return null;
14.        }
15.
16.        // does echo node have number in it?
17.        if (! $node->expr instanceof LNumber) {
18.            return null;
19.        }
20.
21.        // yes, here we have exact pattern match
22.        // change value from a number to a string
23.        $node->expr = new String_((string) $node->expr);
24.        // return node to modify it
25.
26.        return $node;
27.    }
28. }
```

Then we run the node visitor:

```
use PhpParser\NodeTraverser();
$nodeTraverser = new NodeTraverser();
$nodeTraverser->addVisitor(
    new ChangeEchoNumberToStringNodeVisitor()
);

// here we parse the file to $astNodes

$traversed = $nodeTraverser->traverse($astNodes);
```

Good job! Now we have modified AST nodes with our desired echo format.

Finally, we print the modified code back to the file:

```
use PhpParser\PrettyPrinter\Standard;
$standardPrinter = new Standard();
$modified = $standardPrinter->prettyPrintFile($astNodes);
// print the code back to file
file_put_contents(__DIR__ . '/SomeCode.php', $modified);
```

Now we collect the fruits of our labor:

```
-echo 1;
+echo '1';
```

You've done it! You can change your whole project with a simple AST script.

## Downsides of Pure Php-parser

The script above works, and technically, it replaces all the occurrences of this exact code:

```
-echo 1;
+echo '1';
```

But it's rarely the case that we work with exact values. What if we used variables instead?

```
$value = 1;
echo $value;
```

Here the `$value` would be skipped, and the `echo` would still have a number. That's not what we want, and we developers clearly see it. The `$value` is an integer that will be echoed, so we should change it to the string:

```
-$value = 1;
+$value = '1';
echo $value;
```

What php-parser misses are **the variable types**. It can see AST nodes, the position in the tree, and the real values that are defined in the code. However, it lacks context.

The missing piece of the puzzle is called PHPStan<sup>9</sup>. A tool that can analyze types from scalar to union, through empty arrays, etc., for almost any node you can think of.

Another issue with vanilla php-parser is rather technical. The php-parser **default printer does not care about spaces**. That's not a feature of AST in any language. The AST doesn't see if you're using tabs or spaces; it only focuses on the content with value.

The following lines will be all parsed:

```
echo 1;
echo 1;
echo 1 ;
```

To the very same AST nodes:

```
new Echo_(new LNumber(1));
```

<sup>8</sup> Real World AST: <https://phpa.me/real-world-ast>

<sup>9</sup> PHPStan: <https://github.com/phpstan/phpstan>

Fortunately, `php-parser` can mimic original spacing with a **format-preserving pretty printer**.

This complex feature name is simply: “we’ll try to keep the format of your code the way it was before”. The downside is it requires quite a complex setup and understanding of lexer, tokens, and changed nodes.

## Rector to the Rescue

To solve all these nitpicking and dull issues, I created a tool called Rector. I already wrote about Rector in the issue from August 2019<sup>10</sup>, so you can go there to read more details if you’re interested.

Here, I’ll give you some special content so it’s worth paying for and so we can get a bit relaxed after the difficult coding:

Fun fact that only a few close friends of mine know. The name “rector” sounds like the leader of a university. But that’s only a coincidence. **The word “rector” is actually a shortcut.** Could you guess for what word?

For the first few months, the tool was called “Refactor”, the composer package, then “refactor/refactor”. The word was perfect, hence “automated refactoring”, but it was too long and hard to remember. Moreover, try googling “php refactor” to find a tool like this. Also, it’s easy to make a typo in the word “refactor” when you type it twice with a / in the middle.

I’ve found a tip in “To Sell is Human” by “Daniel Pink”: every brand word must be 2-syllables at the most so people can remember and relate to it. For example, “Goo-gle”, “Face-book”, “A-pple”, “Coke”, “Space-X”, “Tes-la”, etc.

So I made the package name shorter and easier to say: Rector.

Also, Rector grew from a tool that can change single simple dull patterns like `echo 1`, to a powerful tool that can change static methods to constructor injection with precise accuracy. So Rector as the leader of a university is now quite fitting as well.

In short, Rector is a smart wrapper around the mass of little problems, so you can focus on writing the patterns you want to refactor.

You can install it like this:

---

```
composer require rector/rector --dev
```

---

It has prepared sets with dozens of node visitors that can upgrade your code from PHP 5.3 to PHP 8.2, upgrade PHPUnit 4 to 9, or even upgrade from Symfony 2.8 to Symfony 6.1.

After installing, you create the `rector.php` configuration file:

---

```
vendor/bin/rector init
```

---

There you complete paths and sets, and you’re ready to go:

---

```
vendor/bin/rector process
```

---

Check <http://github.com/rectorphp/rector> for more details.

Now let’s jump to 2nd tool from our toolkit, that will help you to keep config changes in sync with your PHP code.

## 2. Yaml Config Parser in PHP

We can change PHP now and it’s an important base. But there are more suffixes in our files in a typical 2022 project running on PHP, right?

One being configuration files; unless they’re already in PHP, like in Symfony 4+ projects or Laravel, there is a big chance they’re in YAML or NEON format. These two formats have very similar syntax. See for yourself:

---

```
parameters:
  paths:
    - src
```

---

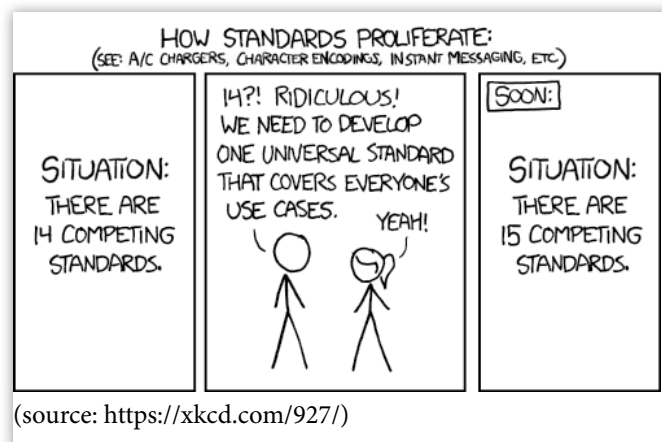
vs

---

```
parameters:
  paths:
    - src
```

---

Yes, you’re probably thinking, “yet another standard for the same code”:



(source: <https://xkcd.com/927/>)

And you would be right.

But what is usually a lack of cooperation that can lead to forks, can also lead to a good thing.

We see YAML the most often. As a format defined as abstract of PHP projects, it can be used with Python, JavaScript, Docker configs, or standalone.

Yet, YAML, the huge dinosaur that it is, has one downside—there is no YAML AST parser written in PHP.

You probably know where we’re heading...the NEON is a less popular format, almost niche. Do you know what tool is used for configuration? We’ve already mentioned it above—the PHPStan.

The smaller and the less popular the tool, the easier innovation is because the changes do not affect many end-users.

<sup>10</sup> Rector Article: <https://phpa.me/migrate-legacy-one-month>



That usually means fewer reported issues about BC breaks on Github and, in a circle, faster innovation.

## Think Outside the Box

This logic does not apply solely on these two formats and their parser package in PHP. Maybe there are more powerful tools that are less popular but can solve your problem. Be creative with your solutions—you might save yourself dozens of work hours.

Back to the YAML AST parser. The current status: there is no YAML AST parser per se. What if I told you, “We have to write one ourselves...”. That would be the moment where this post just lost you. **We want to use existing tools to make our work easier.** Not to get more work and create a new package.

We have good news: NEON has an AST parser! And in 99% of cases, the YAML and NEON syntax are the same.

## Lucky 1 %

But what if we're unlucky enough to hit the 1%? What can we do?

- we can handle the upgrade manually
- we can change these few lines to uniform syntax and enjoy the power of AST from now on

We faced a project where we had to change a whole three lines of code to make that happen—a three-minute job.

## How to Parse Yaml to AST

Let's get down to real code again. Code along for real-time feedback:

---

```
composer require nette/neon
```

---

Let's create:

- a refactor-yaml-pattern.php with our custom refactoring logic
- and a simple.yaml that could represent a config file with services definitions

---

```
services:
```

---

```
-
    class: App\SomeClass
    public: true
```

---

Now we write refactoring logic to refactor-yaml-pattern.php: (See Listing 3)

In the end we get Nette\Neon\Node\BlockArrayNode, an AST node for our YAML code. That's a success!

Once we can parse the YAML, we can modify it.

Now we want to make services private to enforce constructor injection in all our use cases:

---

```
services:
-
    class: App\SomeClass
    public: true
+
    public: false
```

---

<sup>11</sup> [packagist.org](https://packagist.org): <https://packagist.org>

### Listing 3.

```
1. <?php
2. require_once __DIR__ . '/vendor/autoload.php';
3.
4. use Nette\Neon\Decoder;
5.
6. $neonDecoder = new Decoder();
7.
8. $file = __DIR__ . '/simple.yaml';
9. $yamlContents = file_get_contents($file);
10. $neonAstNode = $neonDecoder->parseToNode($yamlContents);
11.
12. var_dump($neonAstNode);
```

How can we do that? The following code might be a bit confusing, so first, we look at the pattern we want to match:

- the public: true is an array item with key: value, so we look for an array item
- in the array item, the key must be a string with "public" value
- only if we have such an item, we look only at values that contain true as a value

That's it. Now let's put that into if/else code: (See Listing 4 on the following page)

We've just changed the value in a specific array item in the YAML file. Good job!

The last step is to print it out to a YAML file. It will be very easy:

---

```
$yamlContents = $traversedNode->toString();
file_put_contents(__DIR__ . '/simple.yaml', $yamlContents);
```

---

That's all for the YAML config AST pattern refactoring.

All we did was rename a key value. But again, when we know how to create pattern refactoring to rename a key in YAML, we have the base knowledge to make any other change.

## What About Non-php, Non-yaml, Non-neon Configs?

Do you use some other config format we have not mentioned yet? First, check [packagist.org](https://packagist.org)<sup>11</sup> to see if there's a parser for it. It's a better place to search for PHP packages than Github, as Packagist contains only PHP packages.

Look for “parser”, “AST”, “abstract syntax tree”, “parsing”, etc. Most likely, there is already some form of AST parser for it.

If not, like in the case of prehistoric \*.ini files, you again have two options:

- do the changes manually
- switch to better-supported formats like YAML and enjoy its tooling from now on

**Always consider the size of the change** Before going to pattern refactoring. If you have to change three files, maybe

## Listing 4.

```

1. <?php
2.
3. use Nette\Neon\Node;
4. use Nette\Neon\Node\ArrayItemNode;
5. use Nette\Neon\Node\LiteralNode;
6. use Nette\Neon\Traverser;
7.
8. // here we parse the YAML content to AST Node
9.
10. $neonTraverser = new Traverser();
11.
12. $traversedNode = $neonTraverser->traverse(
13.     $neonAstNode,
14.     function (Node $node) {
15.         // is it array item?
16.         if (!$node instanceof ArrayItemNode) {
17.             return null;
18.         }
19.
20.         // is it "public" key? continue
21.         if (!$node->key instanceof LiteralNode &&
22.             $node->key->value !== 'public') {
23.             return null;
24.         }
25.
26.         // is it `false` value already? skip it
27.         if ($node->value->toValue() === false) {
28.             return null;
29.         }
30.
31.         // Change the value to `false` - job done!
32.         $node->value = new LiteralNode(false);
33.         return $node;
34.     }
35. );
36. );

```

manual work is faster. If it's 300 files and the upgrade will take six months, the pattern refactoring might be useful in the long term.

### 3. AST in Twig Templates

We have covered PHP and config formats. I think we're confident enough to look at something more chaotic and out of control, like templates.

Let's say we have a class `Product`, and we rename one of its methods with the help of `php-parser` or `Rector`:

```

final class Product {
-   public function getTitle()
+   public function getName()
    {
        // ...
    }
}

```

We already know how to achieve this with pattern refactoring in pure PHP and YAML, but what about TWIG?

What happens if we rename the method in PHP and keep the TWIG untouched?

---

```
{{ product.title }}
```

---

It crashes on `getTitle()` method with a not found error, because only `getName()` method exists now. This only happens when the template content is actually run. Thanks to Murphy's law, it will likely be on a Friday evening when you have a 5-year-anniversary date with your wife in the restaurant from your first date.

How could we make both, the date and the production code, go well with pattern refactoring?

Our goal here is to update the TWIG with the new method name.

---

```
-{{ product.title }}
+{{ product.name }}
```

---

We need something like "Twig AST parser". Do we look for a new package? Do we hack around with another tool? Or do we have to write it ourselves (just kidding! keep reading)?

What if I told you that\*\* \*\*Twig already has a built-in AST parser and traverser?!

I was surprised, like you probably are now. It's been documented since version 0.9<sup>12</sup> in 2010. I wonder why there are no Rector-like tools for Twig yet. Unfortunately, the Twig does not include the printer, so we would have to write it ourselves.

What does the simplest parsing file look like?

Let's create:

- a `refactor-twig-pattern.php` with our custom refactoring logic
- and `simple.twig` file with product content

(See Listing 5 on the next page)

---

```
{{ product.title }}
```

---

We lack types, so the script only tries to approximate the PHP form of the code, but the benefits of fast pattern refactoring in Twig with very good accuracy, beat the lack of types any day.

When we have a node, we can traverse it and modify it! (See Listing 6 on the following page)

In the end we have changed the Twig AST!

The lack of a printer that can write the AST back to a twig file would require us to write a printer; however, that is out of scope for this piece. Also, I may have missed it in the documentation.

The important part is we can change a Twig template file with pattern refactoring.

<sup>12</sup> TwigPHP: <https://phpa.me/twigphp>

## Listing 5.

```

1. <?php
2. // refactor-twig-pattern.php
3.
4. use Twig\Environment;
5. use Twig\Loader\ArrayLoader;
6. use Twig\Parser;
7. use Twig\Source;
8.
9. require_once __DIR__ . '/vendor/autoload.php';
10.
11. $environment = new Environment(new ArrayLoader([]));
12.
13. // here is the file contents
14. $fileName = __DIR__ . '/simple.twig';
15. $fileContents = file_get_contents($fileName);
16. $fileSource = new Source($fileContents, 'simple_file');
17. $tokenStream = $environment->tokenize($fileSource);
18.
19. // here we parse tokens to AST nodes
20. $twigParser = new Parser($environment);
21. $twigAstNode = $twigParser->parse($tokenStream);
22.
23. var_dump($twigAstNode);
24. // we get "Twig\Node\ModuleNode" here

```

Note: are you interested in more AST TWIG related topics? Check [phpstan-twig-analysis](https://github.com/phpstan-twig-analysis)<sup>13</sup>—it's a great source of knowledge about TWIG AST written by Matthias Noback.

### Other Templates?

If you're working with different templates than we've covered so far, you have three options:

- look for the parser directly in your templating system
- take advantage of similar syntax, find a package that can parse your code even if not tailor-made
- switch your templates to better supported syntax - this will pay off in the future, as your upgrades will be very smooth and stable thanks to better tooling support

## Keep Expanding Your Toolkit

Today you have three extract tools in your toolkit when dealing with code changes across the whole project.

- PHP
- YAML
- TWIG

Try them one by one, give it time and take it slow—experiment with small changes in your project. Soon you'll start to "get it", and you'll be able to make more significant changes with better confidence. It took me six months just to get php-parser, so you're doing great!

## Listing 6.

