

php[architect]

The Magazine For PHP Professionals

Evolving PHP

Value Objects

Where's The Internet Going?

ALSO INSIDE

The Workshop:
Databases as a Service

DDD Alley:
Create Observability

PHP Puzzles:
Maze Rats, Part 3

Security Corner:
Types of Tokens

Education
Station:
Dude, Where's My Server?

PSR Pickup:
PER: Coding Style

Artisan Way:
Defensive Programming

Barrier-Free Bytes
Welcome to a new column

finally{}:
Do We Deserve To Be Here?



Infobip CPaaS speaks your language

> Integrate feature-rich CPaaS solutions with your tech stack

Integrate any communication channel or module by using a flexible and programmable API stack that supports PHP. Tap into the open-source SDKs designed to get you up and running fast—with just a few lines of code.



Sign-up and get started with Infobip APIs.

[Try for free]

CONTENTS

JUNE 2023
Volume 22 - Issue 06



php[architect]

2 Ever Evolving Landscape

Eric Van Johnson

3 Value Objects

Łukasz Bacik

8 Where's the Internet Going?

Derek Pyatt

13 Dude, Where's My Server?

Education Station

Chris Tankersley

17 Types of Tokens

Security Corner

Eric Mann

19 Welcome to

Barrier-Free Bytes

Maxwell Ivey

21 Databases as a Service

The Workshop

Joe Ferguson

25 Maze Rats, Part Three

PHP Puzzles

Oscar Merida

29 Create Observability, Part 1

DDD Alley

Edward Barnard

32 PER: Coding Style

PSR Pickup

Frank Wallen

34 Defensive Programming

Artisan Way

Matt Lantz

37 Do We Deserve to Be Here?

finally{}

Beth Tucker Long

Edited without AI (yet)

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

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

Subscriptions

Print, digital, and corporate subscriptions are available. Visit <https://www.phparch.com/magazine> to subscribe or email contact@phparch.com for more information.

Advertising

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

Contact Information:

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

Copyright © 2023—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.

Ever Evolving Landscape

Eric Van Johnson

Over the past several months, I've reflected a lot on my experience as a PHP developer. Specifically, I've shared with some friends that I couldn't have predicted that this is where I would be when I was younger. So many aspects of my job didn't exist back then, including the Internet as we know it today. Additionally, the language I love to develop in, PHP, wasn't even a thought. It's amazing to think about how much PHP has evolved over the past several years as well.

At last month's php[tek], I heard a similar story echoed by a few people. It got me thinking and realizing a trait that I've noticed in many friends and colleagues in the PHP community. People who share my passion also share a thirst for knowledge. Whether it's diving deep into a new workflow or design pattern, or understanding how to leverage database optimization, we continually strive to augment our understanding and hone our skills. This never-ending quest for learning and unwavering curiosity fuels our growth and propels us forward in the ever-evolving landscape of PHP development.

php[architect] has played a pivotal role in my evolution as a PHP developer. While I still scour the internet for good articles on PHP, there seems to be little consistency, even from PHP-focused blogs. For over 20 years, this magazine has been a consistent resource for me. It has become my extended learning and a library of reference material.

There is always something new to learn in the world of technology. New services seem to emerge every week, from AI-assisted development to cloud services such as Databases as a Service (DBaaS). In Joe Ferguson's column, *The Workshop*, he writes about *Databases as a Service* which is an increasingly popular web technology.

In the DDD Alley column, Edward Barnard discusses the importance of timing in code performance with his article, *Create Observability, Part 1:*

Local Timing, which introduces a simple timing library based on PHP's `microtime()` function. Also in this issue, Matt Lantz presents six critical principles for defensive programming and strategies to combat common threats defined by OWASP, including Cross-Site Scripting and SQL Injection Attacks, Broken Authentication, and DDoS.

In his Security Corner, Eric Mann explains the different meanings of the term 'token' in software development and security, emphasizing the importance of precise terminology. He touches on CSRF, Access and Refresh, and Peripherals. Frank Wallen introduces us to PER: Coding Style, a new living document by PHP-FIG that provides clear guidelines for coding in PHP. It establishes a set of rules, surveys, and discussions to establish best coding practices.

In this month's edition of PHP Puzzles, Oscar Merida continues his exploration of complex code with *Maze Rats, Part 3*, sharing tips on how to navigate through it. The new column Barrier-Free Bytes focuses on web accessibility and introduces Maxwell Ivey, who stresses the importance of accessibility in his first column, *Introductions*. Finally, in his column *Dude, Where's My Server?*, Chris Tankersley discusses the evolving landscape of system operations and evaluates the role of the "server."

In my experience with PHP, despite the "Ever-Evolving Landscape," many people still dislike the language. These conversations often arise when deciding which language to use for a project. Beth Tucker Long addresses this in her column, "Do We Deserve To Be Here?" This month's feature articles include Łukasz Bacik's breakdown of Value Objects.

This issue includes a feature article by Derek Pyatt on the future of developers. He discusses how the Internet of Things (IoT) and Extended Reality (XR) are changing the way we interact with the world.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit <https://phpa.me/write> or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcements. Make sure you're connected with us.

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

Download the Code

Archive:

https://phpa.me/June2023_code

Value Objects

Lukasz Bacik

This article discusses the use of Value Objects in the context of PHP applications. The article presents step-by-step implementations of an example Value Object (class Person), as well as the use and discussion of the `lbacik\value-object`¹ library. Examples utilize the `phpspec`² framework as the test platform (one does not live by PHPUnit alone, after all!), and the entire code is available on GitHub³. Enjoy!

The definition of Value Object can be found in many publications, including, of course, Wikipedia⁴. I would like to begin with a quote from the book *Domain-Driven Design in PHP* (it's not a definition, but rather a pointer towards some – interesting to me – qualities).

One of the main goals of Value Objects is also the holy grail of Object-Oriented design: encapsulation. By following this pattern, you'll end up with a dedicated location to put all the validation, comparison logic, and behavior for a given concept.

Other publications you can refer to when looking for information on Value Objects:

- Domain-Driven Design, Eric Evans
- Implementing Domain-Driven Design, Vaughn Vernon
- Patterns of Enterprise Application Architecture, Martin Fowler

In the context of this article, out of the features of Value Objects mentioned in these publications, I will focus on the following:

- Immutability
- Replaceability
- Value Equality
- Side-Effect-Free Behavior

In addition, a key (in my opinion) feature of Value Objects is the entry

data *validation* mentioned in the opening quote. I mean here particularly the rule that a Value Object cannot be created for erroneous (incorrect) data, i.e. if an object exists, data is correct!

Execution

The form I decided to adopt is based on testing all the discussed functions in parallel to their presentation. For each function, I will first write an appropriate test and then present code that passes the test (the *red/green/refactor* mantra usually associated with the TDD approach⁵. `phpspec` itself is referred to as BDD⁶ (Behaviour-Driven Development) framework – though BDD originated from TDD, and all rules associated with TDD are fully respected.

An example Value Object is a `Person` class representing basic data describing *some person*:

- name
- age

As a reminder – since we are discussing Value Objects, we have no identity here (the `Person` object is not an entity, a record in a database, hence – among other things – it has no ID).

Setup

Let's create a new (empty) directory for our tests and then start configuring the project.

`phpspec` installation:

```
$ composer req --dev phpspec/phpspec
```

Let's define our test application's namespace in `composer.json` (here: `App`): (See Listing 1)

Listing 1.

```
1. {
2.     "require-dev": {
3.         "phpspec/phpspec": "^7.2"
4.     },
5.     "autoload": {
6.         "psr-4": {
7.             "App\\": "src/"
8.         }
9.     },
10. }
```

We will also configure `phpspec` – it is enough to add a `phpspec.yml` file with the following contents to the main directory:

```
suites:
  acme_suite:
    namespace: App
    psr4_prefix: App
```

The last thing to do is to update the autoloader:

```
$ composer dump-autoload
```

Test No. 1

With `phpspec`, using the *red/green/refactor* rule becomes practically second

1 `lbacik\value-object`: <https://phpa.me/packagist>

2 `phpspec`: <http://phpspec.net/en/stable/>

3 GitHub: <https://github.com/lbacik/value-object-spec>

4 Wikipedia: https://en.wikipedia.org/wiki/Value%5C_object

5 TDD approach: <https://phpa.me/wikitestdriven>

6 BDD: <https://phpa.me/wikibehaviordriven>

nature. Starting with the test – first, I instruct the framework that an App\Person class should exist in the app (our test Value Object):

```
$ vendor/bin/phpspec describe 'App\Person'
Specification for App\Person created in
    /.../spec/PersonSpec.php.
```

phpspec has created a test (a specification available in the spec directory – the tests directory is not being used here). The created test (specification) verifies the existence of the indicated class – let's run it:

```
$ vendor/bin/phpspec run
```

The call will fail – the required class (Person) does not exist yet (at this stage, there is even no src directory). We'll get a notification that the test failed, and a question if a missing class should be created. If we answer *Yes*, both a src directory and file with a Person class will be created.

Currently, the Person class will look like this:

```
namespace App;

class Person
{
}
```

Basic Checks

We have the most important components – specification, class file, and we know how to run tests (the phpspec run command). In addition, every current test (for now, only the verification if a Person class exists) runs correctly!

Let's disturb this *perfect balance*. Let's add the following code to the specification (spec/PersonSpec.php):

```
private const NAME = 'foo';
private const AGE = 30;

public function let(): void
{
    $this->beConstructedWith(self::NAME, self::AGE);
}
```

The let method is an equivalent of PHPUnit's setUp method (plus/minus) – i.e., the indication of an action performed before each test. We want our Value Object to represent a person and have two parameters – name and age. We want these parameters to be passed when creating instances of our Person object – hence the beConstructedWith call in our let method. Those familiar with PHPUnit may immediately see a difference in approach towards the test – here, *the specification* is also an instance of the tested class!

Let's verify our specification:

```
$ vendor/bin/phpspec run
```

The test will fail – there is no constructor in our class – we need to add one: (See Listing 2)

Listing 2.

```
1. class Person
2. {
3.     public function __construct(
4.         public string $name,
5.         public int $age,
6.     ) {
7.     }
8. }
```

Now the test will change to *green*.

We add a test to check if the values passed in the constructors have actually been assigned to the correct properties (the PersonSpec.php file):

```
public function it_contains_data(): void
{
    $this->name->shouldEqual(self::NAME);
    $this->age->shouldEqual(self::AGE);
}
```

phpspec gives us some intuitive possibilities, doesn't it? By the way – snake_case in the method's name is a phpspec convention. If we check the compliance of our code with PSR-12 using phpcs, we need to add *an exception* from the standard (in the configuration file) for classes in the spec directory. I put an example of phpcs configuration in the repository <https://github.com/lbacik/value-object-spec>⁷.

Let's run a verification. There shouldn't be any errors. The Person class takes on the values for the \$name and \$age parameters in the constructor and saves them in the corresponding fields. "So far, so good".

But! A Value Object should be an immutable object (*immutability* is the first feature mentioned in the introduction). The user must not change the values in an already created object. Let's illustrate it with the following check (in the PersonSpec class): (See Listing 3)

Listing 3.

```
1. public function its_props_are_ro(): void
2. {
3.     try {
4.         $this->name = 'bar';
5.     } catch (\Throwable $e) {
6.     }
7.
8.     $this->name->shouldEqual(self::NAME);
9. }
```

⁷ <https://github.com/lbacik/value-object-spec>: <https://github.com/lbacik/value-object-spec>

An attempt to assign a new value to the `$name` field (generally every field, but this test only concerns the `$name` field) should generate an exception (ignored in this particular test – it could probably be written... more neatly but I haven't found a way). In the final line of the test I check if the `$name` field value has actually remained unchanged (that is – in this case – if the value remains equal to the value passed to the `Person` class constructor in the specification's `let` method).

Let's run a verification (`phpspec run`).

We get an error message – our fields **are** editable – and the value in the `$name` field can be changed without a problem. How do we correct this? Since PHP 8.1, we have been able to set the parameters of the `Person` class constructor as `readonly`: (See Listing 4)

Listing 4.

```
1. class Person
2. {
3.     public function __construct(
4.         public readonly string $name,
5.         public readonly int $age,
6.     ) {
7.     }
8. }
```

And the test is passed :)

In the PHP versions before 8.1, we could only set the visibility of our fields to `private` or `protected` and create the appropriate *getters*.

Set

Value Objects are *immutable*. This does not mean, however, that we can't perform *assignment* operations – simply put, an attempt to assign a new value to a pre-existing object should end in a new object being created! This *feature* (called *Replaceability*) was described in *Domain-Driven Design in PHP* as follows:

This kind of behavior is similar to how basic types such as strings work in PHP. Consider the function `strtolower`. It returns a new string rather than modifying the original one. No reference is used; instead, a new value is returned.

Let's assume that the assignment is realized through the `set` method – let's write a corresponding test

```
public function it_has_set_method(): void
{
    $newPerson = $this->set(age: 40);
    $newPerson->age->shouldEqual(40);
    $this->age->shouldEqual(self::AGE);
}
```

Here I use *named arguments*, so PHP 8 is required. In the `set` method we can specify any number of arguments; in the example, we only specify the value for the `$age` field – in this case, in the newly created `$newPerson` object (the `set` method returns a new `Person` object), the value in the `$name` field remains unchanged from the `$this` object (the object with the `$PersonSpec` specification).

Verification (`phpspec run`) will return an error – a `set` method doesn't exist in our `Person` class. Do not create it automatically! We'll use its implementation defined in the `lbacik/value-object` library. Let's install the library:

```
$ composer req lbacik/value-object
```

Now we can update the `Person` class to make it inherit from the `ValueObject` class defined in the installed library as `\Sushi\ValueObject`: (See Listing 5)

Listing 5.

```
1. use Sushi\ValueObject;
2.
3. class Person extends ValueObject
4. {
5.     public function __construct(
6.         public readonly string $name,
7.         public readonly int $age,
8.     ) {
9.     }
10. }
```

After rerunning the tests, it turns out we have returned to the *green* state – now all tests should be passed!

In the `lbacik/value-object` it is assumed that we cannot create new fields with the `set` method (in new instances of a given class created with the method) – such attempts will generate a `ValueObjectException`. Let's check it! Here's a test:

```
public function it_has_the_same_keys(): void
{
    $this
        ->shouldThrow(ValueObjectException::class)
        ->during('set', ['otherName' => 'foo']);
}
```

This notation verifies (unfortunately, in this particular case, the `phpspec` notation isn't too... clear) if the execution:

```
$this->set(otherName: 'foo');
```

It will generate a `ValueObjectException`, which should, of course, happen (running tests should confirm that).

Isequal

Another *feature* listed in the introduction is *Value Equality* – two objects (of the `Value Object` type) are assumed to be equal if they *store* the same values, not when they are *identical*.

In other words, we are interested in comparing the values of these objects properties. We don't compare *keys* (which makes sense in the case of entities) or the addresses of these objects in the memory – we compare the value of each of their properties!

A test will show it better:

```
public function it_can_be_compared(): void
{
    $person1 = new Person('Thor', 25);
    $person2 = new Person(self::NAME, self::AGE);

    $this->isEqual($person1)->shouldEqual(false);
    $this->isEqual($person2)->shouldEqual(true);
}
```

The test should be passed immediately – the `isEqual` method was also defined in the `\Sushi\ValueObject` class.

A note: comparing within the `isEqual` method is realized by comparing the values of the same properties of two objects. Generating hash values and comparing only hashes is not utilized. Optimization by generating and comparing hashes is a topic for other library versions.

Side-effect-free Behavior

In a nutshell – after creating a Value Object, the values of its arguments cannot change. This assumption eliminates any worries about side effects⁸) (the properties of our Value Object should not be *in danger* of being changed). Of course, we can point towards *problematic* situations – let's consider, for example, a case where the value of one of the fields of a Value Object in an object not fulfilling the same criteria, i.e., not inheriting (in the context of this article) from the `ValueObject` class. In this case, the values of the object's fields will not have the same restrictions as the values of the fields of a Value Object. For example, in the case of the definition below: (See Listing 6)

Listing 6.

```
1. class Gender
2. {
3.     public const Female = 'female';
4.     public const Male = 'male';
5.
6.     public function __construct(
7.         public string $value,
8.     ) {
9.     }
10. }
11.
12. class Person extends ValueObject
13. {
14.     public function __construct(
15.         public readonly string $givenName,
16.         public readonly string $familyName,
17.         public readonly Gender $gender,
18.     ) {
19.     }
20. }
```

The `Gender` class does not inherit from the `ValueObject` class – when comparing the `Person` objects, the reference to the `Gender` class will be included, not its value!

Conclusion: ideally, the only objects saved in the `ValueObject` fields are other `Value Objects`!

If the `Gender` class in the above example inherits from the `ValueObject` class, the `value` parameter would be included when comparing `Person`-type objects!

I added tests illustrating these differences to the GitHub repository I mentioned in the introduction (it also contains the full code discussed in this article). The `PersonWithGenderSpec` (the `Gender` class, as above, does not inherit from `ValueObject`) and `PersonWithCitySpec` (where the `City` class inherits from the `ValueObject` class) tests.

Invariants

Now we have *validation* left. Type alone is absolutely not enough! Sometimes (often!), it will be necessary to use more advanced methods of validating data transferred when creating an object (remember the rule that an incorrect object should not be created in the first place).

In the case of the discussed `lbacik/value-object` library, the term *invariant* has been adopted to denote the rule that should be satisfied by the input data for an object to be created.

Every `ValueObject`-type class can contain any number of *invariants*.

Let's analyze the test below:

```
public function it_is_an_adult_person(): void
{
    $this
        ->shouldThrow(\Throwable::class)
        ->during('set', ['age' => 12]);
}
```

We are testing the following call:

```
$person->set(age: 12);
```

And expect the call to generate an exception because we want to add an *invariant* to our `Person` Value Object, which ensures that all created objects represent people over 18 years of age (people exactly 18 years old will also fulfill the criterion).

Let's look at the implementation: (See Listing 7)

Observe the constructor – it has to contain the base class constructor call – this ensures checking the *invariants*!

Invariants themselves are methods with an assigned `#[Invariant]` tag (attribute). If such a method throws (any) exception, it means that the tested criterion was not fulfilled when creating an object.

Python's `simple-value-object`⁹ library was the inspiration here. Of course, attributes (used here) work differently from Python's decorators (used in `sample-value-object`). Still, the

⁸ <https://phpa.me/wikisideeffect>

⁹ <https://pypi.org/project/simple-value-object/>

Listing 7.

```

1. use Sushi\ValueObject;
2. use Sushi\ValueObject\Exceptions\InvariantException;
3. use Sushi\ValueObject\Invariant;
4.
5. class Person extends ValueObject
6. {
7.     private const MIN_AGE_IN_YEARS = 18;
8.
9.     public function __construct(
10.         public readonly string $name,
11.         public readonly int $age,
12.     ) {
13.         parent::__construct();
14.     }
15.
16.     #[Invariant]
17.     protected function onlyAdults(): void
18.     {
19.         if ($this->age < self::MIN_AGE_IN_YEARS) {
20.             throw InvariantException::violation(
21.                 'The age is below ' .
22.                 self::MIN_AGE_IN_YEARS
23.             );
24.         }
25.     }
26. }

```

effect is (practically) identical – we have a *tag* (`#[Invariant]`), which we can use to mark all methods which should be called when creating an object. We can create any number of invariants, freely defining their rules.

Let's take a look at another example:

```

public function its_name_is_not_too_short(): void
{
    $this
        ->shouldThrow(\Throwable::class)
        ->during('set', ['name' => 'a']);
}

```

Let's assume we don't want the `name` attribute to be shorter than three characters. Of course, we can define the *invariant* similarly to the previous one, but let us consider another *possibility*.

Let's add `phpunit` to our application:

```
$ composer req phpunit/phpunit
```

Important! We are adding the `phpunit` library without the `--dev` parameter of the `composer` command – so, we are adding it as a part of the application rather than an element of the development environment. This may appear odd though probably not to everybody – using assertions in an application code (not just in tests) appears to be quite popular (although I

haven't done any research on the topic). However, it is (probably) not popular enough for assertions to be delegated to some smaller library (to avoid adding the entire `PHPUnit` when we want to use the application for that one element).

Let's see how we can use assertions in the `Person` class: (See Listing 8)

Listing 8.

```

1. use function PHPUnit\Framework\assertGreaterThanOrEqual
2.
3. ...
4.
5.     private const NAME_MIN_LENGTH = 3;
6.
7. ...
8.
9.     #[Invariant]
10.    public function checkName(): void
11.    {
12.        assertGreaterThanOrEqual(
13.            self::NAME_MIN_LENGTH,
14.            mb_strlen($this->name)
15.        );
16.    }

```

Let's run tests – everything should be in perfect order!

Does using assertions give us any concrete advantage? I'll leave it to the reader's judgment.

Summary

Using Value Objects can significantly increase the readability of our code. Of course, this solution will not work every time, but it is worth considering in most cases.

The `lbacik/value-object` underwent a considerable metamorphosis in version 1.x – a large amount of code from the previous versions was simply deleted¹⁰. Not all problems have been (so far) solved in an... ideal way. The current version, though, is fully functional, and as they say: *done is better than perfect* – test away!



I have a passion for coding, which is probably not surprising. I began with C++ during my university studies but then shifted my focus to Networking and DevOps. However, my love for coding ultimately prevailed, and ten years ago, I made the decision to pursue it as a profession. I have never looked back since then, and I am grateful for that decision every day.

¹⁰ <https://phpa.me/github>

The Next Generation of Developers: Where is the Internet Going?

Derek Pyatt

The world is changing before our eyes, and how we interact with it is adapting too. The Internet of Things (IoT) and Extended Reality (XR) are two growing technologies that are ushering in a new era of global connectivity. The next generation of developers is driving this new era—the ones who are growing with technology in their hands and are ready to take it to the next level. With each passing year, technologies like the IoT and XR will become increasingly prevalent in our daily lives. Modern developers specializing in these fields are not only creating games and apps for entertainment; they're also working on training simulations for doctors or building apps to help people with disabilities interact with the Internet. This article will explain how emerging technologies in the world are evolving and how the next generation of developers is changing with them.