```

1. <?php
2. use Twig\Environment;
3. use Twig\Loader\ArrayLoader;
4. use Twig\Node\Expression\ConstantExpression;
5. use Twig\Node\Node;
6. use Twig\NodeTraverser;
7. use Twig\NodeVisitor\NodeVisitorInterface;
8. // here we parse the input to Twig AST Node
9. final class ReplaceTitleWithNameNodeVisitor
10.     implements NodeVisitorInterface
11. {
12.     public function enterNode(
13.         Node $node,
14.         Environment $env
15.     ): Node {
16.         // we look for "title"
17.         if (! $node instanceof ConstantExpression) {
18.             return $node;
19.         }
20.
21.         if ($node->getAttribute('value') !== 'title') {
22.             return $node;
23.         }
24.
25.         return new ConstantExpression(
26.             'name',
27.             $node->getTemplateLine()
28.         );
29.     }
30.
31.     public function leaveNode(
32.         Node $node,
33.         Environment $env
34.     ): ?Node {
35.         return $node;
36.     }
37.
38.     public function getPriority()
39.     {
40.     }
41. }
42. $nodeVisitor = new ReplaceTitleWithNameNodeVisitor();
43. $twigTraverser = new NodeTraverser(
44.     $environment,
45.     [$nodeVisitor]
46. );
47. $traversedNode = $twigTraverser->traverse($twigAstNode);

```



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 FriendOfPHP which is updated daily with meetups all over the world. Connect with him on twitter [@votrubaT](https://twitter.com/votrubaT).

<sup>13</sup> [phpstan-twig-analysis](https://github.com/phpstan-twig-analysis): <https://phpa.me/phpstan-twig>



The Web Developer's

## SECRET WEAPON!

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

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

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

### DevOps monitoring, for developers. \*gasp!\*

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

#### Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

#### Uptime Monitoring

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

#### Check-In Monitoring

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



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



# Bring Value To Your Code

Dmitri Goosens

Through its iterations, PHP has become as appropriate as any other language to express Domain-Driven Design and implement other concepts and patterns that are more complex. One of these is one of the most important building blocks in Domain Driven Design (DDD), the Value Object... But what is a Value Object? Is it only useful in DDD? How, where, and why should one use them? Let's try to check it out...

Eric Evans, in *Domain-Driven Design*.—also called *the Blue Book* or the DDD Bible—gives the following definition of a Value Object:

*An object that represents a descriptive aspect of the domain with no conceptual identity is called a VALUE OBJECT. VALUE OBJECTS are instantiated to represent elements of the design that we care about only for what they are, not who or which they are.*

And in his *Implementing Domain-Driven Design*, aka *the Red Book*, Vernon Vaughn dedicates a whole chapter to Value Objects and lists their characteristics:

*It measures, quantifies, or describes a thing in the domain; It should be immutable; It captures a whole value; It is replaceable; It prevents side-effects.*

That's all very DDD... But, as we'll see below, Value Objects are very useful even if one doesn't work with the Domain Driven Design paradigm.

## Identity... Not!

Evans highlights the key characteristic of a Value Object, that it is defined by its value and not by its identity.

If you plan to paint your house in, let's say, pink, you go out to the store and buy loads of buckets of pink paint. Back home, you just pick the first bucket and start painting. Although they are different buckets, what you really care about is the color it contains. The value of the bucket is defined not by its identity, but by the color it contains; therefore, all the buckets are equal.

In PHP, this translates into objects that have no `id` and one or more attributes that define their value. Side note: we are focusing on the PHP object for the moment, not how the data is stored in a database. This distinction is very important. We will address persistence later on.

As PHP developers, we all know at least one famous Value Object: `DateTimeImmutable`:

```
$now = new \DateTimeImmutable();
```

The `DateTimeImmutable` object has values for year, month, day, hour, minute, seconds, and timezone ... but not a single property that sets an identity.

With PHP 8.1, Enums were introduced, and they work pretty well as Value Objects:

```
enum Suit
{
    case Hearts;
    case Diamonds;
    case Clubs;
    case Spades;
}
```

As one can see, no identity... Value only.

**Side-note:** Enums are a little peculiar in regards to the identity characteristic. They are, in fact, Singletons; therefore, they will always return the same instance. So they kind of are their own identity value. This does not disqualify them as Value Objects. As one can read below, Enums definitely match all the Value Object's criteria.

Last but not least, we can also define our own custom Value Objects:

```
class EmailAddress
{
    public function __construct(
        public string $value
    ){
    }
}
```

Again... no identity and only a value.

**Side-note:** the absence of an identity is the main difference between a Value Object and an Entity, whether a Domain Driven Design Entity or an ORM Entity.

**Rule 0: A Value Object is a thing that defines itself by its value, not by its identity.**

## Descriptive

Another important keyword in Evans' definition above is *descriptive*. And Vaughn writes a Value Object **describes, measures, or quantifies** something.

To handle these characteristics, a Value Object has one or more attributes: (See Listing 1)

Listing 1.

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

In the above code listing, we have a class that is able to define a color with the RGBA model. When talking about the color of light, like on a screen: Red, Green, Blue, and Alpha are the four channels that define a color.

Every attribute of the Color class indicates each channel's exact intensity that is required to obtain the exact color we want:

```
$black = new Color(0, 0, 0, 1);
$white=new Color(255, 255, 255, 1);
$redTrans=new Color(255, 0, 0, .25);
```

Interestingly enough, instead of scalars, these attributes could also be other Value Objects. For instance: (See Listing 2)

In the above example, the VirtualPencil is defined by a Color, a Sharpness and a Hardness attribute, and all of them are themselves Value Objects.

**Rule 1: A Value Object has one or more attributes whose values define the Value Object. The attributes are either primitives or other Value Objects.**

## Immutable

To share the value object within an application and avoid any side effects,

it is highly recommended that they be immutable so that they can't change suddenly.

Doing so will avoid annoying, sometimes embarrassing, and always hard-to-debug issues like this: (See Listing 3)

As one sees, now Marc will end up with a blue mustache if we were to use the set color later on in a virtual pencil. And nobody would want Marc to have a blue mustache, or would we?

More seriously, this shows that sometimes an object, or any referenced variable, may get changed when passed on to another scope, without the original scope being aware. As we can see above, this may cause quite some trouble.

This can be easily prevented by making the Value Object's attributes immutable. Easy in PHP 8.1, with the latest readonly property: (See Listing 4 on the following page)

And will even be easier in PHP 8.2 with readonly classes: (See Listing 5 on the following page)

If you're not lucky enough to work with PHP's latest versions, you still can use private properties and public getter methods for the Value Objects, but obviously, no public setter methods.

**Side note:** *there are cases where one explicitly wants/needs mutable Value Objects. This is acceptable, but in that case, these Value Objects should **never** be shared.*

**Rule 2: Value Objects should almost always be immutable.**

## Whole Value

As Evans theorized DDD and talked about Value Objects, one often thinks he also invented them. But the origin of the Value Object appears to be the *Whole Value*, introduced by Ward Cunningham.

In short, a Value Object is either complete with all its attributes or not. In the color examples above, it would not make sense to leave out one of the

Listing 2.

```
1. enum Sharpness {
2.     case VerySharp;
3.     case NotSoSharp;
4.     case NotSharpAtAll;
5. }
6.
7. enum Hardness {
8.     case OhSoHard;
9.     case RegularHard;
10.    case Medium;
11.    case Soft;
12. }
13.
14. class VirtualPencil
15. {
16.     public function __construct(
17.         public Color $color,
18.         public Sharpness $sharpness,
19.         public Hardness $hardness,
20.     ) {
21.     }
22. }
23.
24. $virtualPencil = new VirtualPencil(
25.     new Color(0, 0, 0, 1),
26.     Sharpness::VerySharp,
27.     Hardness::OhSoHard,
28. );
```

Listing 3.

```
1. class VirtualPencilExtensionHandler
2. {
3.     public function __construct(
4.         public Color $color,
5.     ) {
6.     }
7.
8.     public function handleExtension(
9.         TheExtensionInterface $extension
10.    ): void
11.    {
12.        $extension->run($this->color);
13.    }
14. }
15.
16. class NastyExtensionThatChangesColor
17.     implements TheExtensionInterface
18. {
19.     public function run(Color $color)
20.     {
21.         $color->blue = 255;
22.         // ...
23.     }
24. }
```

## Listing 4.

```

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

```

## Listing 5.

```

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

```

channels, and even if we were to make the \$alpha attribute optional, it should never be null, as it would no longer be a valid color.

Therefore, it's highly recommended to instantiate a Value Object once with all its attributes, like the examples above. Setting all the readonly properties in the constructor makes it impossible to end up in an invalid state.

Now, in some cases, this is complicated. For more complex objects, we might need to convert data or extract it from a database, use a webservice or parse a file to gather the required information to inject into the Value Object or use. For instance, a Strategy Pattern to calculate the value of the attributes we want to inject into the Value Object.

In short, there are many use cases where it is not that simple to instantiate the object on the fly.

In those cases, it is recommended to use a factory within the Value Object itself or, when it gets really complex, in a dedicated class. (See Listing 6)

These factories may also be interesting if we added additional static constructors to the Value Objects. For instance, in our color example, we might need to be able to instantiate an object with other color notations: (See Listing 7 on the following page)

By the way, these static constructor methods allow for more explicit and expressive method names, which is always useful, even when not following the DDD principles. Therefore, it is very common to define private constructors and implement one or more static methods to initiate the Value Object, even if there is only one: (See Listing 8 on the following page)

An alternative to factory methods is to use an evolvable object: (See Listing 9 on the following page)

## Listing 6.

```

1. class Color
2. {
3.     // ...
4.
5.     public static function ofHexColor(
6.         string $hexColor,
7.         float $opacity
8.     ): self
9.     {
10.         $hexColor = ltrim($hexColor, '#');
11.         $parts = match(strlen($hexColor)) {
12.             3 => [
13.                 str_repeat(substr($hexColor, 0, 1), 2),
14.                 str_repeat(substr($hexColor, 1, 1), 2),
15.                 str_repeat(substr($hexColor, 2, 1), 2),
16.             ],
17.             6 => [
18.                 substr($hexColor, 0, 2),
19.                 substr($hexColor, 2, 2),
20.                 substr($hexColor, 4, 2),
21.             ],
22.             default => throw new InvalidColor(
23.                 sprintf('%s is not a valid CSS color', $hexColor)
24.             )
25.         };
26.
27.         return new self(
28.             hexdec($parts[0]),
29.             hexdec($parts[1]),
30.             hexdec($parts[2]),
31.             $opacity
32.         );
33.     }
34. }
35.
36. var_dump(Color::ofHexColor("#f00", 1));
37.
38. // Color Object
39. // (
40. //     [red] => 255
41. //     [green] => 0
42. //     [blue] => 0
43. //     [alpha] => 1
44. // )

```

As one can see in this very simple example, one can chain a series of more or less complex methods, each one generating a new valid Value Object, until we end up with the final Value Object we aimed to achieve. Please note that to get there, we did take care to define valid default values for the Color's four channels.

**Rule 3: Value Objects are whole objects and should be complete.**

## Listing 7.

```

1. class Color
2. {
3.     // ...
4.
5.     public static function ofHsvColor(
6.         int $hue,
7.         int $saturation,
8.         int $value,
9.         float $opacity
10.    ): self
11.    {
12.        // convert, validate, initiate here
13.    }
14.
15.    public static function ofCmykColor(
16.        int $cyan,
17.        int $magenta,
18.        int $yellow,
19.        int $black,
20.        float $opacity
21.    ): self
22.    {
23.        // convert, validate, initiate here
24.    }
25. }

```

## Listing 8.

```

1. class Color
2. {
3.     private function __construct(
4.         public readonly int $red,
5.         public readonly int $green,
6.         public readonly int $blue,
7.         public readonly float $alpha,
8.     ) {
9.     }
10.
11.    public static function ofRgbaColor(int $red,
12.        int $green,
13.        int $blue,
14.        float $alpha = 1.0
15.    ): self
16.    {
17.        // validation
18.        return new self($red, $green, $blue, $alpha);
19.    }
20. }

```

## Listing 9.

```

1. class Color
2. {
3.     public function __construct(
4.         public readonly int $red = 0,
5.         public readonly int $blue = 0,
6.         public readonly int $green = 0,
7.         public readonly float $alpha = 1,
8.     ) {
9.     }
10.
11.    public function withRed(int $red): self
12.    {
13.        return new self(
14.            $red,
15.            $this->blue,
16.            $this->green,
17.            $this->alpha
18.        );
19.    }
20.
21.    public function withBlue(int $blue): self
22.    {
23.        return new self(
24.            $this->red,
25.            $blue,
26.            $this->green,
27.            $this->alpha
28.        );
29.    }
30.
31.    // etc with Green & Alpha
32. }
33.
34. $color = new Color();
35. var_dump($color);
36. // Color Object
37. // (
38. //     [red] => 0
39. //     [green] => 0
40. //     [blue] => 0
41. //     [alpha] => 1
42. // )
43.
44. $otherColor = $color->withRed(255);
45. var_dump($otherColor);
46. // Color Object
47. // (
48. //     [red] => 255
49. //     [green] => 0
50. //     [blue] => 0
51. //     [alpha] => 1
52. // )

```



## Replaceable

If a Value Object is immutable (see above), one cannot simply change its value or the value of its attributes. That seems pretty obvious.

This means that when we need to change the represented value, we should completely replace it with a new Value Object. This is very easy to understand when using plain scalars:

```
$firstname = 'Eric';
$firstname = 'John';
```

In the above two lines, Eric did not become John, but the value of `$firstname`, that was initially set to Eric, was replaced with the value John (any resemblance with reality is a mere coincidence).

The same goes for Value Objects:

```
$firstname = new Firstname('Eric');
$firstname = new Firstname('John');
```

```
// and not
$firstname->value = 'John';
```

**Rule 4: Referenced Value Objects are not updated but replaced with new instances.**

## Side-effect-free

This is a good practice that comes from functional programming and is tightly related to the immutability and replacement principles explained above.

*Side-effect-free* means that any operation on an object, in this case, a Value Object, should not alter the object's state. (See Listing 10)

In the example above, the value of `$originalTotal` doesn't change. This remains true even when a method is called on it and a new Value Object, `$newTotal`, is created with a different value.

### Listing 10.