Introduction

Devices are all around us, constantly sending and receiving data. Technology is becoming ubiquitous in life; everyone is surrounded by the Internet no matter where they go. While it can be challenging to foresee which technologies our society will completely adopt, it is clear that some are already making headway toward changing our lives. Utilization of Virtual reality (VR) and Augmented reality (AR), commonly grouped together as Extended Reality (XR), for purposes other than entertainment as well as the Internet of Things (IoT), is ushering in a new era of global connectivity. The Internet of Things is a web of interconnected physical devices that can collect and exchange data with each other. IoT is everywhere we look today. These objects are changing how infrastructure is crafted, from smart doorbells and thermostats to smart cities. Additionally, tools to create experiences beyond gaming in VR and AR are more available than ever. Anyone can push the limits of what technology is capable of with just an idea, and as time passes, we will see how far those limits can be pushed.

The Internet has been around long before me and has grown rapidly in front of my eyes. Contrary to how many early developers started their

careers, I was taught to use the Internet since I was old enough to read. Having access to the Internet through a family computer was a steady introduction, but the first real device that sparked my creative interest in development was a smartphone. Having the entire world in my hand was exhilarating. With near-unlimited knowledge at my fingertips, I knew that whatever I did with my life must include the Internet. Now, as a college student, I find myself working on what I dreamed the future of the Internet would be like. What technologies are essential for the next generation of developers to learn, though? In this article, we will focus on two growing technologies: IoT and XR. These technologies appear to be becoming a bigger part of life with each new generation. Also, as part of this conversation, we will discuss how programs like Games, Interactive Media & Mobile Technology (GIMM)¹ at Boise State University prepare students to work with these technologies in the real world.

The Internet of Things

The Internet of Things is a network of devices connected to the Internet that can communicate with each other

and exchange data. You are probably already familiar with some types of IoT, whether you realize it or not; devices like smart assistants, smart lighting, smart security systems, and even smart cities are all part of the IoT. Thanks to the Internet of Things, we can collect and analyze massive amounts of data about our environment and access tools to

make well-informed decisions. By embracing the IoT's potential, individuals, organizations, and governments can stay ahead of the curve and spur innovation to develop sophisticated services and devices. These pieces of hardware are often called "smart devices" because of their web-enabled functionality, referring to a device's ability to make use of the Hypertext Transport Protocol (HTTP) to interact with other programs or information while using the Internet as a communication hub.

The best way to conceptualize the Internet of Things is to imagine a spider web where we are a spider in the middle and the web itself represents the Internet (Figure 1). Surrounding the perimeter of the web is our structural foundation of smart devices transmitting information through the web. As we move inward from the perimeter, the thread starts to cross over different strands

¹ Games, Interactive Media & Mobile Technology (GIMM): <https://www.boisestate.edu/gimm/about/>

Figure 1.



and strengthen the overall structure, allowing the devices to communicate with one another and transmit data. The data travels as vibrations through the web connecting the smart devices together, eventually making their way to us at the center. The ability to react and perform functions based on this exchange of data is what makes the foundations smart devices.

As advanced and groundbreaking as the idea of the IoT is, this technology is not brand new. The IoT only exists because of significant technological achievements made in the past. These achievements include but are not limited to, ever-smaller sensors, the increased capability of being connected to the Internet, and the increasing standardization of IoT devices. The first achievement to look at is the ever-shrinking size of parts like temperature sensors, pressure sensors, accelerometers, gyroscopes, and light sensors within smart devices. Smaller sensors use less energy, making them ideal for IoT devices that rely on battery power or other restricted energy sources. Their small size makes them ideal for incorporating into compact IoT devices

like wearables or smart home sensors. Another important aspect of smart devices is their ability to communicate via the Internet using Wi-Fi, Bluetooth, cellular connectivity, and 5G. Without Internet connectivity, IoT devices would be constrained to functioning in isolation or through localized networks, decreasing their efficacy and utility. Furthermore, Internet connectivity allows IoT devices to benefit from cloud-based services, machine learning, and other cutting-edge processes, vastly improving their capabilities and potential. Lastly, standardization of IoT is crucial to future-proofing the concept into ubiquity. Due to standardization's critical role in the successful development of IoT devices, organizations like the International Organization for Standardization (ISO)² and the International Electrotechnical Commission (IEC)³ have set up standards for "compatibility requirements and model for devices

within industrial IoT systems⁴. Standardization requirements like this will allow the IoT to become a manageable reality for the public.

The purpose of the IoT is to allow connected devices to interact with one another and share data in real time without requiring human intervention. Data collected by these devices can then be assessed to make more informed decisions and improve processes. For example, a smart home security system, like Ring Doorbell, can employ motion sensors to detect movement and send messages to a homeowner's smartphone if it detects unusual activity. The theory of IoT is that by linking physical devices through the Internet and allowing them to interact, we can build a more efficient, intelligent, and automated society.

The current applications that make up the IoT ecosystem include items such as the smart home, virtual devices, and more. One such application of IoT that most people are familiar with is "smart homes". Today, consumers can access any number of "smart appliances" they can control right from their phones. From smart fridges that tell you what you need from the store to thermostats that know when to turn on and regulate the home's temperature, the smart home is taking over many American households. The most interesting type of smart device that now lives alongside us might be the smart assistant. Whether they have Amazon Echo or a Google Smart Home, consumers can now have a virtual assistant on standby for their every need. With the ability to answer most requests using voice alone, smart assistants are far more accessible than most devices that require a screen. Furthermore, the more IoT-enabled devices one has, the more powerful the assistants become, allowing the user to control the other smart devices by voice.

Thinking even bigger than the home, we have the "smart city". A smart city is one in which a network of sensors collects electronic data from and about

² <https://www.iso.org/home.html>

³ International Electrotechnical Commission (IEC): <https://www.iec.ch/homepage>

⁴ compatibility requirements and model for devices within industrial IoT systems: <https://www.iso.org/standard/53282.html>

people and infrastructure to enhance efficiency and quality of life. Due to its creative application of technology and data to raise the quality of life for its residents, encourage sustainability, and foster economic development, cities are starting to incorporate more aspects of smart city technology and infrastructure. For example, Barcelona, Spain, has recently started to use a “digital twin” to assess its current and future city development⁵ and is working towards becoming a smart city. A digital twin is a virtual duplicate of a physical asset, in this case, a city, that can be utilized to run trial simulations and analyze their potential results. Barcelona is utilizing its digital twin for better municipal planning, optimizing services, and increasing public involvement. In order to do this, the city used open data to create an interactive map that monitors facilities and services that allows planners to identify which areas of town are underserved. Technology like this has the potential to change urban planning and make cities more sustainable and efficient.

In the coming years, the Internet of Things is expected to drastically change many facets of our life. According to reports by McKinsey Global Institute, IoT solutions have the potential to produce \$4-11 trillion in economic value by 2025. Following in Barcelona’s footsteps, cities around the globe are anticipated to improve by embracing the IoT. Based on reports from Allied Market Research, the worldwide smart cities market was worth \$648.36 billion in 2020 and is estimated to be worth \$6,061.00 billion by 2030. Such smart city improvements could include integrating IoT sensors and smart devices to enhance public services, transportation, and infrastructure - or maximizing supply chains and energy management to minimize the amount of waste people create.

Yet, as more devices and systems are linked to the Internet, the possibility of security risks will only increase. Robust security measures will be required to prevent the abuse of IoT operations to resolve these problems. The ability for devices and systems to interact and cooperate successfully will depend on future-proofing the IoT systems via standardization that can enhance the overall user experience, increasing everyone’s access to and benefit from them.

Extended Reality

Both Virtual Reality (VR) and Augmented Reality (AR) are included under the term Extended Reality (XR). XR is the use of technology to either build virtual worlds or incorporate digital aspects with the actual world. The key distinction between the two is that AR projects digital data into the real world, whereas VR offers a completely immersive digital experience.

Although AR and VR have been around for a while, technology has advanced dramatically in recent years. According to the Virtual Reality Society⁶, the first VR systems were created in the 1960s and 1970s. It wasn’t until the 1990s that game consoles like the Nintendo Virtual Boy made

VR more publicly available, though. You will find a concise history of AR on Wikipedia. In their piece, they claim while Tom Caudell and David Mizell initially introduced the term “augmented reality” in 1992 to refer to a system that utilized head-mounted displays to overlay digital information for manufacturing Boeing aircrafts⁷, Harvard Business Review states that the first augmented reality technology was built in 1968 at Harvard by computer scientist Ivan Sutherland. Thanks to tremendous advancements in hardware and software, these technologies are becoming more accessible than ever; newer VR systems can be played without fixed sensors, while Lidar improves AR tracking and occlusion. Consumers now have easier access to AR thanks to smartphones and tablets, while VR devices have dropped in price and continued to improve in performance. In addition, virtual engine worlds, like Unity and Unreal Engine, have made it simpler for developers to construct AR and VR apps.

The potential for XR extends beyond gaming as well. These technologies can change the workflow in many other fields completely. For example, VR can be utilized for immersive interactive training experiences for industries like healthcare, the military, and others where individuals need to perform physical movements that require precision. VR also has a role in visualizing the previously discussed topic of digital twins. AR can be used in architecture and design to help envision and test concepts in a real-world setting. Moreover, AR and VR have the potential to revolutionize the way we work and communicate by enabling distant teams to engage instantaneously through virtual meetings. As XR technologies advance and become more widely used and affordable, more industry sectors will be capable of benefiting from them.

Educating the Next Generation of Developers

With the estimated 27 billion IoT devices to be connected by 2025⁸ and massive XR projects like the Metaverse, who will be developing this technology, and how will they be trained? For traditional college programs like computer science, computer engineering, and electrical engineering that already have a large amount of established content knowledge to teach, what programs will have the time to spend extended periods of time creating projects on topics like IoT and XR? One program ready to help train the next generation of developers is Boise State University’s GIMM degree, which devotes extended attention to developing technology, ensuring that students gain experience with various emerging technologies, including IoT and XR.

While the name Games, Interactive Media, & Mobile Technology sounds like a primarily game design major, GIMM specializes in teaching students what is currently being used in the industry, emerging tools, and technologies, and preparing them for the job market. The GIMM curriculum provides students with a diversity of courses that covers the game development pipeline, covering both the artistic

⁵ <https://phpa.me/barcelona-planning>

⁶ <https://www.vrs.org.uk/virtual-reality/history.html>

⁷ https://en.wikipedia.org/wiki/Industrial_augmented_reality

⁸ <https://iot-analytics.com/number-connected-iot-devices/>

and coding aspects of the process. In addition, GIMM also instructs students in web development, database management, and physical computing, such as Arduinos.

Community and collaboration are important factors enabling the major's ability to sculpt effective developers. Through various activities like game jams and classroom all-nighters, students are given opportunities to collaborate and work on large-scale projects with each other. However, GIMM's biggest strength is its ability to independently prepare students for learning new technology. By educating students on how to teach themselves, GIMM produces strong, tenacious graduates who are driven to learn on their own.

Teaching IoT

Although not a course that appears to be taught in many other college programs, GIMM covers IoT as part of its 4-year curriculum. In the course designated for IoT, students were placed into small groups and asked to use Amazon Echo Frames to develop their own smart devices using an Alexa application. Furthermore, each application needed to incorporate either machine learning or Arduino functionality. After coming up with a scenario or functionality that would all be controlled by voice alone through the use of Echo Frames and learning about necessary IoT systems like cloud computing (e.g., Amazon Web Services), they completed their Alexa applications. After completing their applications, students created a demo reel for their project to display what they developed and explain its functionality. As part of this course, the project my group developed was a smart toilet⁹ that was designed to complete the following task by voice command: (1, 2) close and open the lid, (3) flush, (4) heat the seat, and (5) turn on a bidet. The purpose of teaching IoT is to prepare students for the multitude of jobs that will start appearing as demand increases for IoT positions. Allowing students to experiment and decide what they want to pursue is an important part of GIMM; getting them to the point where they feel confident enough about a topic to apply for a job in its field is a primary goal of the overall degree.