```
1. class Total
2. {
3.     public function __construct(
4.         public readonly int $value,
5.     ) {
6.     }
7.
8.     public function add(int $valueToAdd): self
9.     {
10.         return new self($this->value + $valueToAdd);
11.     }
12. }
13.
14. $originalTotal = new Total(2);
15. $newTotal = $originalTotal;
16. $newTotal = $newTotal->add(3);
17.
18. echo $originalTotal->value; // 2
19. echo $newTotal->value;    // 3
```

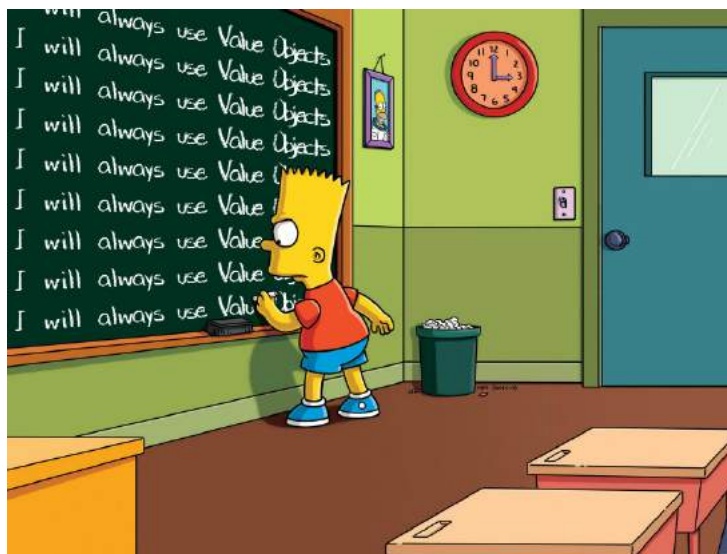
**Rule 5: Value Objects remain unaffected by the operations applied to them.**

## Ok Great... Now What?

Now that we more or less understand what a Value Object is and have seen a series of very simple PHP examples, next month we'll bring you part 2 and have a closer look at the benefits of using them.



*Dmitri is a Belgian backend PHP Dev and Architect, working at Ardenne-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](https://twitter.com/dgoosens)*





# Local Dev with Lando

Joe Ferguson

This month we're going to visit a topic near and dear to my heart: local development environments. I've been maintaining Laravel Homestead since late 2016. I have been a longtime tinkerer of just about every local development tool you've ever heard of, and they all have their strengths and weaknesses. Lando is just about the only tool I haven't had a chance to dive into, so join me in this dive into the world of local development with Lando.dev.

One Lando<sup>1</sup> feature which initially drew my curiosity was the claim that "Lando can isolate all your dev needs on a per-app basis". This sounds great, but I can show you long commit histories of tools and services with similar claims that have come and gone over the years from my projects. Lando will wrap our development environment services in Docker and use a proxy service to serve our application to a local development URL. Lando comes bundled with Docker Desktop on macOS and Windows, so you're ready to go out of the box on a fresh installation. Lando differs from Valet, where the services will be running in Docker containers instead of directly on your system. What sets Lando apart from Homestead is you're only virtualizing the necessities required to operate the application instead of the entire computer and operating system. Lando is going to eat a lot of resources, so beware; the more power your machine has will ensure a smoother runtime experience.

We'll configure a project with Lando, a member management system that utilizes Apache, PHP 8.0, MySQL, and Redis in production. Before we get too far ahead, we should start downloading and installing Lando. We'll download Lando from their official GitHub releases<sup>2</sup> and select our package preference. I'm running Lando on a first-generation M1 Apple Silicon mac mini with 16GB of RAM. Lando supports Linux, Windows, and macOS, which is a welcome feature of any local development tool. We all know developers rarely agree on operating system choices. When tools can support all of the options, it greatly reduces the friction teams may have when adopting new tools. Isn't this cross-platform compatibility the future we were all promised?

## Install Lando

Installation requires privileged access, so you will be prompted for your password as shown in Figure 1.

Afterwards, you should see that Lando has finished installation successfully and we now have access to the `lando` command in our CLI. (See Figure 2)

Running `lando init` to create our configuration will begin a series of questions we answer about our project so Lando can create the appropriate services we need. We'll select the

Figure 1.

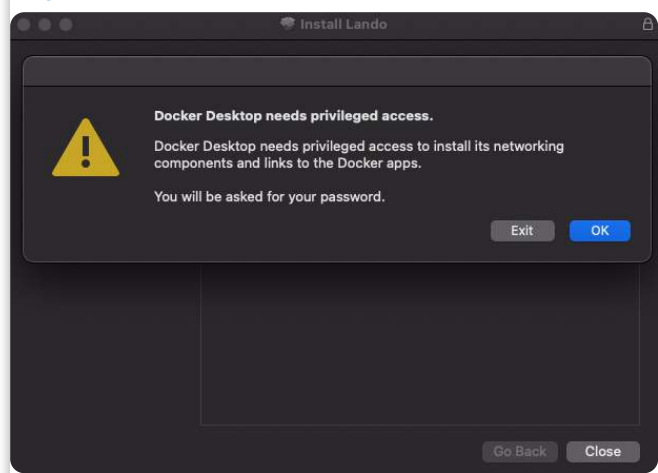


Figure 2.



current working directory for our project: Laravel, specify our public folder, and then name our application. (See Figure 3 on the next page)

The result of our `init` command outputs a `.lando.yml` configuration file. We need to override the defaults, which we can do by passing them in the config top-level item. We'll set

<sup>1</sup> Lando: <https://lando.dev>

<sup>2</sup> GitHub releases: <https://github.com/lando/lando/releases>

[illegible]

We'll also leverage Lando's built-in service to add our Redis service to our configuration.

```
name: rfid
recipe: laravel
config:
  webroot: public
  php: '8.0'
  database: mysql
  xdebug: true
  cache: redis
```

## Lando Start!

Depending on your connection speed and how many Docker images you may not already have on your system, you should arrive at the “Booomshakalaka!!!” screen indicating success on my M1 system. Taking just over three minutes to complete. (See Figure 5)

If you're curious about what's happening behind the scenes, we can run `docker ps` to see the services running: `rfid_rfid-redis_1`, `bitnami/mysql:5.7.29-debian-10-r51` our database, `devwithlando/php:8.0-apache-4` Apache/PHP, and `traefik:2.2.0` which is used to make all of the URL proxy magic happen without us having to understand reverse proxies and a lot of other logic that our time would be better

What if we forgot a service or want to change something? We can use this as an example of how to add a new service to our application: Mailhog for local mail processing. We'll update our `.lando.yml` configuration file:

```
services:
  mailhog:
    type: mailhog:v1.0.0
    portforward: true
    hogfrom:
      - appserver
```

### Listing 1.

1. Your app has started up correctly.
2. Here are some vitals:
- 3.
4. NAME rfid
5. LOCATION /Users/halo/Code/rfid
6. SERVICES appserver, database, cache, mailhog
7. APPSERVER URLS <https://localhost:64387>
8. <http://localhost:64388>
9. [http://rfid.lndo.site/](http://rfid.lndo.site:8080/)
10. <https://rfid.lndo.site/>
11. MAILHOG URLS <http://localhost:64386>

```
(~/Kado/Fid)/lindo-dev 771:5-11 no-remote)
lindo start
Let's get this party started! Starting ap proxy...
Creating network 'lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.default' with default driver
Creating volume 'lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.data' with default driver
Creating volume 'lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.home' with default driver
Pulling ca (dewhildlindo/1:4)...
  1. Pulling from dewhildlindo/1
    a948185d0d8: Pull complete
    885ed2c13b: Pull complete
Digest: sha256:13b2133829fc43778a26114f1d8316a07f98b50a26771bcf5511ecd4735
Status: Downloaded newer image for dewhildlindo/1:4
Creating lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.ca_1 ... done
outpsmp 37:00:22:40 INFO ==> Looks like you do not have a Lando CA yet! Let's set one up!
outpsmp 37:00:22:40 INFO ==> Trying to setup root CA with...
outpsmp 37:00:22:40 INFO ==> Lando/CA.Cert: /lando/certs/lindo_site.pem
outpsmp 37:00:22:40 INFO ==> Lando/CA.Key: /lando/certs/lindo_site.key
outpsmp 37:00:22:40 INFO ==> /lando/certs/site.pem not found... generating one
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
a is 65337 (0x00000000)
outpsmp 37:00:22:41 INFO ==> /lando/certs/lindo_site.pem not found... generating one
outpsmp 37:00:22:41 INFO ==> CA generated at /lando/certs/lindo_site.pem
Killing lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.ca_1 ... done
Going to remove lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.ca_1 ... done
Cleaning up resources for lindocontainer0k38hsoxwkbF232744ab0b9C604263c544502b4404e4.ca_1 ... done
==> Version 1 Landoes detected, attending upgrade...
Creating network 'lindoproxyhyper000g0nalfedfion.sudo' with driver "bridge"
Creating volume 'lindoproxyhyper000g0nalfedfion.proxy.config' with default driver
Creating volume 'lindoproxyhyper000g0nalfedfion.data.proxy' with default driver
Creating volume 'lindoproxyhyper000g0nalfedfion.home.proxy' with default driver
Pulling proxy (traefik/2.2.0)...
  2.2.0: Pulling from library/traefik
    29c5d0a048c1: Pull complete
    ff86e07d86: Pull complete
    c9d3c34f6: Pull complete
```

[illegible]

www.phparch.com \ November 2022 \ 17



#### Listing 2.

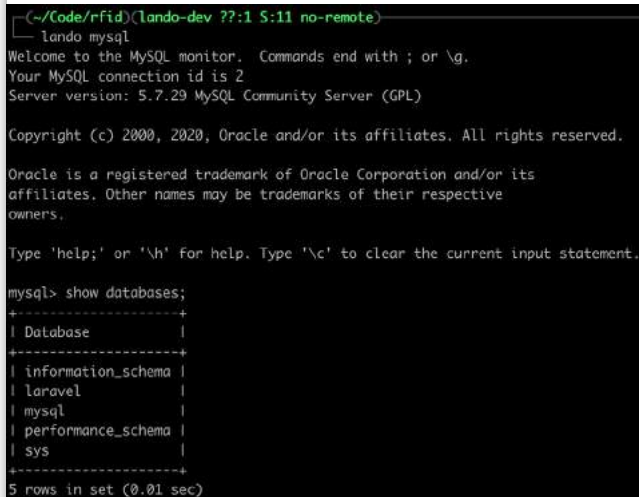
```
1. ...
2. APP_URL=https://rfid.lndo.site
3. APP_SERVICE=rfid.lndo.site
4.
5. DB_CONNECTION=mysql
6. DB_HOST=database
7. DB_PORT=3306
8. DB_DATABASE=laravel
9. DB_USERNAME=laravel
10. DB_PASSWORD=laravel
11. ...
```

2 minutes and resulted in adding our Mailhog URLs. (See Listing 1)

We'll also need to update our application's `.env` to point it at Lando's services: (See Listing 2)

We can use Lando's built-in `lando mysql` command to jump directly into a MySQL shell, which can be helpful for debugging our database. (See Figure 6)

Figure 6.



```
(~/Code/rfid)(lando-dev ??:1 5:11 no-remote)
lando mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.29 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| laravel |
| mysql |
| performance_schema |
| sys |
+-----+
5 rows in set (0.01 sec)
```

#### lando mysql command usage

If you would prefer to use a graphical SQL client you can retrieve the port information from `lando info`, which returns information about the services of the project. (See Listing 3)

We can see our database service is listening for *external connections* on IP 127.0.0.1 and port 64802. We can plug these values into PhpStorm, for example: (See Figures 7 and 8 on the next page)

We can see our database from PhpStorm by connecting it to Lando's MySQL service.

#### Listing 3.

```
1. {
2.   service: 'database',
3.   urls: [],
4.   type: 'mysql',
5.   healthy: true,
6.   internal_connection:
7.     { host: 'database', port: '3306' },
8.   external_connection:
9.     { host: '127.0.0.1', port: '64802' },
10.  healthcheck: 'bash -c "[ -f .mysql_initialized ]"',
11.  creds:
12.    { database: 'db', password: 'pw', user: 'user' },
13.  config: {},
14.  version: '5.7',
15.  meUser: 'www-data',
16.  hasCerts: false,
17.  hostnames: [ 'database.rfid.internal' ]
18. },
```

## Using and Abusing Lando

Since we're starting fresh in a new database, we need to run our migrations and database seeds. Traditionally, we'd do this with `php artisan` commands; however, to run them in the correct container, we can use `lando artisan migrate` and `lando db:seed` to set up our application. If something goes wrong, we can trash everything we've done so far via `lando destroy`, which will delete the database and anything in the application's storage folder. If you want something less destructive, `lando stop` will stop all running services. If you've reached this far and are done, you can also `uninstall Lando`<sup>4</sup> easily and return to your preferred local development environment.

Just as we added `lando` to our `artisan` and `db` commands, we can use `lando composer install` or other Composer commands on our project, just as we typically would to install dependencies or adjust other options.

Another common task is exporting and importing databases for our project. We can use `lando db-export filename` to export our DB to a local file. To import a SQL export into our database, we would use `db-import filename`. Neither of these commands requires us to rebuild the environment; you should see the changes immediately after the import.

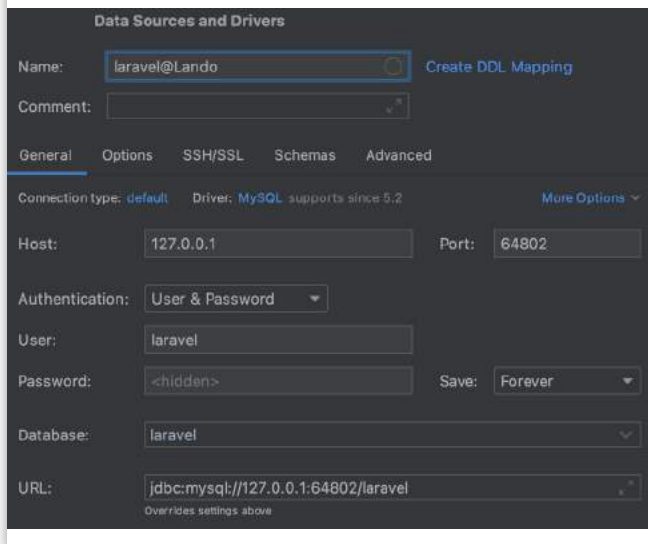
What about NPM and Node—how do we run `npm run dev` to build our front-end assets? We can add a NodeJS service to our `.lando.yml` and provide a command to run, such as `npm run dev`. In my use cases, I still typically run NPM commands directly on my files instead of connecting to the running container such, as `lando artisan` and `lando php` commands would do. This is a personal preference, and you should use what makes the most sense for you. Our updated `.lando.yml` adding Node might look like this: (See Listing 4 on the next page)

<sup>4</sup> Uninstall Lando: <https://phpa.me/lando-dev>



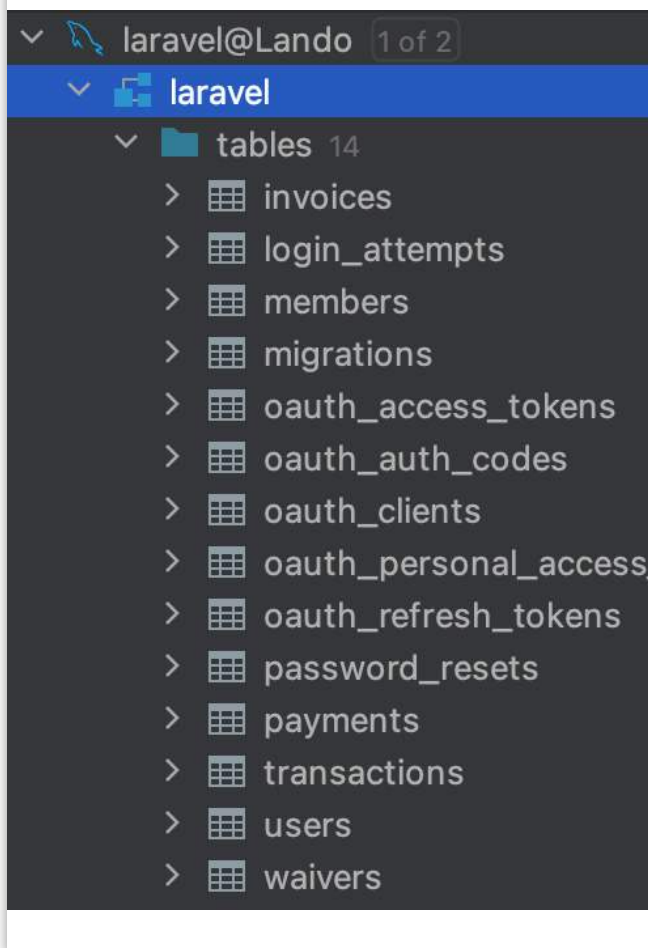
Since we've changed our configuration file, we need to run `lando rebuild -y` to apply the changes, and we'll see another port added for our Node URLs: (See Listing 5 on the next page)

Figure 7.



The screenshot shows the 'Data Sources and Drivers' configuration window. The 'Name' field is 'laravel@Lando'. The 'Comment' field is empty. The 'General' tab is selected. The 'Connection type' is 'default' and the 'Driver' is 'MySQL, supports since 5.2'. The 'Host' is '127.0.0.1' and the 'Port' is '64802'. The 'Authentication' is 'User & Password'. The 'User' is 'laravel' and the 'Password' is '<hidden>'. The 'Database' is 'laravel'. The 'URL' is 'jdbc:mysql://127.0.0.1:64802/laravel'. The 'Save' dropdown is set to 'Forever'.

Figure 8.



You'll want to add this Node service to your configuration if you want to run these commands inside Lando. There are some things to consider with this as your `node_modules` PATH variable may not be what you'd expect since they'll be paths for the container services and not what you might be used to. Experiment and see what come up with; there's certainly the ability to override and change behaviors around Lando. I would expect just about any project would be able to get up and running quickly.

But I need MSSQL! Lando's got you covered with a MSSQL plugin<sup>5</sup>. But we use a highly specialized Varnish caching layer! Lando has you covered with a Varnish<sup>6</sup> plugin. What makes me excited about Lando is it makes adding these services trivial. When you remove the complexity of configuring and installing services, you enable developers to use those tools and experiment with other solutions. Lando isn't just for Laravel, whether supported via Services or Recipes, you can easily use WordPress, Joomla, Drupal, Symfony, and just about any other PHP framework of interest. We've only configured one application with Lando; you can keep adding sites or start removing them as much as you'd like to and as much as you have the system resources to run all of the services.

What's the result of our Lando test drive? One added file to our project, which is our `.lando.yml` file, and our entire development environment is created from there. This is an incredibly powerful tool, as even something such as Sail requires adding supplemental files and configuration templates to your project folder. We're also using many defaults, so we're not near Lando's limits, and our modest M1 2020 system is not showing any signs of stress, such as other

Listing 4.

```
1. services:
2.   mailhog:
3.     type: mailhog:v1.0.0
4.     portforward: true
5.     hogfrom:
6.       - appserver
7.   node:
8.     type: node:16
9.     command: npm run dev
```

Listing 5.

```
1. NAME      rfid
2. LOCATION  /Users/haLo/Code/rfid
3. SERVICES  appserver, database, cache, mailhog, node
4. APPSERVER URLS  https://localhost:50803
5.             http://localhost:50804
6.             http://rfid.lndo.site:8080/
7.             https://rfid.lndo.site/
8. MAILHOG URLS  http://localhost:50799
9. NODE URLS    http://localhost:50800
```

<sup>5</sup> MSSQL plugin: <https://docs.lando.dev/mssql/>

<sup>6</sup> Varnish: <https://docs.lando.dev/varnish/>



applications lagging and becoming unresponsive. Granted, you may need to add custom configuration files to override PHP configurations like you would with Laravel Sail. Lando has been a breath of fresh air for someone who has a lot of opinions on how local development environments work.

Lando is a pretty fantastic local development environment tool. It allows you to leverage the complexity of Docker without having to be an expert. The tools integrate well with typical PHP development workflows, and the documentation is concise while descriptive. The most important part of a local development environment is that Lando got out of my way. I felt immediately productive and didn't have to struggle. If you run into trouble, the docs also contain a troubleshooting guide and a list of known issues you might run

into while using Lando. Lando is a great solution to a very difficult problem: keeping developers happy while building applications. I would certainly recommend it as a tool in your toolbox.

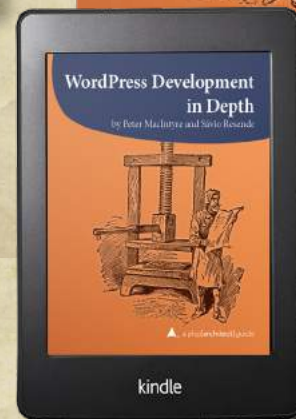
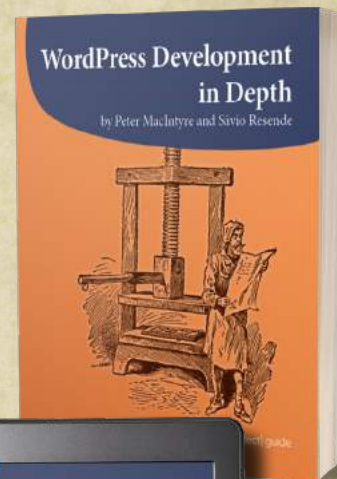


*Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They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

## Build a custom, secure, multilingual website with WordPress

Written by PHP professionals Peter MacIntyre and Sávio Resende, this book distills their experience building online solutions for other WordPress site builders, developers, and designers to leverage and get up to speed quickly.

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





# Direct Object References

Eric Mann

Building APIs in PHP often exposes us to the potential of obscure bugs that can otherwise compromise the security of our application. Building too pure of an API—and relying on clients to provide too much information about the objects they’re referencing—is one such risk.

Earlier versions of the OWASP Top Ten enumerated the specific risk of insecure direct object references. In a nutshell, this is exposing an object identifier directly as part of an API pattern. Such exposure makes adversarial surveillance easy.

And when it comes to software development, “easy” is a bad word. For many reasons.

## Vulnerable Code

Assume your application is split into two components. On the front-end is a React-based single page application (SPA). It’s fast, well-polished, and presents a fantastic user experience. On the back-end you have a Laravel-powered API that presents routes for all of the data the SPA needs. The API is just that, a set of REST-ful endpoints rather than a full Laravel application with view rendering. We effectively just need routes and controllers.

Rather than discuss the entire application, let’s focus on accounts. We want to provide a set of endpoints that allow creating and managing user accounts. Everything will stem from a single controller, and the routing can handle common CRUD (create, read, update, delete) functionality in a straightforward manner:

```
use App\Http\Controllers\AccountController;

Route::controller(AccountController::class)->group(function () {
    Route::post('/account', 'create');
    Route::get('/account/{id}', 'read');
    Route::patch('/account/{id}', 'update');
    Route::delete('/account/{id}', 'delete');
});
```

The controller itself then handles incoming requests appropriately: (See Listing 1)

Those of you who focus in API security can probably already identify *at least* one problem with the controller above. There is absolutely no authentication! Anyone can create an account, fetch an account, change or delete an account. This is inherently dangerous and we *must* bake in authentication before moving forward.

Thankfully, authentication comes somewhat standard in Laravel<sup>1</sup>. We can protect these read, update, and delete endpoints merely by adding the auth middleware to them.

### Listing 1.

```
1. namespace App\Http\Controllers;
2.
3. use App\Models\Account;
4. use Illuminate\Http\Request;
5.
6. class AccountController extends Controller
7. {
8.     public function create(Request $request)
9.     {
10.         $account = new Account;
11.
12.         // ...Populate the model
13.         $account->save();
14.
15.         return $account;
16.     }
17.
18.     public function read(int $id)
19.     {
20.         return Account::findOrFail($id);
21.     }
22.
23.     public function update(Request $request, int $id)
24.     {
25.         $account = Account::findOrFail($id);
26.
27.         // ...Populate changes onto the model
28.         $account->save();
29.
30.         return $account;
31.     }
32.
33.     public function delete(int $id)
34.     {
35.         return (Account::findOrFail($id))->delete();
36.     }
37. }
```

For example:

```
// ...
Route::post('/account', 'create');
Route::get('/account/{id}', 'read')->middleware('auth');
// ...
```

<sup>1</sup> Laravel Authentication: <https://phpa.me/laravelauthentication>



Now, anyone trying to reach a sensitive endpoint—one where they could read or manage their account data rather than merely creating an account—the application will only work if they're authenticated. On the surface, it feels like we've fixed the problem.

But we haven't.

The endpoints we've defined use the account ID directly within the request path to identify the account we want to manage. Even though they're now authenticated, Laravel does not enforce that a user is editing their own account.

Any user could leverage a web inspector (via Chrome or Firefox or similar) to identify the paths used by the application. They could then just as easily modify the `id` parameter to read data from other users' accounts or, more damaging, make changes.

If our database is using integer IDs for users this is even more problematic. An attacker could use these endpoints to enumerate *all* user accounts and potentially make off with a comprehensive portfolio of our customers' data! All because we directly expose an object reference (the account ID) in an authenticated yet insecure fashion.

## Quick Fixes

Luckily, this particular bug is a quick fix, particularly in Laravel.

First, we need to define a check that compares the authenticated user's ID with the ID of the account they wish to modify. Then we apply that check to our authenticated routes

Listing 3.

```
1. namespace App\Http\Controllers;
2.
3. use App\Models\Account;
4. use Illuminate\Http\Request;
5. use Illuminate\Support\Facades\Gate;
6.
7. class AccountController extends Controller
8. {
9.     // ...
10.
11.     public function read(Request $request, int $id)
12.     {
13.         if (!Gate::allows('get-account', $id)) {
14.             abort(403);
15.         }
16.
17.         return Account::findOrFail($id);
18.     }
19.
20.     // ...
21. }
```

Listing 2.

```
1. use App\Models\User;
2. use Illuminate\Support\Facades\Gate;
3.
4. /**
5.  * Register any authentication / authorization services.
6.  *
7.  * @return void
8.  */
9. public function boot()
10. {
11.     $this->registerPolicies();
12.
13.     Gate::define('read-account',
14.         function (User $user, int $accountId) {
15.             return $user->id == $accountId;
16.         }
17.     );
18.
19.     Gate::define('update-account',
20.         function (User $user, int $accountId) {
21.             return $user->id == $accountId;
22.         }
23.     );
24.
25.     Gate::define('delete-account',
26.         function (User $user, int $accountId) {
27.             return $user->id == $accountId;
28.         }
29.     );
30. }
```

to prevent (potentially) malicious misuse. In Laravel, we do this by defining a series of Gates<sup>2</sup>: (See Listing 2)

We then utilize these gates within our controller methods and, when they fail, abort the operation: (See Listing 3)

It is generally possible to retrieve the account from our data store *first* and pass the entire object into the gate. This would allow us to respond with a 404 if the account doesn't exist. However, doing so would inadvertently leak information to an attacker—a 404 tells them an account doesn't exist and a 403 would tell them it does. There's not much they could do with that information, but when it comes to security we want to lock down *any* potential leaks in information.

If for some reasons you don't want to (or can't) use Gates, you can store the authenticated user's ID in a session variable then check within the controller that the route requested matches. For example: (See Listing 4 on the next page)

## Keeping Ahead

Secure software is easy to write. Insecure software is even easier. The fundamental difference between the two is your

<sup>2</sup> Gates: <https://laravel.com/docs/9.x/authorization>





## Listing 4.

```

1. namespace App\Http\Controllers;
2.
3. use App\Models\Account;
4. use Illuminate\Http\Request;
5.
6. class AccountController extends Controller
7. {
8.     // ...
9.
10.    public function read(Request $request, int $id)
11.    {
12.        $userId = $request->session()->get('user_id');
13.
14.        if ($userId !== $id) {
15.            abort(403);
16.        }
17.
18.        return Account::findOrFail($id);
19.    }
20.
21.    // ...
22. }

```

mindset: how often do you pause to consider and model potential threats<sup>3</sup> to your application and its security?

<sup>3</sup> consider and model potential threats:  
<https://phpa.me/articlesecuritycorner>

Today's focus is on a particular member of the OWASP Top Ten—Insecure Direct Object References. It's a potentially catastrophic error that's remarkably easy to make. But the possible human errors in your code are legion and could introduce even worse vulnerabilities if the impact of any operation is not carefully considered.

To keep ahead of the threats, make sure you spend adequate time thinking about and protecting against this kind of threat. Bake threat modeling into code review and invest time revisiting old code to ensure bugs haven't crept their way back into the stack long after a deployment.



*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*

## Related Reading

- *Security Corner: Broken Authentication*, August 2022.  
<https://phpa.me/security-aug-2022>
- *Security Corner: Surviving Cybersecurity*, September 2022.  
<https://phpa.me/security-sept-2022>
- *Security Corner: Cybersecurity Checkup*, October 2022.  
<https://phpa.me/security-oct-2022>

## 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>





# PSR-11: Container Interface

Frank Wallen

In this issue, we'll discuss the Container Interface, PSR-11. The Container's job is to act as a registry and return a service object identified by a unique key or name. Many modern frameworks and codebases implement Dependency Injection as a form of inversion of control, where the client requesting the service object does not need to know how to construct it. Usually, containers return service objects, but it could really be any valid object in the application. The responsibilities of the Container are configuring and fetching entries, leaving it up to the framework to fetch the necessary objects to compose the application.

The specification for the Container is straightforward:

- **Entry Identifiers** — Any PHP-legal string of at least one character. The identifier should not be expected to describe the returned object. However, it is common practice to apply semantic meaning either by a descriptive identifier or the fully qualified name of the object (such as class name with namespacing).
- **Reading from a Container** — The `ContainerInterface` defines two public methods: `get` and `has`. `get` has a mandatory argument, the entry identifier. `get` can return anything or throw a `NotFoundExceptionInterface` if the identifier is not known. The returned object SHOULD be the same value on successive calls. However, this is not a requirement, and a different value may be returned based on the implementor design or user configuration.
- `has` takes a single argument: an entry identifier. It MUST return `true` if the identifier is known to the container; otherwise, it MUST return `false`. If `has` returns a `false` for an identifier, then `get` MUST throw a `NotFoundExceptionInterface` for that identifier.

The Container, as recommended by PSR 11, should not be treated as a service locator and passed to the constructor of the object, where the object's dependencies could then be retrieved. This is because it is less interoperable, meaning that the code is now only compatible with the Container PSR, limiting the choices for using a different container library. Here's an example of using the Container as a Service Locator: (See Listing 1)

## Listing 1.

```
1. class ServiceLocatorExample
2. {
3.     protected $service;
4.
5.     public function __construct(ContainerInterface $cont)
6.     {
7.         $this->service = $cont->get('service');
8.     }
9. }
```

This class is now directly coupled to using PSR 11's `ContainerInterface` and cannot use a different container library that does not implement that interface. PSR 11 points out several additional reasons why this is not recommended:

- The service object registered in the container must now be named 'service' and could conflict with another registered entry.
- It is harder to test. In the test environment, a mocked service object would have to be registered in the container.
- It is not clear in the code what `ServiceLocatorExample` requires as the service object; the dependency is hidden.

The recommendation is to do something similar to:

```
class DependencyInjectionExample
{
    public function __construct(
        protected DatabaseServiceInterface $dbService
    ) { }
```

The framework or application then provides the requested service object (`DatabaseServiceInterface`), retrieved from the Container, and is free to use a different container library. `DependencyInjectionExample` clearly defines the service object it depends on. In the test environment, the user can now mock an object with `DatabaseServiceInterface` without registering the mocked object in the container.

There are cases where the container must be passed to the constructor. Some good examples provided by PSR-11 Meta Document<sup>1</sup> include routers and factories. These are cases where the Container is necessary as it will be used to provide the dependencies. In the router example, the Entry Identifier will be derived from the URL and passed to the Container to fetch a controller or response object: (See Listing 2).

Consider the following factory example: (See Listing 3).

The factory's responsibility is to return a `DocumentRepository` that MUST have access to the document store. Therefore the factory requires the Container to provide the document

<sup>1</sup> PSR-11 Meta Document: <https://www.php-fig.org/psr/psr-11/meta/>



store service to the repository. In both examples, the router and factory act as service locators.

How the container will handle returning objects is out of the scope of PSR 11. There are many approaches, some of which include:

- The Container acts as a factory, creating new objects.
- The Container implements `ArrayAccess` and stores objects in an array.
- The Container stores configurations for objects and their definitions and returns the prepared objects.
- The Container uses auto-wiring, where, through Reflection, the container can identify the dependencies called for in `__construct()` and attempt to provide them.

## Conclusion

Choosing or creating a container depends greatly on the needs of the application and what the environment provides. If your application may be short on memory, using an array to store all the registered objects may be the wrong choice, and a factory may be better suited. However, an array could provide the same object on successive calls, returning the same instance and saving memory by not creating a new one. Take a look at existing container libraries to see their approach and strategy. Here are some good examples (order does not indicate priority or value) for further reading:

- Container<sup>2</sup> by The PHP League
- PHP-DI<sup>3</sup>
- Pimple<sup>4</sup>

## Related Reading

- *PSR Pickup: PSR-6 Caching Interface* by Frank Wallen, September 2022.  
<https://phpa.me/psr-sept-2022>
- *PSR Pickup: PSR-7 HTTP Message Interface* by Frank Wallen, July 2022.  
<https://phpa.me/psr-jul-2022>
- *PSR Pickup: Psr-3 Logger Interface* by Frank Wallen, May 2022.  
<https://phpa.me/2022-05-psr>
- *PSR Pickup: PSR 12 Extended Coding Style Standard* by Frank Wallen, April 2022.  
<https://phpa.me/2022-04-psr>

### Listing 2.