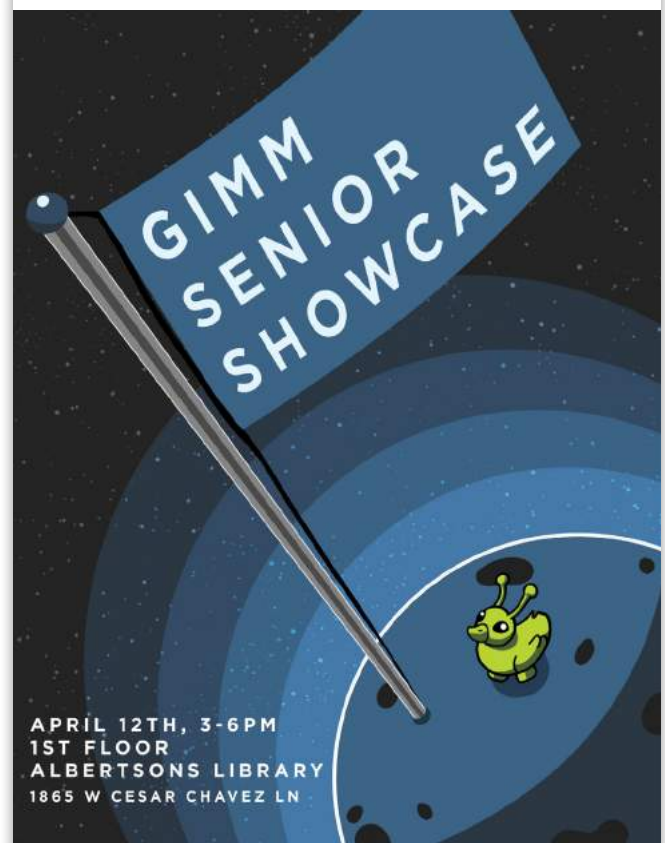
Teaching XR

A large portion of the GIMM major, myself included, felt that one of the biggest draws to the program was its emphasis on VR game development. Starting in the sophomore year, students develop a VR game idea that they will continuously work on throughout their major. They create game design documents and repeatedly pitch their ideas to faculty until a polished idea is developed. Work on this VR game continues throughout the program culminating in a capstone project. Students learn from each other and personally work on every aspect of their games, from UI and mechanics to art direction. After user experience testing and faculty feedback, students present their final VR game to the public via the yearly senior showcase (Figure 2). Teaching VR is important as the medium has only been growing and becoming more accessible for

games, simulation training, and remote work productivity. Giving students the ability to create their own environments greatly assists in solidifying their understanding of VR, which can be applied to many professional positions.

Augmented reality in GIMM is taught through Apple's RealityKit and is centered around creating accessibility applications for people with intellectual disabilities. Students start the course by learning foundational Swift and developing their abilities by creating a small AR game that can be played on an iPhone or iPad. Next, groups are given Swift topics (MapKit, CoreML, Object capture, SiriKit, etc.) to present to the class. These are designed to familiarize students with all potential functionality they can utilize in making their final accessibility app in groups. This year, students were tasked with creating an app to help someone intellectually challenged around the Boise State campus. Students created apps to assist with navigation, ordering food, and using printers in the library. Studying AR helps students establish a diverse toolbox they can pull from in various positions; even more,

Figure 2.



Flyer Credit:

Jasmine Reyes - jreyes60109@gmail.com

Brilynn Funderburg - brilynnfunderbu@boisestate.edu

9 smart toilet: <https://www.youtube.com/watch?v=QRFo5C3XNUU>

accessibility, compliance, and compassion are always in-demand skills in every field.

Conclusion

The technology that surrounds us is evolving every day. Paying attention to where the Internet is going will clue us into how future technology will affect our everyday lives. With each passing year, technologies like the Internet of Things and Extended Reality will become increasingly prevalent in our daily lives. The next generation of developers needs to be ready to inherit this evolving digital world and make changes for the better. Properly educating them and providing them with the tools they will need to create change is necessary to maintain the current state of technology and push beyond it in a positive direction. Programs like the Games, Interactive Media, and Mobile Technology program at Boise State University highlight the ability to teach the next generation of developers about emerging technologies via its curriculum. The next generation of developers will have a diverse

background of knowledge and will be the ones to lead the Internet into its next form.



Derek Pyatt is a third-year honor student at Boise State University. He is part of the Games, Interactive Media, and Mobile (GIMM) program while also working as a Web and Digital Coordinator for BSU's Admissions office. As a student in GIMM, he has had the opportunity to work on various mobile applications, websites, and games to increase his knowledge base and understanding of how applications are developed.
[@pyatt_derek](#)



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



Become a Pro or Business supporter to help ongoing development.

Supporters get help via email and elevated issue priority.

<https://xdebug.org/support>

support@xdebug.org



Dude, Where's My Server?

Chris Tankersley

It is amazing how fast technology changes, especially around programming. In the span of one human lifetime, we have gone from needing rooms of equipment to run basic mathematical computations to being able to click a button and not even knowing where our code is being run. Currently, as I write this article, I completed a demo where my setup was clicking a button that deployed the demo to render.com. At no point was I told where the code was going to run, and I did not care.

Computing, and programming, are a long list of solutions to the problem “I do not want to deal with this”. Since CPU, memory, and hard drive space have become less expensive over time, we can start to think of them as background choices. When you go to a restaurant, do you ask them what brand of cheese to put on your hamburger or which farm the lettuce came from for your salad? No, you know that you want cheese, and that salad better have lettuce that is not iceberg.

Deploying software has seen radical changes over the last twenty years. My first professional job in programming was deploying to a physical server in the “server room” of the company I worked for, and it was a glorified desktop. Shifting forward, not even a year, I am suddenly buying servers to run virtual machines. Developers moved away from heavy virtual machines to lighter-weight container technologies. We now live in a world where we do not even care about that.

Why and how did all this change, and what does that mean for us today?

From Bare Metal

Computers are nothing without software to run on them. While the first arguable “computer” was the Analytical Engine¹, as it was a general-purpose device that could be programmed to do something. It was designed by Charles Babbage² in 1837 but was never actually built. Ada Lovelace is credited with writing the first computer program³, which computed Bernoulli numbers⁴. This program, or more accurately, algorithm, would have run directly on the hardware of the Analytical Engine had Charles Babbage been able to finish it.

This started a long historical record of software being very tightly coupled to the devices that they are deployed to. As modern computers evolved, programmers still clung to this idea, as the devices were not generally powerful enough to deal with higher-level languages or were still very mechanical in design.



Daniel Stori {turnoff.us}

1 <https://phpa.me/analytical-engine>

2 https://en.wikipedia.org/wiki/Charles_Babbage

3 <https://phpa.me/ada-lovelace>

4 <https://phpa.me/bernoulli-number>



As higher-order languages evolved, like Assembly in 1949⁵, we still developed software for a machine. Even today, Assembly is tied directly to the CPU of the device on which it will run. Assembly code written for an x86 CPU, like a Pentium, will not work on an M1 processor from Apple. You even have instances where there are variants of the same family. There are many versions of the x86 instruction set.

Since resources were minimal, software developers worked within the constraints of the systems they knew they could deploy to. For decades, software was designed for single-threaded CPU operations and megabytes of RAM. It was not uncommon for there to be questions like “How many Apache httpd threads can I run before the CPU is overloaded?” or even “What happens if a user wants to upload a 100-mega-byte file?”

Commercial software publishes minimum system requirements—it still does to this day. A quick glance at Steam, the gaming store and platform, shows that Dwarf Fortress requires a 64-bit processor, a dual-core CPU running at least 2.4 gigahertz, and 4 gigabytes of RAM. While web developers back in the day never had to publish such stats, it was not uncommon to see instructions to tweak memory usage in the PHP ini file.

It was also not uncommon for more strenuous applications to need their own server. One application I had helped install and run for a former employer did have minimum system requirements, and we had to dedicate an entire server for it to run. It was a digital imaging system with a web front-end bundled as a large Java application. It had specific memory, hard drive speed, and CPU requirements that had to be met; otherwise, it would not even boot all the services properly.

To Virtual Machines

Ignoring the Analytical Engine, it did not take long before people noticed that machines were not running at full capacity one hundred percent of the time. Since many early computers, especially the ones that filled up entire rooms, were expensive, there was a desire to squeeze every last ounce of productivity from machines. This started with allowing many users to use the machines concurrently through systems like Time Sharing⁶.

It did not take long until software and hardware developers started to look at virtualization as a method for having one single machine act as multiple machines. By the mid-1960s, IBM had the IBM M44/44X⁷ simulating many 7044 machines simultaneously. This machine was a research system but helped show off what things like virtualization could achieve. This was termed partial virtualization as it did not virtualize the hardware.

By the late 1960s, IBM had released CP-67/CMS⁸, an operating system for the IBM System/360-67, and it is considered the first widely available virtual machine architecture. Its predecessor, CP-40, was a full virtualization operating system. This meant that everything was virtualized to the point that it could replace a bare metal machine, and, in theory, the software could be written for the virtualized environment instead of the bare metal.

All of this ignores emulators⁹, which go back to the early 1960s with the IBM System/360. Emulators were used for backward compatibility, whereas virtualization was used for better time-sharing or prototyping software. This was also distinct from Process Virtual Machines, which is closer to how languages like .NET and Java currently run. Process Virtual Machines arose in the late 1960s, with one of the earliest examples being the O-code Machine¹⁰.

Since it required a large amount of computing power to run virtual machines, the idea stayed very much in the “big iron” and mainframe space. CPU, memory, and disk space plummeted in price over the years, and by 1999 VMWare introduced VMWare Workstation¹¹, which could run fully virtualized machines on commodity hardware. By 2003 Xen¹² was released as an open-source hypervisor and has been a part of the Linux Kernel since 2011.

The key to this was that virtualization wedged a shim, called a hypervisor, between the operating system inside the virtual machine and the actual hardware. This hypervisor could control access to the hardware and limit the resources that the operating system had access to. Hypervisors could chunk up a system with multiple CPUs and gigabytes of RAM into smaller chunks allowing for better resource utilization and security. This created a bit of overhead as it was a full machine, which needed an operating system, and operating systems eat up resources.

On the web development front, shared hosting services started to offer “Virtualized Private Servers¹³,” more commonly called VPSs. Hosting companies who had used software like Plesk or OpenVZ to provide multi-tenant hosting on a single machine (many customers on a single server) could now offer customers entire machines to run their software. IT services could save money by buying fewer, larger machines and turning them into many machines.

This meant that the software, much more than the hardware, was a more pressing matter to developers. While someone could deploy to a very small server, it was easier to make sure that the bulk of a developer’s time was spent on software dependencies. CPU and memory were cheap; we now cared about what software version was installed. Virtualization lets

5 <https://phpa.me/lang-history>

6 <https://en.wikipedia.org/wiki/Time-sharing>

7 https://en.wikipedia.org/wiki/IBM_M44/44X

8 <https://en.wikipedia.org/wiki/CP/CMS>

9 https://en.wikipedia.org/wiki/Virtual_machine

10 <https://en.wikipedia.org/wiki/BCPL>

11 <https://phpa.me/vmware-workstation>

12 <https://en.wikipedia.org/wiki/Xen>

13 <https://phpa.me/vps>



us run Windows and Linux side-by-side on the same hardware or even many versions of Linux side-by-side.