```
1. class Router
2. {
3.     public function __construct(
4.         protected ContainerInterface $container
5.     ) {
6.     }
7.
8.     public function handle($request)
9.     {
10.         $identifier = $this->getControllerIdentifier(
11.             $request->getUrl());
12.         $controller = $this->container->get($identifier);
13.         // ...
14.     }
15. }
```

### Listing 3.

```
1. interface FactoryInterface
2. {
3.     public function newInstance();
4. }
5.
6. class DocumentFactory implements FactoryInterface
7. {
8.     public function __construct(
9.         protected ContainerInterface $container
10.     ) {
11.     }
12.
13.     public function newInstance(): DocumentRepository
14.     {
15.         return new DocumentRepository(
16.             $this->container->get('document_store')
17.         );
18.     }
19. }
```



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](https://twitter.com/frank_wallen)

<sup>2</sup> Container: <https://container.thephpleague.com>

<sup>3</sup> PHP-DI: <https://php-di.org>

<sup>4</sup> Pimple: <https://github.com/silexphp/Pimple>



# Making Our Own Web Server: Part 2

Chris Tankersley

One of the biggest laments of PHP is that it does not have a dedicated web server and instead relies on an external program like Apache or NGINX to handle the incoming request. Modern versions of PHP do ship with a development server, but that server is not recommended for production use at all. This is not just because it is normally single-connection only (though there are flags to make it multi-threaded) but because there are better tools to handle the web serving—like Apache and NGINX.

Last month we looked at exactly what it takes to make our own web server. While we did not replicate exactly how something like Apache works, we built a single-threaded, single-connection “server” that can accept an incoming connection and return a response. It works well enough, but we can do better.

## Handling Concurrent Connections

In our current server, we create a socket and then just listen on that socket until someone connects. This connection will then be the only traffic we look at until we finish it. If another client attempts to connect, they will either be blocked, and the connection will not work, or stall until the connection opens up, depending on the socket config. Either way, for the end user, only one connection can be handled at once.

C, the language underpinning PHP itself, has a solution for this. The `select()` method in C is exposed to PHP userland code as `socket_select()`<sup>1</sup>. `socket_select()` allows you to define an array of sockets for reading from and writing to, and provision these sockets to allow multiple connections. We can then loop over these sockets and read from them just like we did before.

For the most part, the initial setup is exactly the same as before. We are going to create a socket, bind it to an address, and then start to listen. The difference will be in how we read from, and handle, the socket connection after this.

```
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
socket_set_option($socket, SOL_SOCKET, SO_REUSEADDR, 1);
socket_bind($socket, '0.0.0.0', 8080);
socket_listen($socket);
```

We are also going to use `socket_set_option()`<sup>2</sup> to set an additional option on our script, `SO_REUSEADDR`. Normally, this is allowed in networking programs to allow an address to bound to immediately if we need to kill and restart our server quickly. If we did not set this option, the address would be held and block our restart for anywhere from 30-120 seconds.

`socket_select()` takes three parameters: an array of sockets to listen on, an array of sockets to write to, and an array of sockets with exceptions. We only have one socket to listen on so the first array will be a single element being the socket we created. The write and exception arrays will be empty as we are not explicitly writing to any sockets directly, and we do not start with any sockets that have issues.

```
$originalReadSockets = [$socket];
$writeSockets = [];
$exceptSockets = [];
```

You may notice that we called the initial set of sockets to read `$originalReadSockets`. We want to ensure that we always read our initial socket, and as connection comes in, they will be opened and added to the list of other sockets to read from. Each time we go through our loop, however, we will want to reset and make sure we re-read our initial socket.

Speaking of the loop, what exactly will we be doing?

Using an infinite loop, we will reset the list of sockets to read back to `$originalReadSockets`. We will then call `socket_select()`, which will return a number of sockets that have “changed,” which, in our case, means they are ready to be read from. If we have changed sockets, a new request has come in. We will accept this connection, read the request, do our work, and then return a response. (See Listing 1)

Ultimately, we are doing the same thing before, but the difference is in how we work with the network stack. Let’s break down the different blocks of code to see what is happening.

```
$readSockets = $originalReadSockets;
$numChanged = socket_select(
    $readSockets, $writeSockets, $exceptSockets, 0
);
if ($numChanged === false) { continue; }
if ($numChanged === 0) { continue; }
```

Here we reset ourselves and make sure that we look at the original socket. We then check to see if any sockets are ready to read from, write to, or are now in an error state. The 0 in the fourth parameter tells `socket_select()` to run with no

1 `socket_select()`: <https://phpa.me/manuelfunctionsocket>

2 `socket_set_option()`: <https://phpa.me/phparch>





## Listing 1.

```
1. use Laminas\Diactoros\Request\Serializer as ReqSerializer;
2. use Laminas\Diactoros\Response;
3. use Laminas\Diactoros\Response\Serializer as Reserializer;
4.
5. while (true) {
6.     $readSockets = $origReadSockets;
7.     $numChanged = socket_select(
8.         $readSockets, $writeSockets,
9.         $exceptSockets, 0
10.    );
11.    if ($numChanged === false) { continue; }
12.    if ($numChanged === 0) { continue; }
13.
14.    foreach ($exceptSockets as $badSocket) {
15.        echo 'Closing bad socket' . PHP_EOL;
16.        socket_close($badSocket);
17.        $unset = array_search($badSocket, $readSockets);
18.        unset($readSockets[$unset]);
19.        $unset = array_search($badSocket, $exceptSockets);
20.        unset($exceptSockets[$unset]);
21.    }
22.
23.    if (in_array($socket, $readSockets)) {
24.        $newSocket = socket_accept($socket);
25.        $origReadSockets[] = $newSocket;
26.        $unset = array_search($socket, $readSockets);
27.        unset($readSockets[$unset]);
28.    }
29.
30.    foreach ($readSockets as $curSocket) {
31.        $data = socket_read($curSocket, 1024);
32.        if ($data === false) {
33.            $unset = array_search($curSocket, $origReadSockets);
34.            unset($origReadSockets[$unset]);
35.            continue;
36.        }
37.
38.        $data = trim($data);
39.        if (! empty($data)) {
40.            $request = ReqSerializer::fromString($data);
41.            $response = (new Response())->withStatus(200);
42.            $msg = 'You requested ' . $request->getUriString() .
43.                ' with verb ' . $request->getMethod();
44.            $response->getBody()->write($msg);
45.            $responseString=ResSerializer::toString($response);
46.
47.            socket_write($curSocket, $responseString);
48.            socket_close($curSocket);
49.            $unset = array_search($curSocket, $origReadSockets);
50.            unset($origReadSockets[$unset]);
51.        }
52.    }
53. }
```

timeout. We will check for a changed socket and immediately return the result.

If false is returned, we ran into an error reading. We are not currently handling that, but instead just trying again immediately. If no sockets have changed and we get back a 0, there is nothing to read from. We continue the loop and immediately check again.

```
foreach ($exceptSockets as $badSocket) {
    echo 'Closing bad socket' . PHP_EOL;
    socket_close($badSocket);
    $sock = array_search($badSocket, $readSockets);
    unset($readSockets[$sock]);
    $sock = array_search($badSocket, $exceptSockets);
    unset($exceptSockets[$sock]);
}
```

If we had at least one socket change, we first need to check if we have any sockets that encountered an error. We check this list first as our main socket may have encountered an error, and we need to close it. If our main socket has errored, we remove it from the list so that we do not attempt to read from it or work with it later in the loop.

```
if (in_array($socket, $readSockets)) {
    $newSocket = socket_accept($socket);
    $originalReadSockets[] = $newSocket;
    $sock = array_search($socket, $readSockets);
    unset($readSockets[$sock]);
}
```

Now we can check to see if our main socket connection is in the list of sockets to read. If so, we use `socket_accept($socket)` to read as we did in our original server. When we perform the read, we get a new socket object, adding it to the list to read from in a few moments. We then remove the original socket from the read list as its job is currently finished.

```
foreach ($readSockets as $currentSocket) {
    // Read each socket
    // If there is data, transform to PSR-15 and process
}
```

I have shortened the next block of code as it works like the original server did, with the main difference is we may want to loop through any remaining sockets to read from. Each of those sockets will call `socket_read()` and get the incoming HTTP request as a raw string. We will then turn it into a PSR-7 request and generate a new PSR-7 response. This response is turned back into a string, and we write it back up the socket. Since we have sent a response, we close the socket.

Where this differs from the original implementation is how `socket_select()` and the underlying C code are handle how sockets are watched and read from. We are not being terribly efficient by running through the full read-process-write procedure all at once. Still, the code will now handle incoming connections better in the backend as `socket_select()` is much better at polling than our original userland code.



## Executing Custom Code

In a way, we have completed a basic web server. It may not be as performant as a true web server, but at the same time, we now have a way to execute code from our own long-running process. Let's take this idea a bit further and be able to execute PHP code.

When using something like Apache or NGINX, an external PHP script is executed by the Zend Engine. Our server is written in PHP, so we could execute PHP without invoking any external scripts. We could wire a route directly to some callable code. Since we are already creating PSR-7 requests to make the incoming requests easier to work with and making PSR-7 responses to make it easier to return a valid response, why not extract that code into callable objects?

In a modern web application, it is not uncommon to use design patterns like Model-View-Controller (MVC) or Action-Domain-Responder (ADR) to structure the business logic for code. Many frameworks pass request objects to these controller types and get response objects back. We have the request and response objects; we just need to match a path to a controller.

```
use Laminas\Diactoros\Request\Serializer as ReqSer;
use Laminas\Diactoros\Response\Serializer as ResSer;
...
$request = ReqSer::fromString($data);
$response = processRequest($request);
$responseString = ResSer::toString($response);
```

If we take our original code that parses the incoming data, we can remove the part where we created the response and replace it with a call to a method. For our example, let's just call it `processRequest()`. We will pass in the request we just made, and it will return a response. (See Listing 2)

If we use ADR as a way to handle our business logic, we can create two handlers. One will handle our homepage, and the other will handle any requests for which we do not have a route. We will make these invocable objects, and the `__invoke()` method will take a PSR-7 request and return a PSR-7 response. Each handler will be able to work with the request as needed and encapsulate any needed code.

`processRequest()` will extract the URI from the PSR-7 request we created and match it against a list of routes. Each route has a corresponding handler. If we find a match, create a new instance of the handler and invoke it. If we do not find a match, make a default 404 handler and invoke it instead. In either case, we should get a PSR-7 response we can return.

Since `processRequest()` returns a PSR-7 response from the handlers, we pass that back, turn it back into a string, and then write it back up the socket.

We can now execute proper PHP code attached to a route using PSR standards. We can turn this into a class instead of global-level loops and methods to make things cleaner. To save some digital page space, you can check out <https://github.com/dragonmantank/waiter-http><sup>3</sup>. This is an implementation

of the code we have been discussing but in an object-oriented form.

The library has a few additional niceties, like working with any PSR-15 middleware handler to parse and handle the more complex generation of responses. It also better handles signals to allow you to stop and restart the service. The core idea is all based on the code we have gone over in this and the previous article.

## Downsides to Our Server

In a way, we have created a pretty basic and stable server that can handle many concurrent requests. Static files can be served, and we can directly execute PHP code and return a valid response. Much like the PHP development server, this code is not necessarily ready for prime time.

One of the biggest issues is that, while we can support multiple connections at once, our server does not handle the global state properly at all. The server is single-threaded, so we can still call any code we want, but the structure of the application necessitates a complete rethinking of how we architect a PHP application.

### Listing 2.

```
1. use Laminas\Diactoros\Response;
2. function processRequest(
3.     RequestInterface $request
4. ): ResponseInterface {
5.     $paths = ['/ => HomepageAction::class];
6.
7.     if (in_array($request->getUri()->getPath(), $paths)) {
8.         $class = $paths[$request->getUri()->getPath()];
9.         return (new $class())($request);
10.    }
11.    return (new FourOfFourAction())($request);
12. }
13.
14. class HomepageAction {
15.     function __invoke(
16.         RequestInterface $request
17.     ): ResponseInterface {
18.         $response = (new Response())->withStatus(200);
19.         $response->getBody()->write('<h1>Hello World!</h1>');
20.         return $response;
21.     }
22. }
23.
24. class FourOfFourAction {
25.     function __invoke(
26.         RequestInterface $request
27.     ): ResponseInterface {
28.         $response = (new Response())->withStatus(404);
29.         $response->getBody()->write('<h1>Not Found!</h1>');
30.         return $response;
31.     }
32. }
33. }
```

<sup>3</sup> <https://github.com/dragonmantank/waiter-http>



In a traditional web application, each request is handled by the traditional “start, execute, stop” of PHP’s interpreter life-cycle. There is no state shared between requests, even requests from the same client. Technologies like cookies and sessions store data outside of PHP and act like catalogs for finding this state information later.

Doing so causes a problem for our code. If we start a session, the same session will be open and visible for anyone making a connection to the application. If we log in a user and store their info in a session, a completely different client that connects will access the same info because PHP opened a session, and this session stays open until the script ends. Our script is not designed to end, however. This is a pretty significant security issue.

We also have an issue with the overall global state of the application. Much of the written PHP code out there assumes that each request is a blank slate, so each request fully bootstraps the application. Various settings are read in, database lookups happen to get initial values, and then the request is routed. Since our application code starts and never ends, we need to think about how we bootstrap our code.

This is not unique to how we built the server but is a general shift in thinking when your web application is now persistent. If you were to build an application in Python or NodeJS, you would have the same general problem. We can no longer just arbitrarily store user data in sessions (unless we retool how sessions work), and we have to be very careful about anything we leave in the global namespace.

The big issue, even with cleaner code implementation in the `waiter-http` library, is blocking. Once a request comes in, we handle it fully to return a response. Since the internals of PHP and most extensions are not asynchronous, we still have a blocking issue. Do not get me wrong, the code can still

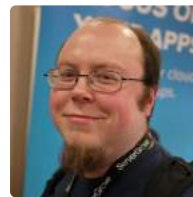
handle many concurrent connections quickly, but this has by no means solved any sort of async problems.

### Understanding is Key

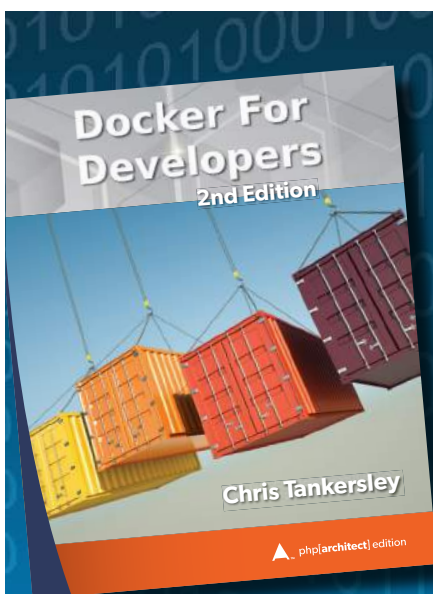
In the end, I hope this short series of articles has helped show off a bit more of what happens during a web request and why PHP works the way it does. I personally think that while the idea of an integrated server sounds interesting, PHP does not necessarily need it. Standalone HTTP servers handling requests is still a valid option.

Much is made about the Unix philosophy of software design as it stresses creating specialized tools that perform one job, and perform that one job well. PHP itself has the job of being a web-first language, and the amount of performance we have gained since the 5.6 days is amazing. PHP handles some of the largest websites in the world and has been getting faster with each release.

Will we have a built-in server one day? Maybe. And if that day comes, hopefully, how it all works will make a bit more sense, and maybe the implementation we built today might help shed some light on new architectural challenges as well as features we have in PHP.



*Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of *Docker for Developers* and works with companies and developers for integrating containers into their workflows. [@dragonmantank](https://twitter.com/dragonmantank)*



*Docker For Developers* is designed for developers looking at Docker as a replacement for **development environments** like virtualization, or devops people who want to see how to take an existing application and integrate Docker into their workflow.

**This revised and expanded edition includes:**

- Creating custom images
- Working with Docker Compose and Docker Machine
- Managing logs
- 12-factor applications

**Order Your Copy**

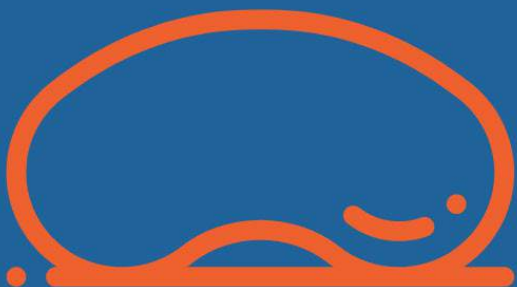
<https://phpa.me/docker-devs>



# Call for Speakers



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



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



# World Cup Draws

Oscar Merida

A random draw is straightforward, but we usually have business rules to implement and consider in any solution. With the World Cup kicking off this month, we get a chance to look at one scenario. Can you draw the teams into proper groups before the tournament starts?

## Recap

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

*default to the List of ISO 3166<sup>1</sup>, but be aware of other considerations if your application needs a simple “List of Countries”.*

The Wikipedia link provided gives us the qualified teams and the confederation to which they belong. I searched briefly to see if there was a JSON or CSV file with the team names, with no success. But we can make that ourselves. A first attempt might look like:

```
AFC, Qatar (hosts)
AFC, Saudi Arabia
AFC, South Korea
CAF, Cameroon
CAF, Ghana
CAF, Morocco
```

## The Research Step

We need to gather some facts before we can proceed. We need to list the confederation for each of the teams that qualified.

*What's a country?*

*This puzzle illustrates how some assumptions we make about the world around us aren't as clear-cut as we'd like. In this case—what is a country? You'll get a different answer depending on whom you consult, the United Nations, the Olympics, or FIFA, in our case. For example, England, Wales, and Scotland compete as separate countries. In the Olympics, they're all part of the Great Britain and Northern Ireland Olympic Team. Both US territories of Guam and Puerto Rico have a national team, as does Martinique, which is an overseas department of France. You might*

If you've researched how the World Cup draw works, you'll notice I've simplified it for this exercise. In reality, the host plus the top 7 teams are seeded first into each group. Doing so ensures traditional powers like Germany, Brazil, and Argentina don't meet in the group stage of the tournament. If the host is not a perennial contender, it ensures their group won't include one of those tougher teams. The remaining 24 teams are put in 3 pots of 8 based on their rankings and are drawn from their pots to help keep them balanced. Inevitably, the groups aren't perfectly balanced, and there's a “group of death”, and the hosts have a surprisingly easy draw. Extra credit if you take this solution further to account for this process.

<sup>1</sup> ISO 3166:  
<https://phpa.me/wikipedia-country-codes>

## Reading the Team Data

A quick PHP solution could use arrays to hold our data. However, with PHP 8's constructor property promotion, we can make a class to hold team data.

```
class Team {
    public function __construct(
        public readonly string $region,
        public readonly string $name,
    ){}
}
```

Next, we need to read our countries. I like using SPL's file objects for working with the filesystem and reading CSV files<sup>2</sup>.

```
$all = [];
$file = __DIR__ . '/teams.csv';
$file = new SplFileObject($file);
while (!$file->eof()) {
    $row = $file->fgetcsv();
    $all[] = new Team($row[0], $row[1]);
}
```

In six lines, we have a collection of all the teams in the competition.

```
1. Array
2. (
3.     [0] => Team Object
4.     (
5.         [name] => AFC
6.         [region] => Australia
7.     )
8.
9.     [1] => Team Object
10.    (
11.        [name] => AFC
12.        [region] => Iran
13.    )
14.    ...
```

<sup>2</sup> Reading CSV files:  
<https://php.net/splfileobject.fgetcsv>



## Groups

We know we'll have eight groups of four teams. Reading over our requirements, we also know there are other constraints:

- Except for European teams (UEFA), no team can be in a group with another team from the same region.
- No group can have more than two European teams.

We can use a `Group` class shown in Listing 1 to hold information about each one and provide helper methods for adding a `Team` to it if possible.

Listing 1.

```
1. <?php
2.
3. class Group
4. {
5.     private array $teams = [];
6.
7.     public function __construct(
8.         public string $label,
9.         public int $max
10.    ){}
11.
12.    public function isFull(): bool {
13.        return count($this->teams) == $this->max;
14.    }
15.
16.    public function addTeam(Team $team): bool
17.    {
18.        // todo
19.    }
20. }
```

The workhorse method for my solution is `Group::addTeam()`. After initializing the eight groups, the rest of the solution shuffles our `$all` array holding all the teams, then goes through the array and assigns each team to a group.

*If we didn't have rules about adding each team to a group, we could use 'array\_chunk()' and we'd be done.*

To get a team, we grab the last one in the `$all` array, which holds the ones that haven't been assigned. Once a group is full, we move it to the `$done` array and grab the next one. If we can't place a team into a group, we put the team back at the beginning of `$all` to deal with later.

After a first pass, I found that the `while` loop needed to short-circuit to avoid getting stuck in an infinite loop. The basic structure is in Listing 2.

Our main assignment loop doesn't have to concern itself with the rules for adding a team to a group. All it needs is to know if that operation was successful. Let's look at the body of `addTeam()`. While working on the solution, I discovered we need a third rule:

Listing 2.

```
1. // create the groups
2. $groups = $done = [];
3. foreach(range('A','H') as $label) {
4.     $groups[] = new Group($label, 4);
5. }
6.
7. // randomize the teams
8. shuffle($all);
9.
10. // start with assigning teams to group 4
11. $target = array_shift($groups);
12. $x = 0;
13. while ($all && $x < 60) {
14.     // grab a team to assign from the top of the stack
15.     $candidate = array_pop($all);
16.     if ($target->addTeam($candidate)) {
17.         if ($target->isFull()) {
18.             // this group is done, move to the next
19.             $done[] = $target;
20.             // and grab the next group
21.             $target = array_shift($groups);
22.         }
23.     } else {
24.         // Add to the bottom of our stack to try again later
25.         array_unshift($all, $candidate);
26.     }
27.
28.     $x++;
29.     if ($x == 60) {
30.         // avoid infinite loops
31.         foreach ($all as $team) {
32.             $target->forceAddTeam($team);
33.         }
34.         $done[] = $target;
35.     }
36. }
```

1. Each team must have at least one team from Europe.

There's an edge case with our approach in that we could conceivably—depending on how `shuffle()` randomizes our array—be left with more than three Europeans to add to the final group. I like to return as soon as I know I can't add a team. If a team makes it through the gauntlet of `if` conditions, we can add it to the group.

And then, I discovered we need a fourth rule to ensure we generate a solution on every run.

1. Teams from CONMEBOL and CONCACAF should not be in the same group.

Even with these rules, I still had runs where teams were not added to the final group because of the rules. In real life, sometimes we must redefine or relax our requirements. In the end, I assign any remaining teams to the last group. Listing 3 on the following page shows the complete code for this method.



Listing 3.

```
1. <?php
2. public function addTeam(Team $team): bool
3. {
4.     if ($this->isFull()) {
5.         return false;
6.     }
7.
8.     if ($team->region == 'UEFA') {
9.         if ($this->getCountForRegion('UEFA') == 2) {
10.            // can't place this team here
11.            return false;
12.        }
13.        $this->teams[] = $team;
14.        return true;
15.    }
16.
17.    if (count($this->teams) == 3
18.        && $this->getCountForRegion('UEFA') == 0) {
19.        // need to save last spot for a European team
20.        return false;
21.    }
22.
23.    if ($team->region == 'CONCACAF'
24.        && $this->getCountForRegion('CONMEBOL') > 0) {
25.        return false;
26.    }
27.
28.    if ($team->region == 'CONMEBOL'
29.        && $this->getCountForRegion('CONCACAF') > 0) {
30.        return false;
31.    }
32.
33.    if ($this->getCountForRegion($team->region) > 0) {
34.        return false;
35.    }
36.
37.    $this->teams[] = $team;
38.    return true;
39. }
```

Listing 4.