As desktops and laptops became more powerful, we saw tools like Vagrant¹⁴ come into existence. Vagrant allowed developers to create many virtual machines on a device. For developers like me doing client work, you could now have a virtual machine per client, with their unique setups, all at your fingertips. Sure, I couldn't run two clients at once, but the fact I could have many unique environments that did not conflict was a major thing.

Short of Laravel Homestead¹⁵, virtualization has lost much of its charm. Virtualization for many years let us decouple our application from the hardware and instead focus more on the software and operating system dependencies. It still has its place as hosts like Digital Ocean¹⁶ and Amazon AWS¹⁷ rake in millions or billions of dollars in hosting virtual machines. Still, for day-to-day development, virtual machines take up less cognitive time for developers.

We still knew roughly where our software was running. We could point to an instance in AWS or Google Cloud and say, "My code is running on this virtual machine."

To Containers

As with everything, we moved on. By 1979 developers had noticed that tools like time sharing and multiple processes could cause problems. While virtualization could solve this problem, virtualization requires expensive hardware. What if we put walls up around a process to isolate it? chroot¹⁸ was introduced in 1979 as part of Version 7 Unix and in 1982 to 4.2BSD.

The idea of process isolation (which has nothing to do with Process Virtualization) continued to grow, especially in the service space. The most well-known was Solaris Zones, which took the idea of chroot and expanded upon it. By 2008, this idea gave life to LXC¹⁹, which allowed operating system-level virtualization through a shared host kernel. This allowed multiple Linux distributions to run as separate processes without the overhead of virtual machines.

Instead of using software like a hypervisor to control access to the hardware, Containers used built-in operating system tools to restrict what resources a process (or series of processes) had access to. This meant that you could run many versions of an operating system as long as they shared the same kernel as the host. In the case of Linux, this meant you could run Fedora and Ubuntu on the same machines inside LXC, as they both used the same standard Linux Kernel as the host.

How did this work? In the case of Linux, once the kernel starts, it begins to launch processes. The Linux kernel has subsystems that allow it to restrict access to what these processes can see, including other processes, hardware, and memory. A distribution like Ubuntu or Fedora is only different in the processes they launch and the configuration they read; they are both controlled by the same thing—the Linux kernel.

dotCloud (now Docker)²⁰ took the idea of operation system containerization one step further and made it easy to isolate an entire process. They built on the technology of LXC, which only saw mainstream adoption at the server or enterprise level, and packaged it up so that it was easier for the day-to-day user to isolate their own processes. With that, Docker was born as a technology geared toward the common person.

We were now one more step removed from virtual machines. Containers did not, and do not, need an "operating system" inside of them to work. Containers need the binaries and configuration to launch as the host's Linux kernel (or, in the case of a Windows container, the NT Kernel) handles running the process and restricting it. Couple this with a virtual machine, and now Linux containers can run on any operating system.

This has caused a few problems that virtual machines solved. If you have an M1 Mac and build a container, by default, it will only run on other M1 hardware. You need a Linux environment, provided by a virtual machine, to build it for an x86 architecture. Most of the containers developers run are built for x86 CPUs, so developers running M1 Macs have to run a Linux virtual machine, all of which is done under the hood of the Docker software.

So in a way, we moved away from caring about the platform to almost having to worry about the platform, like we were writing assembly. Three steps forward and two steps back.

From a deployment perspective, though, we now point to "I deployed in Kubernetes" or to a region more than a specific server. Container systems are now designed to run in clusters of machines across many regions for fault tolerance. You cannot point to where one copy of the code is running at any instance because it no longer matters. The container hosting software handles it for you.

To Serverless

Now, what if we take the idea of containers and strip out the container? What if we could tell some service that I am running a PHP application, here's my composer.json file, please run my application? In that case, we have what we call Serverless computing²¹.

The idea behind serverless computing is that now you deploy code—not a virtual machine or a container—to a host, and the host handles it all for you. You no longer worry

14 <https://www.vagrantup.com>

15 <https://laravel.com/docs/10.x/homestead>

16 <https://www.digitalocean.com>

17 <https://aws.amazon.com>

18 <https://en.wikipedia.org/wiki/Chroot>

19 <https://en.wikipedia.org/wiki/LXC>

20 https://en.wikipedia.org/wiki/Docker,_Inc.

21 <https://phpa.me/serverless>



about the hardware, the operating system, or even the software alongside your code; you say, "Here's my source code; please run it." Short of pointing at a vendor, AWS or Heroku, or Cloudflare, you have almost no idea where your code is being run.

Serverless is arguably the first modern technology we have talked about. Google released Google App Engine in 2008 using a custom Python framework with metered billing, which is another stable of serverless computing. You no longer get billed by machine or month; you get billed by execution time. Serverless software is only run when invoked.

If I look back at the demo I launched this morning, I launched it on a serverless provider. I told it I had a Node.js application and needed a Postgres database, and they handled everything else. While I have a say in the limits of the resources, that is about it. I could care about the geographic region it is launched in, but do I care? (To be fair, in a production environment, you would, but you worry about customer location more than "I need to run X server in Y datacenter".)

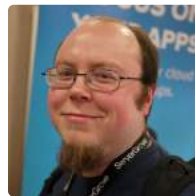
Serverless and PHP

Serverless share a bit of commonality with Platforms-as-a-Service like Heroku and, in many ways, are interchangeable. The most significant issue for PHP developers has been access to serverless platforms. This was the domain of other languages like Node.js and Python for many years. In some cases, vendors like AWS allowed for custom integrations, which give us tools like Bref²².

²² <https://bref.sh/>

In the larger scheme of things, tools like the Serverless Framework²³ (which Bref uses) help standardize being able to deploy serverless code to multiple providers. Software and standards like Apache OpenWhisk²⁴ make moving serverless code from one vendor to another easy. There are providers like Appwrite²⁵ that support PHP natively through their Functions offering.

The idea of serverless is interesting and almost harkens back to the early days of PHP development, where you could just choose almost any host and throw code up on it. The lack of support is the only thing holding it back for PHP developers. Next month let's take a look at the serverless landscape as it exists for PHP developers and how we might be able to leverage it.

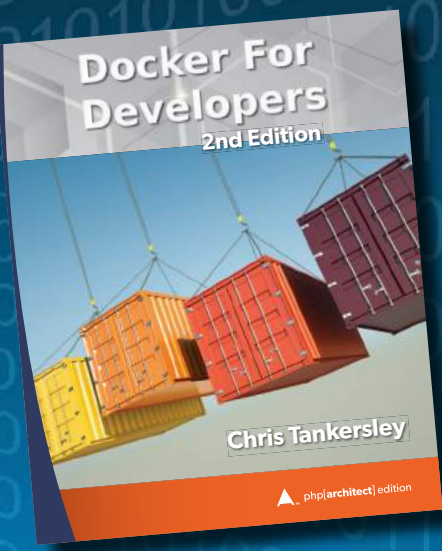


Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with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

²³ <https://www.serverless.com>

²⁴ <https://openwhisk.apache.org>

²⁵ <https://appwrite.io>



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>

Types of Tokens

Eric Mann

Terminology in security can be a finicky thing. When talking about either security-related or adjacent topics, it's best to be precise in what each term you choose actually means.

In most developer circles, explaining that I started in software development, working on WordPress, earns one of two responses: either a knowing nod from a fellow, self-taught engineer or a rude sneer from someone who thinks less of me due to my history. Regardless of the response, I know where I came from, and that background gives me a particular kind of respect for those in similar circumstances.

Specifically, I started working in security after experiencing a few scary situations in the world of WordPress. A wildly popular plugin I'd written exposed a fatal SQL injection vulnerability. Later, a client experienced a serious hack due to a poorly configured development environment. I moved immediately into security so I could learn more about the field—how to protect my own systems and how to teach others to stay safe.

One of the first lessons I learned was about vocabulary. Specifically, some terms I'd come to use in my WordPress days meant different things to other communities. A particular example is the **nonce**.

A real nonce is a “number used once”. It's a specific kind of token added to a request that the server remembers—any subsequent request with the same nonce is, therefore, invalid (as the nonce has already been used). In the WordPress world, however¹:

Technically, WordPress nonces aren't strictly numbers; they are a hash made up of numbers and letters. Nor are they used only once: they have a limited “lifetime” after which they expire.

Having used the word “nonce” in the WordPress way for so long, it was sometimes difficult for me to understand documentation from other tools and systems that used the term *correctly*. To help you avoid making the same mistake, I want to take a moment to explain some various terms that are, unfortunately, used interchangeably with the term “token”.

With any luck, you'll choose the proper term at the right time and avoid many of the missteps past-me took while working with nonces.

CSRF

Anyone building a userland application that allows for data to be input from a web browser should be leveraging cross-site request forgery (CSRF) prevention tokens. In reality, this kind of token is pretty similar to what a WordPress nonce really is.

When the backend of your application creates and renders a web form, it inserts a hidden field into that form containing an opaque token. This token could be random, or it could be tied to the specific request for the form.

A random token is used *exactly* as a nonce. The server keeps track of every token submitted during a particular time period and rejects the request containing the duplicated token when it sees a token being reused. Doing so helps the server avoid **replay attacks**—where a malicious third party tricks you into submitting the same request multiple times (likely triggering duplicate purchases or transfers or other behavior).

A non-random token is used to authenticate that the form submission is coming from the form created by the server. It's a way to prevent malicious users from crafting an automated attack against a server or potentially tricking

a user into submitting a forged request to the server from some other website (a “cross-site request forgery”). WordPress' nonces most closely mimic this behavior as they're tied to the server, the type of request being made, can be linked to specific other details on the page, and expire in a short window of time.

Access and Refresh

Whenever you log in to a system using tools like OAuth, you are granted two specific tokens to use in subsequent requests. The first is an access token², an opaque token that represents a valid, authenticated session with the server. To maintain your active session, you must present this access token with each subsequent request, usually as a header value. Access tokens have a limited scope (i.e., just the permissions granted to the user) and lifetime (they expire in a relatively short period).

The second is a refresh token³, another opaque value that grants no permissions but has a longer lifetime than the access token. Refresh tokens have but one purpose, a client can use them to request a new access token after an existing one has expired. Such



1 <https://phpa.me/nonces>

2 <https://phpa.me/access-tokens>

3 <https://phpa.me/refresh-tokens>



a request will present the client with a new access token and a new refresh token that can be used similarly down the road.

Both the access and refresh tokens *must be kept secret* by the client as they represent already-authenticated sessions. The server will have no way to identify who is using the token beyond the token itself (an attacker who steals your access or refresh tokens can impersonate you *perfectly*).

When leveraging OpenID, which itself is built atop OAuth, access tokens become less opaque and instead represent signed attestations of identity. Keep in mind, though, that the data embedded in such a token cannot be (easily) manipulated, and these tokens should be leveraged similarly to more opaque (read: random string) tokens within your application.

Peripherals

Third-party hardware used for authentication is also sometimes referred to as “tokens,” albeit these are in the hardware rather than the software world. Devices like the Yubikey⁴ present strong means of providing multi-factor authentication when logging into sensitive systems (like your root AWS console). Unlike “soft tokens,” like time-based one-time passwords, physical tokens are far more resistant to hacking and spoofing by malicious actors. They also require a physical presence and actual interaction for authentication (“something you have” in the authentication pyramid).

Clarify ...

When someone asks for your “token,” take time to clarify what exactly they’re asking for. Are they talking about CSRF protection for a web form? Authentication? Physical multi-factor authentication? Your miniature on a D&D map?

When in doubt, ask and clarify. It never hurts to explain, “I don’t know what you’re talking about”. The more dangerous solution is to *assume* you understand what they’re asking for and erroneously provide the wrong answer, detail, implementation, or level of security.

Software engineering is all about communication and removing avenues for misunderstanding. Now that you know some of the ways one particular security term can be used, you’re better equipped to discuss it with your peers and identify when you’re using the same term to refer to very different things.



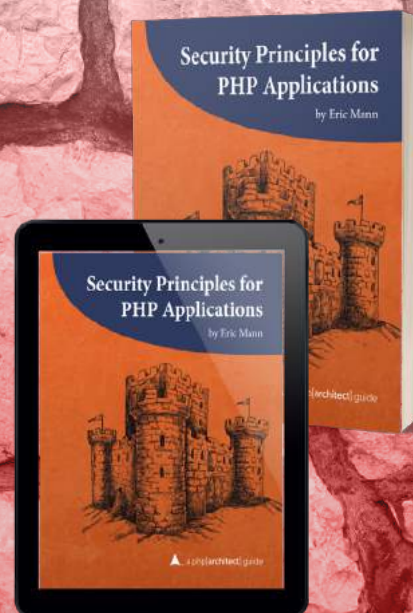
Eric is a seasoned web developer experienced with multiple languages and platforms. He’s been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: [@EricMann](https://twitter.com/EricMann)

4 <https://amzn.to/3HOME5l>

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





Welcome to Barrier-Free Bytes

Maxwell Ivey

Welcome to our new column, Barrier-Free Bytes. We hope to create a space to discuss and learn about accessibility and inclusion for all in our community. Without further ado, let's learn more about the author, Maxwell Ivey.

I'm Maxwell Ivey, known around the world as The Blind Blogger. I have gone from failed carnival owner to respected amusement equipment broker, award-winning author, motivational speaker, online media publicist, and host of the What's Your Excuse podcast.

Thanks to encouragement from many friends, I have realized that the 15 years I have spent working online dealing with websites, apps, and content with varying levels of accessibility makes me the ideal person to help others avoid the pitfalls of inaccessibility.

I love helping people and companies grow their businesses by making their products, services, and marketing content more inclusive of people with disabilities.

I'm honored to have the opportunity to share the knowledge and experience I have gained regarding accessibility and inclusion.

In this column, I plan to share things with you that will allow you to expand your understanding of the needs of people with disabilities so you can improve your coding to make it better for every user. As a result, I want to help you reach a new broader audience of highly loyal consumers of products, services, and content who will draw attention to your work.

Most importantly, I want to build relationships with you. My ultimate goal is that you will have the desire to make your work more accessible and will feel comfortable enough to reach out to me for help doing so.

It is also easier to answer an awkward question than to make people guess. So, feel free to ask away. Nothing is off-limits.

I look forward to starting conversations that will lead to a better

understanding of the needs of people with disabilities. Because I truly believe that achieving an accessible online world will only come through communication and collaboration.

Now, I'd like to tell you a little about how I got here and what makes me an expert.

I didn't start to be what I am now. I never planned to be an author, speaker, publicist, accessibility expert, or podcast host.

I grew up in a family of carnival owners, and all I ever wanted to do was to be part of the family business someday, running a carnival of my own.

I also grew up knowing that eventually, I would lose most, if not all, my vision to RP, retinitis pigmentosa. My vision loss started gradually, with a significant drop off in sight when I entered junior high school. My vision would stay constant until I set off to college. By the time I graduated, I had only limited light perception, which is what I have now. I can tell if a light is on or off, but only by looking directly at where the light is located.

Regardless my family did their best to include me in the business.

By age five, I was helping out in my Grandma's cotton candy stand, boxing the popcorn, making the snow cones, working the counters, and doing all manner of odd jobs.

Later I would work games and help set up and take down rides. You know, in all the years, no one ever complained about or even wondered about the wisdom of a blind guy helping put carnival rides together, even if my only job was to help carry heavy pieces of equipment.

Eventually, I would get involved in helping book events for our small seven-ride show.

I was blessed to work alongside my dad and brothers for over fifteen years before my dad's early death from lung cancer resulted in the closure of our small show.

We eventually joined up with my Uncle Albert's carnival, but there wasn't anything for me to do on his midway. I needed something to give my life meaning and decided to see if I could make a profession by helping others sell their used rides, games, concessions equipment, etc.

I started a website called The Midway Marketplace. www.midwaymarketplace.com¹

When I filed for the domain name, I didn't know how to build the website that went with it. I had so much to learn.

For starters, I had to figure out how to create and grow a website. As a blind person in 2007, this was very challenging. I didn't have much talent, ability, or money. It was before WordPress, Wifi, or Facebook. I eventually found my way to the w3c school, where I taught myself to code HTML.

I will admit that my early website wasn't perfect. In fact, thanks to some poor color choices, I was told more than once that my site was so vivid that Ray Charles and Stevie Wonder could have argued over it.

Instead of thinking my site was hideous, taking it down and giving up. I decided to leave it online. I figured it served its purpose in allowing people to see what was for sale. Since I focused on what I could do, the site helped me sell hundreds of thousands of dollars worth

¹ www.midwaymarketplace.com:
<http://www.midwaymarketplace.com>



of equipment on five continents before moving to WordPress and making the site more respectable.

In addition to the website, I had to learn how to recruit clients, set fees, write copy, build an email list, manage social media, start a blog, record videos, and much more.

People started commenting about how inspirational I was for taking on so many difficult challenges. They encouraged me to share more about being a blind entrepreneur.

That led to starting a second website, The Blind Blogger. www.theblindblogger.net²

Since then, I have written four books, two of which have been award winners. I have traveled the country solo, speaking and singing in public. I have appeared on hundreds of podcasts, instructed people on being great guests, booked friends and clients on shows, and started my podcast, What's Your Excuse.

All along the way, I have continued to come up against inaccessible websites, apps, platforms, and content. Instead of getting mad about that, I have always seen opportunities to educate the other person and work together to figure out a solution that works for everyone.

This leads me to why I jumped at the opportunity to write for this respected publication. I look forward to sharing with a new audience of people who are in a position to help make the world more accessible.

If you have a question about anything at all, please just ask. This offer doesn't just extend to questions about accessibility.

I know you may have questions about my vision loss or about the visually impaired that you may be uncertain about asking.

I am also open to creating future posts to address specific issues around accessibility and inclusion. Whether they are general questions or you need help addressing a specific problem with your own website, app, or content.

I look forward to hearing from you. I would love to hear about some of your experiences with accessibility. I also want to learn more about you as a person, your team, your business, your brand, etc.

Until next time take care out there, Max.



Maxwell Ivey is known around the world as The Blind Blogger. He has gone from failed carnival owner to respected amusement equipment broker to award-winning author, motivational speaker, online media publicist, and host of the What's Your Excuse podcast. [@maxwellivey](https://twitter.com/maxwellivey)

2 www.theblindblogger.net: <http://www.theblindblogger.net>



You're the Team Lead—Now What?

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

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

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



Databases as a Service

Joe Ferguson

Whether you're working on a legacy project or scaling up a modern application, one popular way would be migrating the database to a Database as a Service (DBaaS). These services can provide great benefits to teams that may not have a dedicated database administrator by offloading the database service to a managed host. This month we will explore the DBaaS cloud landscape and take a test drive through migrating an application to a DBaaS host.

Split Your Database from Your PHP

When managing databases for web applications, separating the database from the application layer is often beneficial. These layers could also be referred to as services, as the web layer would include PHP and nginx services, whereas the database layer would be MariaDB. Separation can be accomplished using a Database as a Service (DBaaS), which offloads the database service to a managed host. The first thing that might be unfamiliar is the database will no longer be on the same server as your PHP application or web server. The `DB_HOST` environment variable will no longer be `localhost` [`localhost: <<http://localhost>>`] or `127.0.0.1`, but instead, point to an IP address or hostname.

One of the main benefits of not having the database on the same server as your PHP application is increased scalability. When the database and application layers are separated, you can scale each layer independently, handling more traffic and data workloads without overloading any single server. This means that as your web application grows in popularity and usage, you can easily add additional resources to the database layer without scaling up the application layer and vice versa as needs grow.

In addition to increased scalability, separating the database and application layers can also improve security. By reducing the attack surface of your application and limiting the potential impact of any security breaches, you

can keep your data safer and more secure. This is especially important for web applications that handle sensitive user information, such as passwords or credit card numbers. DBaaS can often be more secure because, out of the box, they typically enforce and allow a list of sources that can access the database. This feature automatically firewalls the database service only to allow the hosts we specify. The hosts could be virtual machines, bare metal servers, or pods in a Kubernetes cluster. If you're not already strictly allowing sources to connect to your database servers, that's a great first step to increasing your security.

When you have removed the database server from the application server, it's easier to see where resources are being used. When you are able to focus on each layer separately, you can optimize them for their specific needs. For example, the database layer can be optimized for the storage and retrieval of data, while the application layer can be optimized for processing and serving web requests. In some cases, database queries can be large and memory intensive; using a DBaaS, you can use nodes focused on memory usage. For the application service, you may want to add processors in order to run more workers ready to handle requests. This can lead to improved performance and reliability for your web application. Upgrading databases also becomes an easier maintenance task typically handled for you. Because we've moved our database server off the application server, there is no threat of something

going wrong and causing an issue with our application.

Overall, separating the database and application layers can provide numerous benefits for web applications, including increased scalability, improved security, and easier management and maintenance of your infrastructure.

Separating the database and application layers allows you to use different technologies and platforms for each layer. For example, you may choose to use a MySQL database for your database and use PHP-FPM + nginx to handle your web requests. By separating the layers, you can take advantage of the strengths of each technology without being limited to a single platform. The database no longer has to fight over resources such as CPU and Memory from PHP, enabling better observability in your SQL queries.

In addition, separation can make it easier to troubleshoot and debug issues with your web application. When the layers are separated, it is easier to isolate the cause of any problems that arise. Whether this is a routing issue within Apache or nginx or debugging a slow query, you can go directly to the isolated source to troubleshoot. Doing so can save you time and resources in the long run, as you can quickly identify and resolve issues within the database server or the application server without troubleshooting the entire infrastructure.

Furthermore, separating layers can make migrating your application to a new hosting provider or platform easier. When the layers are separated, you can easily move the database layer to a



new DBaaS provider or a self-managed database server without worrying about the application layer. This can make it easier to take advantage of new technologies and platforms as they become available without having to rewrite your entire application.

The “default” PHP DB Server...

MariaDB and MySQL are popular database servers among PHP developers because of their ease of use, compatibility with PHP, performance, scalability, and the large community of users and developers that support them. Thanks to their seamless integration and communication with PHP, PHP developers can easily work with MariaDB and MySQL databases without learning a new database language or technology. Ultimately I blame the early web PHP tutorials all pushing me to use MySQL, which is always interesting to see as in the Python world, the de facto standard is PostgreSQL.