```
1. <html>
2. <style>
3.     .row { display: flex; flex-wrap: wrap}
4.     .card {
5.         border: 1px solid #888;
6.         padding: 1rem;
7.         min-width: 220px;
8.         margin: 0.5rem;
9.         border-radius: 0.5rem;
10.    }
11.    span { color: #999}
12.    ul {padding: 0 0 0 1rem}
13. </style>
14. <body>
15. <h1>Draws</h1>
16. <div class="row">
17.     <?php
18.     foreach ($done as $group) {
19.         ?>
20.         <div class="card">
21.             <h3><?=$group->label ?></h3>
22.             <ul>
23.                 <?php foreach ($group->getTeams() as $team) { ?>
24.                     <li>
25.                         <?=$team->name ?>
26.                         <span><?=$team->region ?></span>
27.                     </li>
28.                 <?php ?>
29.             </ul>
30.         </div>
31.     <?php
32.     }
33.     ?>
34. </div>
35. </body></html>
```

## Rendering the Output

I usually write my output to the console, but let's go the extra mile. The output here is suited to bulleted lists in cards. HTML with flexbox allows us to do that without much hassle.

The serviceable code in Listing 4 gives us something like Figure 1.

## Changing Requirements

Beyond trivial scenarios, you'll frequently need to negotiate how the solutions work in the end. In this case, the problem statement did not consider many of the edge cases we encountered. Because it used a simplified approach, it was not informed by the experience of past draws and the layers of requirements built up from each one.

Figure 1.



Sometimes, in simplifying a problem, we throw out the wisdom of experience.



## Sticker Swapping

Related to this tournament, fans around the world collect stickers with players, crests, and team photos for each team. 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.

### 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.



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)

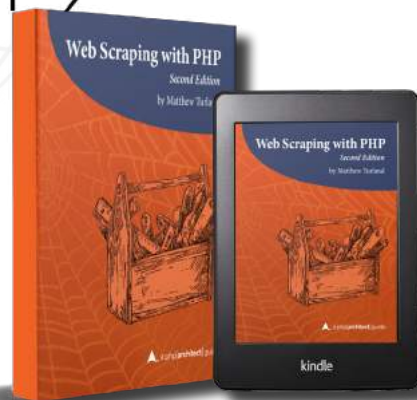
3 PsySH: <https://psysh.org>

## What if there's no API?

Web scraping is a time-honored technique for collecting the information you need from a web page. In this book, you'll learn the various tools and libraries available in PHP to retrieve, parse, and extract data from HTML.

**Order Your Copy**

<https://phpa.me/web-scraping-2ed>



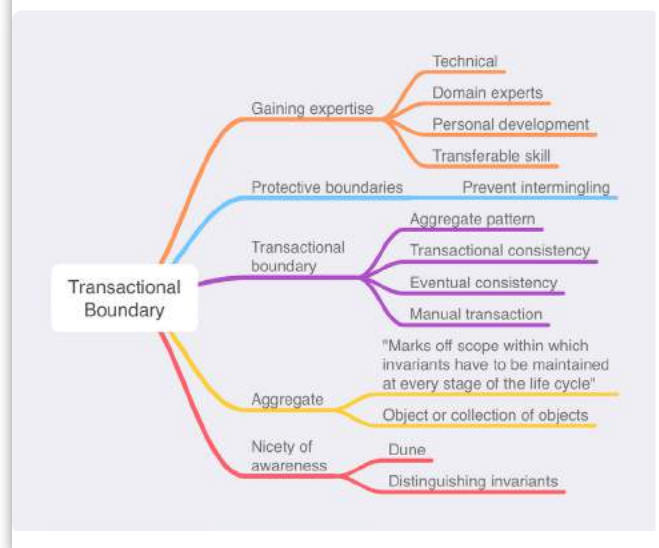


# Transactional-Boundary

Edward Barnard

Domain-Driven Design's Aggregate pattern is perhaps the most powerful of DDD's tactical patterns. However, strangely enough, the pattern's power doesn't come from the Aggregate itself. The power comes from the Aggregate's underlying concept—the transactional boundary.

Figure 1.



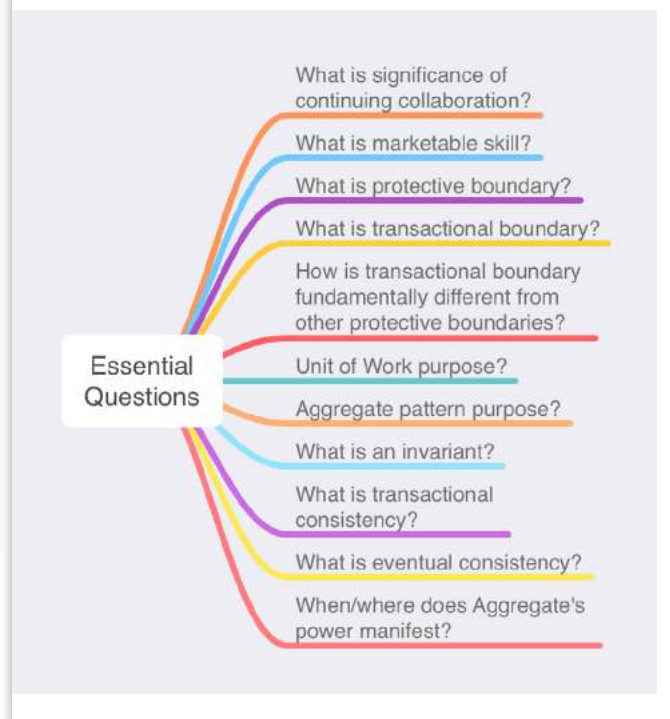
Code is available on GitHub at [ewbarnard/strategic-ddd](https://github.com/ewbarnard/strategic-ddd)<sup>1</sup>.

## Essential Questions

Upon surviving this article, you should be able to answer the following (see Figure 2):

- What is the significance of continuing collaboration between developers and subject-matter experts?
- What is the marketable skill we gain as developers?
- What is a protective boundary?
- What is a transactional boundary?
- How is a transactional boundary fundamentally different from other protective boundaries in our software?
- What is the Unit of Work pattern's purpose?
- What is the Aggregate pattern's purpose?
- What is an invariant?
- What is transactional consistency?
- What is eventual consistency?
- When and where does the Aggregate pattern's power manifest?

Figure 2.



## The Right Expertise

Strategic Domain-Driven Design is about communication and collaboration. That is, it's about "knowledge crunching," sharing information between business experts and technology teams. Eric Evans, in *Domain-Driven Design: Tackling Complexity in the Heart of Software* [Evans], warns (page 14):

*Good programmers will naturally start to abstract and develop a model that can do more work. But when this happens only in a technical setting, without collaboration with domain experts, the concepts are naive. That shallowness of knowledge produces software that does a basic job but lacks the deep connection to the domain expert's way of thinking.*

Are we responsible for building domain expertise into the software itself? Yes! That is, in fact, the whole point of

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



Domain-Driven Design. [Evans] presents our ideal process (page 15):

*The interaction between team members changes as all members crunch the model together. The constant refinement of the domain model forces the developers to learn the important principles of the business they are assisting, rather than to produce functions mechanically. The domain experts often refine their own understanding by being forced to distill what they know to essentials, and they come to understand the conceptual rigor that software projects require.*

Now consider that impact on ourselves as developers (ibid.):

*All this makes the team members more competent knowledge crunchers. They winnow out the extraneous. They recast the model into an ever more useful form. Because analysts and programmers are feeding into it, it is cleanly organized and abstracted, so it can provide leverage for the implementation. Because the domain experts are feeding into it, the model reflects deep knowledge of the business. The abstractions are true business principles.*

In other words, software development is about more than writing code. I consider this an extremely unfortunate fact because I'd prefer to just write the code. I'm not a people person. But software development translates what people want into instructions a machine can execute. Software development is most definitely a "people" sport.

I'm pointing out the obvious so we can look at Evans' observation in this light. We are each gaining skill in recasting "the model into an ever more useful form." Writing lines of code is a minor detail. The greater skill is, as Evans describes. We are each getting better at designing software by designing better software.

This is a transferable skill. As we each gain experience in creating ever-better software based on increasingly deep knowledge of the business, we learn how to take this skill to the next project (or task within the project), and the next after that.

All of this comes down to a way of thinking. For many of us, it's a new and different way of thinking. Rather than gaining greater skill with techniques, tools, and design patterns, we're aiming for greater skill in the business itself.

Do you see the problem here? Developing greater skill in any particular business sector isn't transferable—it locks us into that business sector. It's a "dead end" situation. That's why we're carefully extracting the abstract career-building principle here.

Martin Fowler, in *Refactoring: Improving the Design of Existing Code*, 2nd Edition, describes "Extract Superclass" (page 376):

*If I see two classes doing similar things, I can take advantage of the basic mechanism of inheritance to pull their similarities together into a superclass.*

Fowler then provides specific steps for identifying the points in common.

That's what we're doing here. The skill is not knowing a lot about a specific business or industry. That's the child class. The skill is in designing and improving software based on collaboration in *any* industry. That's the superclass.

## Protective Boundaries

What we've just described is the key skill of Domain-Driven Design. Exercising that skill leads to the next problem, and that's why we are here. How do we *implement* that understanding in code? In particular, how do we implement that understanding within our modern PHP ecosystem?

Scott Millett with Nick Tune, in *Patterns, Principles, and Practices of Domain-Driven Design* [Millett] explains (page 59):

*It is important to understand that there is no best practice when it comes to selecting a pattern to represent your domain logic. As long as you isolate domain logic from technical concerns you can implement Model-Driven Design and hence Domain-Driven Design.*

It's not the implementation that matters. It's the protective boundary that matters. [Millett] warns (page 74):

*No matter how many models you have you will find that they will need to interact to fulfill the behaviors of a system. It is when models are combined by teams without a clear understanding of what concept they apply to that they are prone to become blurred and lose explicitness, as concepts and logic are intermingled.*

*Therefore it is vital to protect the integrity of each model and clearly define the boundaries of their responsibility in code.*

We're about to look at another protective boundary—but this one is different. Up to this point, every protective boundary has aimed to *prevent* blurring, intermingling, and unfortunate interactions.

This new boundary is protective, but more in the sense of The Three Musketeers<sup>2</sup>, "One for all and all for one" (as shown in Figure 3 on the following page). This new boundary is the database transaction boundary.

## The Crucial Concept

The transaction boundary first leads us to a crucial concept. In fact, we have been steadily sneaking up on this concept that's crucial to Domain-Driven Design (DDD). It's the Aggregate pattern. It's arguably the most important and most powerful tactical pattern in DDD. We have already used

<sup>2</sup> *The Three Musketeers*:

[https://phpa.me/wiki/three\\_musketeers](https://phpa.me/wiki/three_musketeers)



Figure 3.



many of its underlying concepts. What do the experts have to say?

Carlos Buenosvinos, Christian Soronellas, and Keyvan Akbary in *Domain-Driven Design in PHP: A Highly Practical Guide* introduce the Aggregate (page 201):

*Aggregates are probably the most difficult building blocks of Domain-Driven Design. They're hard to understand, and they're even harder to properly design.*

I have generally seen the Aggregate pattern implemented as a Unit of Work. Martin Fowler, in *Patterns of Enterprise Application Architecture*, catalogs the pattern (page 184):

*Unit of Work maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.*

*When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.*

*A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.*

Many PHP frameworks implement Unit of Work. CakePHP 4.x, for example, provides sample code at Saving With Associations<sup>3</sup>.

However, Unit of Work must *not* be confused with the Aggregate pattern. The Aggregate pattern's power comes from careful thought about your design. Vaughn Vernon, in *Implementing Domain-Driven Design* [Vernon], warns (page 348):

*There are various ways to model Aggregates incorrectly. We could fall into the trap of designing for compositional convenience and make them too large. At the other end of the spectrum we could strip all Aggregates bare and as a result fail to protect true invariants. As we'll see, it's imperative that we avoid both extremes and instead pay attention to the business rules.*

## Transactional Versus Eventual Consistency

Fortunately, Vernon does provide useful insight for us to follow. We'll be following this advice (Ibid., pages 353-355):

*Rule: Model true invariants in consistency boundaries.*

*When trying to discover the Aggregates in a Bounded Context, we must understand the model's true invariants. An invariant is a business rule that must always be consistent. There are different kinds of consistency. One is **transactional consistency**, which is considered immediate and atomic. There is also **eventual consistency**. When discussing invariants, we are referring to **transactional consistency**.*

*Therefore, Aggregates are chiefly about consistency boundaries and not driven by a desire to design object graphs.*

To make clear Vernon's distinction, let's suppose an e-commerce site awards "rewards points". The business rule is that once you reach 100 reward points, you get a 5% discount for online purchases. Here's the question: How soon must the discount take effect?

- If the discount must take effect with the current purchase that's in the process of reaching 100 reward points, we need transactional consistency.
- If the discount should take effect the next business day, we only require eventual consistency.

## Manual Transaction

If we are processing an invoice containing line items, our PHP framework is probably smart enough to perform all

<sup>3</sup> Saving With Associations: <https://phpa.me/saving-assoc>





necessary database updates within a single database transaction. That's the Unit of Work pattern.

However, our Domain Events feature forced us to create our own manual transactions. That's because the Application Event is not obviously related to whatever else is being updated simultaneously. In the previous "Random and Rare Failures" article, section "Overkill", we took on the business rule that the Application Event gets published *if, and only if*, the database update is successful.

We did, in fact, sneak up on the Aggregate pattern! We identified an invariant, a business rule that must always be consistent. This business rule (i.e., that Application Events can *only* be published on successful updates) required transactional consistency. We implemented the transactional consistency with `$connection->transactional()` inside a try / catch.

## What is an Aggregate?

Domain-Driven Design treats an Aggregate as a "thing", an object (or collection of objects). Given that perspective, the Unit of Work implementation makes sense. But thinking of Aggregates as "things", collections of objects, misses the Aggregate pattern's power.

The pattern's power manifests when *designing* the Aggregate. It's not about tables or objects related to each other, or a Unit of Work that needs to be persisted within the same transaction.

[Evans] exposes the true power inherent in the pattern (page 135):

*AGGREGATES mark off the scope within which invariants have to be maintained at every stage of the life cycle.*

I'm reminded of a conversation in the book *Dune* by Frank Herbert. The Duke, Paul's father, is speaking with our hero, Paul (pp. 45-46):

*"Gurney tells me you did well in weapons today," the Duke said.*

*"That isn't what he told me."*

*The Duke laughed aloud. "I figured Gurney to be sparse with his praise. He says you have a nicety of awareness—in his own words—of the difference between a blade's edge and its tip."*

*"Gurney says there's no artistry in killing with the tip, that it should be done with the edge."*

*"Gurney's a romantic," the Duke growled. This talk of killing suddenly disturbed him, coming from his son. "I'd sooner you never had to kill. . . but if the need arises, you do it however you can—tip or edge."*

What's important, then, is creating a nicety of awareness distinguishing invariants. That's how we know where to draw the transactional boundaries.

## Essential Questions Answered

See Figure 4.

- *What is the significance of continuing collaboration between developers and subject-matter experts?* The model evolves into ever-more-useful form.
- *What is the marketable skill we gain as developers?* Designing and improving software based on collaboration in any industry.
- *What is a protective boundary?* A separation designed to prevent blurring and intermingling of separate models.
- *What is a transactional boundary?* A separation designed to provide transactional (atomic) consistency to everything within the boundary.
- *How is a transactional boundary fundamentally different from other protective boundaries in our software?* The other protective boundaries protect from outside influence. The transactional boundary ensures continuous consistency between everything within the transactional boundary.
- *What is the Unit of Work pattern's purpose?* Coordinates updating the database based on what changed within a business transaction.
- *What is the Aggregate pattern's purpose?* Mark off the scope within which invariants have to be maintained at every stage of the life cycle.