Both MariaDB and MySQL are known for their performance and scalability, making them ideal choices for web applications that need to scale quickly. They are capable of handling large amounts of data and high traffic volumes. Additionally, they have various features and tools that allow developers to optimize their database performance and tune it to meet their specific needs.

MariaDB and MySQL also have a large community of users and developers that support them, making it easy to find resources and support. Their popularity means that many third-party tools and integrations are available, making it easier to develop and deploy web applications. The ease of use, compatibility, performance, scalability, and community support make MariaDB and MySQL popular choices among PHP developers. This means to the average user that it's easy to search for problems and find help.



What's in a DBaaS Provider?

When choosing a Database as a Service (DBaaS) provider, there are several key features to look for. First and foremost, it is important to consider the provider's uptime and reliability. A good DBaaS provider should have a proven track record of uptime and reliability and a solid disaster recovery plan in case of any issues. All providers will have some regions where things are more prone to issues. It's highly worthwhile to look back at outages and spot what could be considered problem regions where issues are common.

Scalability is another important feature to consider. A good DBaaS provider should be able to scale your database up or down, as needed, without causing any downtime or data loss. This is especially important for web applications that may experience sudden spikes in traffic or usage. All of the major providers are going to charge you a premium for your node of choice because they're managing *everything* for you. Operating System updates, reliability, and security patches, most of which are completely automated.

Security is also a critical consideration when choosing a DBaaS provider. Look for a provider that offers strong encryption for data at rest and in transit and other security features such as two-factor authentication and regular security audits. Another aspect of security would be what compliance does the

service offer? Many small businesses are likely on the lookout for hosts who can offer at least SOC type 2¹ certifications. SOC 2 is a report which defines criteria for managing customer data.

It is important to consider the cost and pricing model of the DBaaS provider. Look for a provider that offers a transparent pricing model with no hidden fees or charges. Some providers may charge based on usage or storage, while others may offer a flat rate. Consider your budget and usage needs when comparing pricing models.

When choosing a DBaaS provider, look for one that offers reliable uptime, scalable infrastructure, strong security features, and a transparent pricing model. Especially look at any cost caps and the overage fees that may be associated with going over the capacity. By taking the time to research and compare different providers, you can find the right DBaaS solution for your web application.

Who are Some DBaaS Players?

You're likely already familiar; they are all essentially doing the same things. First, they all promise “ease of use” but what they really mean is you don't have to install and maintain the service. You should pick the provider you are already using to start out with unless you're already experienced and ready to drop your Azure DB into RDS' Aurora.

¹ SOC type 2: <https://phpa.me/cloud>

Microsoft

Azure Database

Azure Database for MySQL is a fully-managed database service that provides developers and DBAs with a fast, secure, and scalable MySQL database. You can easily deploy and manage your databases in the cloud without worrying about infrastructure management or maintenance. Azure isn't limited to only MySQL, as options are available for PostgreSQL, MariaDB, and Redis.

The typical sales pitch is Azure Databases provides built-in high availability, automatic backups, and advanced security features to help you protect your data and ensure maximum uptime. You can also take advantage of Azure's global data centers to deploy your databases closer to your users, reducing latency and improving performance. With Azure Databases, you can scale your databases up or down as needed, without any downtime, and pay only for what you use; just like most cloud offerings, you're paying for the infrastructure, but you don't have to manage the service.

Amazon RDS

Amazon RDS is a managed relational database service that provides easy setup, maintenance, and scaling of popular relational databases such as MySQL, MariaDB, PostgreSQL, Oracle, and SQL Server. The service is designed to make it easy for developers to set up, operate, and scale a relational database in the cloud. PHP developers can use Amazon RDS to easily manage and scale their database workloads without the need for complex infrastructure management.

One of the main stated benefits of Amazon RDS is its "ease of use". Of course, this AWS ecosystem is notoriously known for being overly complicated, so "ease of use" is doing a bit of work in this description. With RDS, developers can easily launch a database instance, configure it, and start using it within minutes. The service takes care of many administrative tasks associated with database management, such as backups, patching, and scaling, allowing developers to focus on their application logic instead of infrastructure. Additionally, Amazon RDS is highly scalable, allowing developers to easily scale their database workloads up or down as needed, with minimal or no downtime.

Amazon RDS also provides a high level of security for developers. The service offers built-in security features such as encryption at rest and in transit, network isolation, and IAM integration. Additionally, Amazon RDS provides automatic backups, automated software patching, and automatic failover, ensuring high availability and reliability for PHP applications. Overall, Amazon RDS is a reliable and easy-to-use database service that can help PHP developers manage and scale their database workloads in the cloud.

Digital Ocean Managed Databases

Digital Ocean Managed Databases offers a fully-managed, scalable, and highly available database service that can be used to power PHP applications. The service supports

popular relational databases such as MySQL, PostgreSQL, and Redis—and provides a simple and intuitive interface for managing your databases. You're probably noticing a pattern of "fully-managed", "scalable" and other buzzwords that it's absolutely impossible to avoid when discussing these services.

One of the main benefits of Digital Ocean Managed Databases is their simplicity and "ease of use". You're already in a much more user-friendly environment compared to Azure or AWS. With a few clicks, developers can easily create a database cluster, configure it, and start using it within minutes. The service also provides similar automatic backups, point-in-time recovery, and automatic failover, ensuring high availability and reliability for your applications. Additionally, Digital Ocean Managed Databases is highly scalable, allowing you to easily scale your database workloads up or down as needed, without any downtime, just as the others can as well.

Example Application Migration

Now that we've covered all the buzzwords and sold you on all the latest DBaaS options, what does it really come down to? We can take an example application, spin up a Managed Database at Digital Ocean, and configure our database connection. Let's start with creating a managed database server for our development environment.

A Managed MySQL v8 database can be created in Digital Ocean's cloud by selecting Databases and Create. For this example, we'll create a new Database Cluster in the New York 1 Datacenter. (See Figure 1)

Figure 1.



Choose a datacenter region

New York • Datacenter 1 • NYC1

Choose a database engine

<input type="radio"/> MongoDB	v6.0
<input type="radio"/> PostgreSQL	v15
<input checked="" type="radio"/> MySQL	v8
<input type="radio"/> Redis	v7

Choosing our database region and engine in Digital Ocean

We will use the cheap/small database configuration because we don't need anything fancy for local development. When ready to take this into production, you should go with one



of the CPU or Storage optimized nodes depending on your anticipated database workload. (See Figure 2)

Figure 2.

Choose a database configuration
You will be able to change the configuration at anytime after the cluster is created.

Plans

- ☒ **\$15/mo (\$0.022/hr)**
Basic / 1 vCPU / 1 GB RAM / 10 GB SSD / Connection limit: 75
- ☐ **\$30/mo (\$0.045/hr)**
Basic / 1 vCPU / 2 GB RAM / 30 GB SSD / Connection limit: 150
- ☐ **\$60/mo (\$0.089/hr)**
Basic / 2 vCPU / 4 GB RAM / 60 GB SSD / Connection limit: 225
- ☐ Additional plans

Finalize and create

Choose a unique database cluster name*
Names must be in lowercase and start with a letter. They can be between 3 and 63 characters long and may contain dashes.

phparch-dbaas

Specifying our configuration and cluster name

It may take a few minutes for our database to become available. During this time, you should set your current IP address to a trusted source so only you will be able to connect. (See Figure 3)

Figure 3.

phparch-dbaas **CREATING**

Overview Insights Logs & Queries Users & Databases Settings

READ ONLY NODES
You'll be able to create read-only nodes once this cluster has at least one backup.

DATABASE CLUSTER TOTAL COST
\$15.00 monthly rate.
This amount is provided, and does not reflect your month to date usage.

TRUSTED SOURCES
[IP Address] [Edit Sources](#)

CONNECTION DETAILS
Public network VPC network Connection parameters

username : doadmin
password :
host : phparch-dbaas-do-user-14051462-0.b.db.digitalocean.com
port : 3306
database : defaultdb
sslmode : REQUIRED

Users: doadmin

Copy Download CA certificate

Example Digital Ocean MySQL 8 Managed Database

Once the system has been provisioned, we can create a database and a user for our application to use. We'll use the username localdev and use the Default MySQL 8 encryption. (See Figure 4)

One trick with Digital Ocean-managed databases is you need to require SSL in your connection. The easiest way to do that in most PHP applications is by setting the DATABASE_URL value instead of breaking down all the connection parameters.

Figure 4.

Overview Insights Logs & Queries **Users & Databases** Settings

Users

Username	Password encryption	Password
doadmin	Default- MySQL 8+	show
localdev	Default- MySQL 8+	AVNS_poVbJ1LlufMd6YEzZJC hide

Digital Ocean will give you this connection string in their UI. We can update our application's .env to reflect our connection string. Our example would be `mysql://localdev:AVNS_poVbJ1LlufMd6YEzZJC@phparch-dbaas-do-user-14051462-0.b.db.digitalocean.com:25060/localdev?ssl-mode=REQUIRED`.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=fresh
DB_USERNAME=root
DB_PASSWORD=
DATABASE_URL="mysql://long.drawn.out.url"
```

With our .env file updated, we can run our migrations to verify everything works as expected:

artisan migrate

INFO Running migrations.

```
2014_10_12_000000_create_users_table .. 355ms DONE
2014_10_12_100000_create_reset_table .. 230ms DONE
2019_08_19_000000_create_failed_table .. 243ms DONE
2019_12_14_000001_create_tokens_table .. 356ms DONE
```

You will notice the speed at which our migrations are running may not be as snappy as before since we're now sending everything across the internet into Digital Ocean's MySQL server. Once you've made it this far, you're done; everything should work as expected with your database on someone else's computer.

No matter which service you choose, they will all sell you the same promise of scalability and ease of use. I recommend sticking with the platforms you're comfortable with and experimenting with other platforms to see what best suits your needs.



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](#)

Maze Rats, Part Three

Oscar Merida

In the prior two installments, we've explored maze-generating algorithms. Once you're in the labyrinth, you'll want a way to find the exit. In this installment, we figure out how to get from the maze's entrance to the exit.

Recap

"What if the treklins trap us in our maze," muses Archlin.

He grabs a parchment and tosses it toward a gold pen which springs to life with a wave of the wizard's hand. "I have your next assignment, apprentice."

The pen scrawls the following instruction: "For any maze you generate, create a map of the maze that traces the path from the entrance to the exit."

Maze-solving Algorithms

As with maze generation, using computers to find a route from an entrance to an exit cell is another well-trodden topic. There are two common approaches to solving this puzzle. If you know what the maze looks like, you can use the dead-end filling or shortest path algorithms. The random mouse, wall flower, and other algorithms are available if you don't know the maze's layout. Wikipedia summarizes common maze-solving algorithms¹.

The "Random Mouse Algorithm" sounds easy enough to implement:

It is simply to proceed following the current passage until a junction is reached, and then to make a random decision about the next direction to follow.

If you reach a dead end, you'd want a smart, non-random way for backtracking. However, depending on random decisions means it can be slow, especially for large mazes.

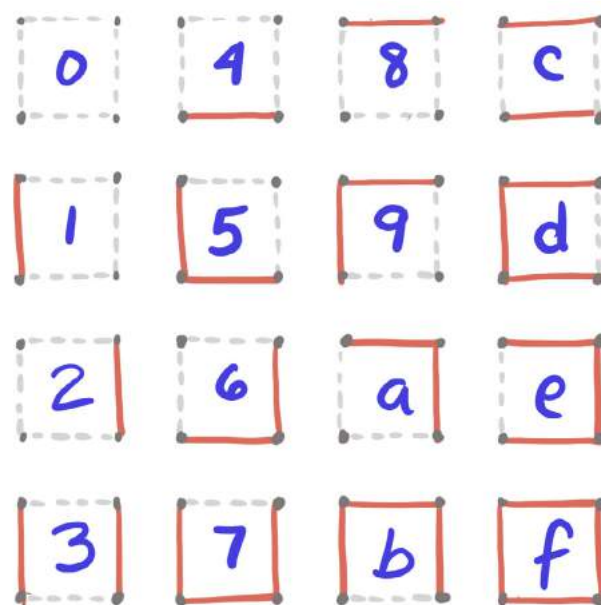
Since we know what the map looks like, we can use the "Dead-end filling" method.

The method is to: 1. find all of the dead-ends in the maze, and then 2. "fill in" the path from each dead-end until the first junction is met.

Finding Dead-ends

How can we identify dead ends? Recall from the first part of this puzzle series that we represent each cell of our maze with a hex value corresponding to the layouts shown in Figure 1.

Figure 1.



Dead-ends are any cells with only one exit. These correspond to any cell with a value of 7, b, d, e. We can loop through our array of cells until we find one that matches. If we don't find any dead-ends, we return false.

Removing a Dead-end

If we have any cell with that value, we can then "remove" it by putting up all the walls for the cell. If we make the cell's value f, that does it. However, putting up one wall affects up to two cells. We need the opposite of our wall-breaking function from prior solutions, as shown in Listing 1 on the next page.

¹ maze-solving algorithms: <https://phpa.me/maze-algo>



Listing 1.