- *What is an invariant?* A business rule that must always be consistent.
- *What is transactional consistency?* Everything within the transaction succeeds together or fails (rolls back) together. Consistency is immediate and atomic.
- *What is eventual consistency?* Consistency need not be immediate nor atomic, but does need to be consistent at some future point.
- *When and where does the Aggregate pattern's power manifest?* In discerning the true invariants and drawing the transaction boundaries.

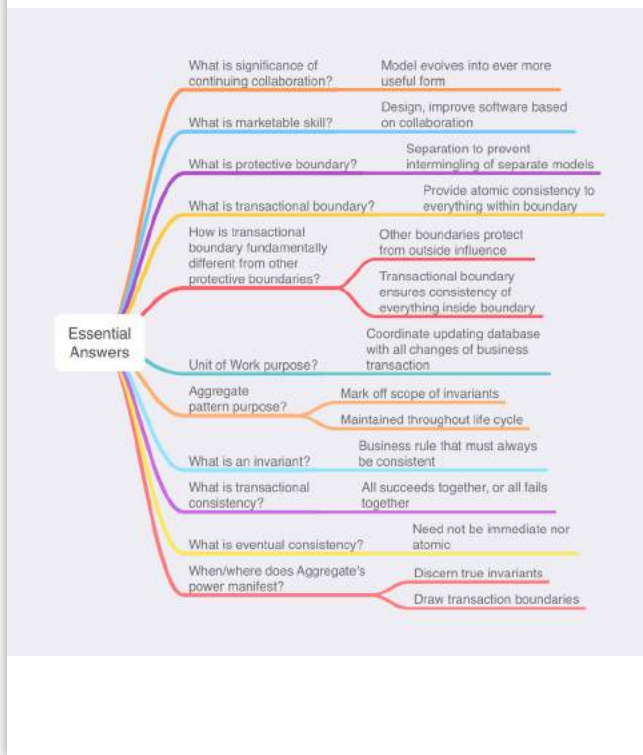
## Summary

A key concept of Strategic Domain-Driven Design is discerning invariants, that is, business rules that must remain consistent under all circumstances. I've seen countless cases where a web application updates several tables and simply assumes that everything will remain consistent. That's where the power of marking off transaction boundaries comes into play.





Figure 4.



We saw the Unit of Work pattern and its purpose to ensure all items that changed during a business transaction get correctly persisted in the database.

We did not specifically look at the Aggregate pattern. Instead, we looked at the Aggregate pattern's powerful purpose: to enforce those business invariants we have identified.




*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](#)

### Related Reading

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



# From Capone to Cray

WHERE COMPUTERS REALLY CAME FROM

by Edward Barnard

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

<https://leanpub.com/capone>



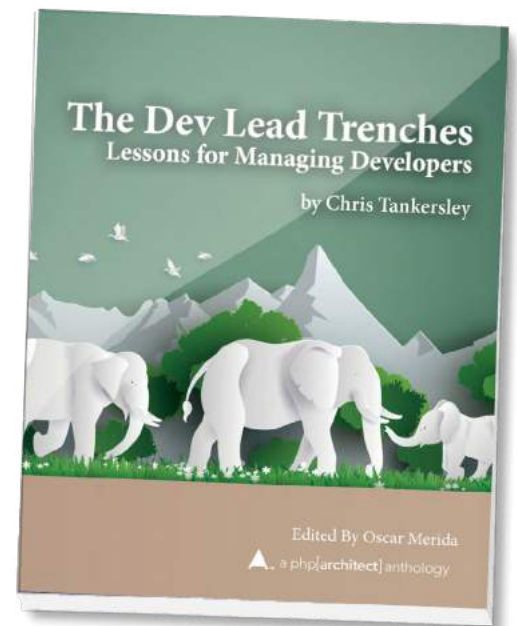
## You're the Team Lead—Now What?

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.

### Get the Most Out of Your Developers

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. But success is more than just slinging code when you're in charge, and this book covers project management and people skills you'll need to hone.

This book collects almost two-years worth of writings based on Chris Tankersley's experience leading development teams.



**Available in Print, PDF, EPUB, and Mobi.**

**Order Your Copy**

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



# Security in Drupal 9

Nicola Pignatelli

When you develop a Drupal site, you often think only of programming and rarely of security. Writing clean and secure code is useless if users can enter 123456 as a password. In this article, I'll explain which modules to use to secure your Drupal website. Of course, there are other techniques and tools to secure a web application, but you can't configure the web server or install third-party applications if you don't have access to the server. Drupal core developers improved code and core modules to minimize risks; they eliminated code that allowed the insertion of PHP code in fields of blocks, content type, etc. In this article, I'll explain which modules to use to secure your Drupal website. Of course, there are other techniques and tools to secure a web application, but you can't configure the web server or install third-party applications if you don't have access to the server. Okay, let's dive into discovering these essential modules for our safety.

## Login Security

Login Security module<sup>1</sup> permits the site administrator to add access control features to the login forms as:

- Limit the number of invalid login attempts before blocking accounts
- Deny access by IP address, temporarily or permanently.
- Email notifications to know about password and account guessing
- Email notifications to know about brute force login attempts

You can also disable Drupal core's login error messages, obfuscating the reason for the login failure.

## Security Review

Security Review module<sup>2</sup> carries out various checks for you that render your site insecure to secure.

Remember that you must make corrections manually.

## Website Security—Secure Login / Network Security

Website Security module<sup>3</sup> provides login security, registration security, brute force attack prevention, IP monitoring and blocklisting, DOS attack prevention, strong password enforcement, etc.

All of these provide you with enterprise-level security, protecting your Drupal site from hackers and malware.

## File Upload Secure Validator

File Upload Secure Validator<sup>4</sup> performs server-side validation for the extension of an uploaded file of any content type's file field. An attacker could change the file extension and upload a malicious content file.

This module uses the PHP library 'fileinfo', so Drupal checks the file's mime type and compares that to the allowed file extensions.

## Security Kit

Security Kit<sup>5</sup> provides Drupal with various security-hardening options, permitting it to mitigate the risks of exploitation of different web application vulnerabilities. Some of the security offerings are:

- Cross-site Scripting: an exploit where the attacker attaches code onto a legitimate website that will execute when the victim loads the website.
- Cross-site Request Forgery: an attack that forces an end user to execute unwanted actions on a web application where they're currently authenticated.
- Clickjacking: an attack that tricks a user into clicking a webpage element that is invisible or disguised as another element.
- SSL/TLS

## Security Pack

Security Pack<sup>6</sup> installs and configures a list of recommended security modules.

After installation, it sets up the following modules:

1 Login Security: [https://www.drupal.org/project/login\\_security](https://www.drupal.org/project/login_security)

2 Security Review module: <https://phpa.me/drupal-security>

3 Website Security module: <https://phpa.me/drupal-security-login>

4 Upload Secure Validator: <https://phpa.me/drupal-security-login>

5 Security Kit: <https://www.drupal.org/project/seekit>

6 Security Pack: [https://www.drupal.org/project/security\\_pack](https://www.drupal.org/project/security_pack)





- Password Policy
- Auto Logout
- Antibot
- Login Security
- Security Kit
- Username Enumeration Prevention

It is a good starting point if you don't know where to begin.

## Recaptcha

reCAPTCHA<sup>7</sup> Uses Google reCAPTCHA to improve the CAPTCHA system. It is tough on bots and easy on humans. There are many captcha modules, and this is just one of them.

You are free to choose the one you think best suits your needs, but having a captcha on your application is recommended to prevent malicious activity.

## Password Policy

Password Policy<sup>8</sup> permits enforcement of restrictions on user passwords by defining password policies. Configure the module, and you can be sure that your users won't be a security bug.

Remember that Password policies apply to passwords set via user web forms, not drush or others.

## Module Security Advisory Coverage Report

Module Security Advisory Coverage Report<sup>9</sup> shows which modules in use have Drupal security advisory coverage, and which don't. this module is useful if corporate Security Assessment Compliance requests a report.

## Spamspan Filter

SpamSpan filter<sup>10</sup> obfuscates email addresses to help prevent spambots from collecting them. Please note that this technique is unlikely to be foolproof. Unfortunately, the war between security and hackers is constantly evolving.

## Advanced Ban

Advanced ban<sup>11</sup> allows administrators to ban visits to their site from IP addresses like the core ban module and has additional features:

- IP range ban (IPv4 only)
- Blocked IPs expire (unblock by cron)

- Unblock all IPs
- Search IP
- Format ban text

## Remove Http Headers

Remove HTTP headers<sup>12</sup> removes configured HTTP headers from the response.

Removing HTTP headers lets you obfuscate that your website is running on Drupal. Please note that this technique is unlikely to be foolproof because if you use a standard path for files (sites/default/files), a hacker or bot can recognize that we are in the presence of a drupal site.

## Restrict Login or Role Access by Ip Address

Restrict by IP<sup>13</sup> permits you to restrict access for users or roles by IP. If you restrict user access, the first administrator user can't log in (user with id=1)

## Private Files Download Permission

Private files download permission<sup>14</sup> permits configuring download by role, directory permissions, and other features. This module is more important because it allows you to divide permissions and increase security on physical files.

## Drupal Perimeter Defence

Drupal Perimeter Defence<sup>15</sup> bans the IPs who send suspicious requests to the site.

Bots send requests without knowing if a website is Drupal, WordPress, Magento, or another. It sends all typologies of requests and waits for the answer to figure out which attack script to use.

So this module bans pages that are selected by patterns with regular expressions.

## Disable Login Page

Disable Login Page<sup>16</sup> is a module that prevents access to the default Drupal Login Page to anonymous users without the use of a secret key. This is useful for sites with no public user login requirements, like a corporate website or a personal blog.

The login page is protected with a secret key name-value pair which the admin can set. Without the secret key, you get an access denied error.

7 reCAPTCHA: <https://www.drupal.org/project/recaptcha>

8 Password Policy: <https://phpa.me/drupal-password>

9 Coverage Report: <https://www.drupal.org/project/msacr>

10 SpamSpan filter: <https://www.drupal.org/project/spamspan>

11 Advanced ban: <https://www.drupal.org/project/advban>

12 Remove HTTP headers: <https://phpa.me/drupal-remove-headers>

13 Restrict by IP: [https://www.drupal.org/project/restrict\\_by\\_ip](https://www.drupal.org/project/restrict_by_ip)

14 Private files: <https://phpa.me/drupal-privatefiles>

15 Perimeter Defence: <https://www.drupal.org/project/perimeter>

16 Disable Login Page: [https://www.drupal.org/project/disable\\_login](https://www.drupal.org/project/disable_login)





## Conclusion

Today I presented a different article than usual, but I believe that in our current era, we must also look at other aspects beyond pure programming.

Too often, we say, “Okay, I’ll check the security of the code, do the update checks”, and then words remain in the wind, to the satisfaction of the hackers and bad guys.

Take a moment to review your sites security today, use some of the recommended modules above to aid in your success.



*Nicola Pignatelli has been building PHP applications since 2001 for many largest organizations in the field of publishing, mechanics and industrial production, startups, banking, and teaching. Currently, he is a Senior PHP Developer and Drupal Architect. Yes, this photo is of him. [@pignatellicom](mailto:@pignatellicom)*

## Tackle Any Coding Challenge With Confidence

This book teaches the skills and mental processes these challenges target. You won’t just learn “how to learn,” you’ll learn how to think like a computer.

**Order Your Copy**  
<http://phpa.me/fizzbuzz-book>





# Our Responsibility in Learned Helplessness

Beth Tucker Long

In our quest to create applications and systems to get exactly what we want from users, are we training our users to be helpless?

One of the most important parts of our job as programmers is to protect servers from malicious, unexpected, and/or inadvertent actions. Think about building a form with a phone number field. Now think about the long, in-depth conversation you will need to have to ensure the phone number will be formatted correctly. Do you need only digits? Do you need 7 or 10 digits? Country code? Parenthesis and dashes? Do you need an extension? No matter how well you write up the instructions; the user will enter their phone number in the format they are most used to.

Enter the validation game. The vast majority of forms out there will check if the phone number is formatted correctly, and if not, will send the form back to the user to fix the data. By doing this, we keep telling the users, “You did it wrong.” The more this happens, the more they learn that they can’t do it right. Eventually, they will stop trying. This is where learned helplessness comes into play.

Let’s start with what the term “learned helplessness” means. According to Wikipedia:

“Learned helplessness is the behavior exhibited by a subject after enduring repeated aversive stimuli beyond their control.”

“In humans, learned helplessness is related to the concept of self-efficacy; the individual’s belief in their innate ability to achieve goals.” <https://phpa.me/learnedhelplessness>

In our case, the aversive stimuli, or negative consequences, for our users is that the form does not submit while giving them an error message and more work to do. The more errors they get, the more likely they will be to give up without even trying to fix the error. They begin to believe that it doesn’t matter if they try to fix the error because they will just keep getting more errors.

Is this really happening? Does this really affect how well our web forms can collect data? According to WPForms:

More than 67% of site visitors will abandon your form forever if they encounter any complications; only 20% will follow up with the company in some way. <https://phpa.me/wpforms>

That is a lot of people to lose if they hit an issue with a form. Just the simple act of asking for a phone number can drastically affect how much data you can collect as 37% of people will quit filling out a form if it asks for their phone number. And before you say that this is just because of privacy concerns, that number only drops to 20% if the phone number field is made optional.

We have exhausted our users into feeling unable to fill out online forms. Our need to get the perfect data at the lowest cost has trained our users not to give us any data at all. This is bad.

How can we fix this? First, we need to empower users. Having a sense of control over what is happening is one of the main ways to combat learned helplessness. Next, we need to do a better job of handling as much data formatting and manipulation as we can behind the scenes.

Do you want the phone number to be formatted as +X-XXX-XXX-XXXX? It’s not simple, but you can do it for the user. Start by filtering out any characters that are not digits. Count the digits starting from the end. Adding a dash after the first four digits, another after three more digits, and a third dash after the next three. Then validate whatever is left against a list of country codes and put a plus sign in front. No method is perfect, but it’s better to reformat the number and then ask the user if it looks correct or if they want to edit it. You are now giving them control over choosing to edit their phone number versus just telling them they did it wrong.

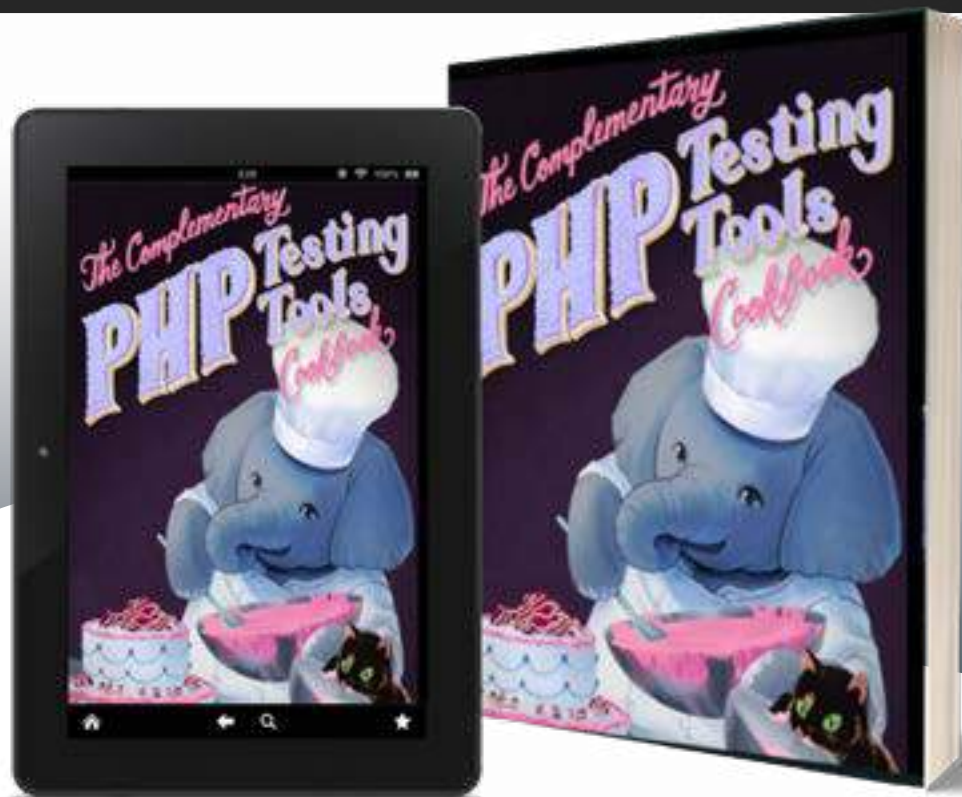
Obviously, no one solution will work for all data, and of course, it will sometimes be necessary to ask a user to go back and fix data on a form manually before submitting it. We can, however, invest more in our interactions with our users. We can be more proactive and creative about formatting and validating data. We can do a lot better than we have been. Let’s stop exhausting our users with tasks we can handle programmatically.

Related URLs:

- Wikipedia: Learned Helplessness - <https://phpa.me/learnedhelplessness>
- Psychology Today: Learned Helplessness - <https://phpa.me/psychtoday>
- WPForms: 101 Unbelievable Online Form Statistics & Facts for 2022 - <https://phpa.me/wpformsstats>
- Logit: Don’t Require a Phone Number in Your Contact Forms if You Want More Leads - <https://phpa.me/logitphonenumber>
- Unbounce: How To Optimize Contact Forms For Conversions - <https://phpa.me/unbounceconversion>
- filter\_var - [https://www.php.net/filter\\_var](https://www.php.net/filter_var)



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)



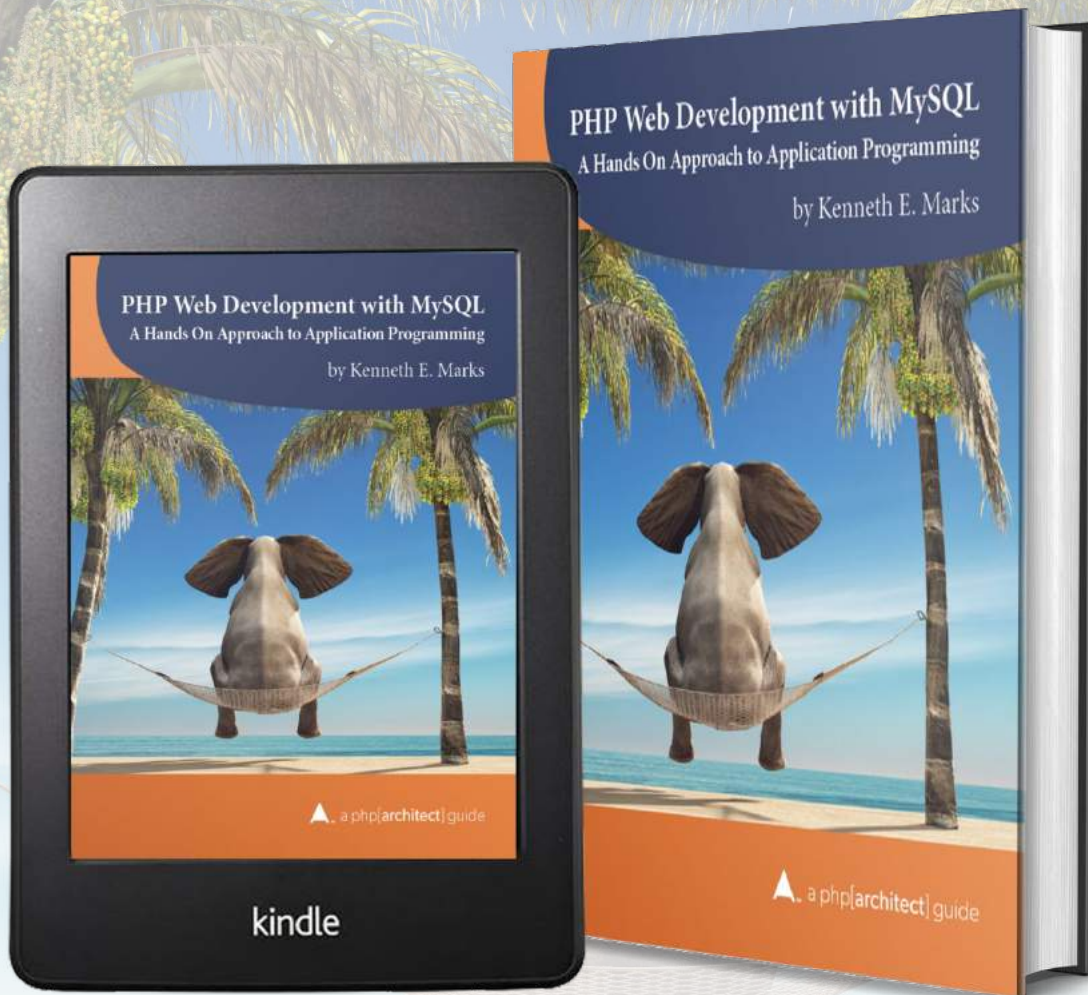
Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

*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>