```

1. public function fillDeadEnd(int $x, int $y): void
2. {
3.     // assume the cell given is a dead end, we could be
4.     // safe and check first.
5.     $current = $this->cells[$y][$x];
6.     // fill it
7.     $this->cells[$y][$x] = self::CLOSED;
8.
9.     // find any neighboring cell also need to update
10.    if (0 === ($current & self::SOUTH)) {
11.        $fill = self::FILL[self::SOUTH];
12.    } else if (0 === ($current & self::NORTH)) {
13.        $fill = self::FILL[self::NORTH];
14.    } else if (0 === ($current & self::EAST)) {
15.        $fill = self::FILL[self::EAST];
16.    } else {
17.        $fill = self::FILL[self::WEST];
18.    }
19.
20.    $newX = $x + $fill['xOffset'];
21.    $newY = $y + $fill['yOffset'];
22.
23.    // ensure we're still in the maze
24.    if ($newX >= 0 && $newX <= count($this->cells[0])
25.        && $newY >= 0 && $newY <= count($this->cells))
26.    {
27.        // recall we can add the value of the wall we
28.        // want to set to the current cell state.
29.        $this->cells[$newY][$newX] += $fill['opposite'];
30.    }
31. }

```

At this point, we have the framework of a naive solution. We can brute-force find a dead-end, fill it, then scan the whole maze for another dead-end, and fill it, until we have no more. Something like:

```

while ($end = $solver->findDeadEnd()) {
    $solver->fillDeadEnd($end[0], $end[1]);
}

```

Listing 2.

```

1. public function findDeadEnd(): bool|array
2. {
3.     // scan the maze for the first dead-end, return it
4.     foreach ($this->cells as $y => $row) {
5.         foreach ($row as $x => $cell) {
6.             if (in_array($cell, [0x7, 0xB, 0xD, 0xE])) {
7.                 return [$x, $y];
8.             }
9.         }
10.    }
11.    return false;
12. }

```

Recursive Efficiency

We can come up with a better solution than the brute-force one above. Each time we fill one cell in that loop, we start looking for the next end at our maze origin at [0][0]. When we fill a cell, we will likely move the dead-end by one space. Why do we have to start looking again every time? If we make a new dead-end, why don't we fill it right then and there?

We should only scan for a new dead-end when we've filled a passage all the way back to a branching point. We can do that with recursion. When we fill in a cell, if the new cell is a dead-end, we fill it. For small maps, this likely won't make a difference. Our solver should be performant if we want to work with very large mazes.

Recursion can be deceptively simple; just remember to have a "stop" condition to avoid infinite recursion. In our case, when we check if the neighbor of our filled-in space has become a dead-end:

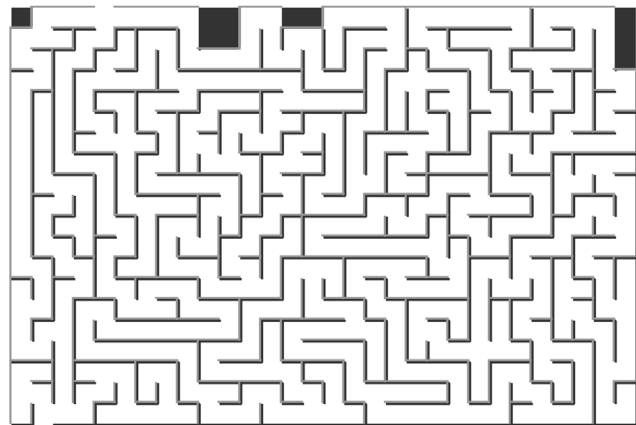
```

// ensure we're still in the maze
if (
    $newX >= 0 && $newX <= count($this->cells[0])
    && $newY >= 0 && $newY <= count($this->cells)
) {
    $this->cells[$newY][$newX] += $fill['opposite'];
    // if we make a new dead-end, go ahead and fill it
    if ($this->isDeadEnd($this->cells[$newY][$newX])) {
        $this->fillDeadEnd($newX, $newY);
    }
}

```

Conceptually, Figure 2 shows how a couple of iterations of the recursive approach close off dead-end paths.

Figure 2.



Presenting the Path

I adjusted the renderer from previous months to show the path in red. I then overlaid the solution path over the maze in an image editor. See Figure 3 on last page of this article.



The final code for the maze solver is shown in Listing 3

Listing 3.

```

1. <?php
2.
3. namespace OMerida\Maze;
4.
5. class Solver
6. {
7.     // constants for the walls of a cell
8.     private const WEST = 0x1;
9.     private const EAST = 0x2;
10.    private const SOUTH = 0x4;
11.    private const NORTH = 0x8;
12.    private const CLOSED = 15;
13.
14.    // fill direction helpers
15.    private const FILL = [
16.        self::WEST => [
17.            'opposite' => self::EAST,
18.            'xOffset' => -1,
19.            'yOffset' => 0
20.        ],
21.        self::EAST => [
22.            'opposite' => self::WEST,
23.            'xOffset' => 1,
24.            'yOffset' => 0
25.        ],
26.        self::NORTH => [
27.            'opposite' => self::SOUTH,
28.            'xOffset' => 0,
29.            'yOffset' => -1
30.        ],
31.        self::SOUTH => [
32.            'opposite' => self::NORTH,
33.            'xOffset' => 0,
34.            'yOffset' => +1
35.        ],
36.    ];
37.
38.    public function __construct(
39.        private array $cells
40.    )
41.    {
42.    }
43.
44.    /**
45.     * @return int[][]
46.     */
47.    public function getCells(): array
48.    {
49.        return $this->cells;
50.    }
51.
52.    private function isDeadEnd(int $cell): bool
53.    {
54.        return in_array($cell, [0x7, 0xB, 0xD, 0xE]);
55.    }

```

Listing 3 continued.

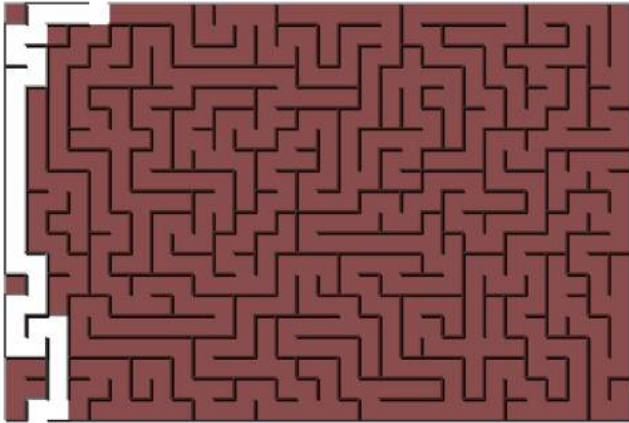
```

56.
57.    public function findDeadEnd(): bool|array
58.    {
59.        // scan the maze for the first dead-end
60.        // and return it
61.        foreach ($this->cells as $y => $row) {
62.            foreach ($row as $x => $cell) {
63.                if ($this->isDeadEnd($cell)) {
64.                    return [$x, $y];
65.                }
66.            }
67.        }
68.        return false;
69.    }
70.
71.    public function fillDeadEnd(int $x, int $y): void
72.    {
73.        // assume the cell given is a dead end,
74.        // we could be safe and check first.
75.        $current = $this->cells[$y][$x];
76.        // fill it
77.        $this->cells[$y][$x] = self::CLOSED;
78.
79.        // do any neighboring cell also need to update
80.        if (0 === ($current & self::SOUTH)) {
81.            $fill = self::FILL[self::SOUTH];
82.        } else if (0 === ($current & self::NORTH)) {
83.            $fill = self::FILL[self::NORTH];
84.        } else if (0 === ($current & self::EAST)) {
85.            $fill = self::FILL[self::EAST];
86.        } else {
87.            $fill = self::FILL[self::WEST];
88.        }
89.
90.        $newX = $x + $fill['xOffset'];
91.        $newY = $y + $fill['yOffset'];
92.
93.        // ensure we're still in the maze
94.        if ($newX >= 0 && $newX <= count($this->cells[0])
95.            && $newY >= 0 && $newY <= count($this->cells)
96.        ) {
97.            $this->cells[$newY][$newX] += $fill['opposite'];
98.
99.            // if we make a new dead-end,
100.           // go ahead and fill it
101.           $cell = $this->cells[$newY][$newX];
102.           if ($this->isDeadEnd($cell)) {
103.               $this->fillDeadEnd($newX, $newY);
104.           }
105.        }
106.    }
107. }

```



Figure 3.



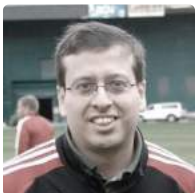
Give Me Directions

“How am I supposed to follow this map if I’m stuck in the maze with treklins chasing me?” The wizard Archlin exclaimed as you handed him a scroll with the exit path marked.

“This simply won’t do, apprentice.”, he mutters with a wagging finger pointed at you. “I want the directions in text. From the entrance, tell me if I should go forward, left, or right. Don’t make me think. I need to get out quickly!”

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 PsySH² 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](#)

² PsySH: <https://psysh.org>

GETTING OUT OF PHP DEPENDENCY HELL WITH COMPOSER

php[architect]

STAY UP-TO-DATE WITH THE LATEST PHP TRENDS AND TECHNOLOGIES

SUBSCRIBE

www.youtube.com/@Phparch



Create Observability, Part 1: Local Timing

Edward Barnard

“Keep it simple” often does pay off. Here’s a method of creating observability, just for yourself, that’s intentionally simplistic and rapid to implement.

Sometimes the simplest, most trivial methods can be the best solution to a problem of the moment. This month we’re exploring one such solution: a 30-line timing library based on PHP’s built-in `microtime()` function.

The next few articles will explore more advanced ways to create “observability” for various reasons, all within the Domain-Driven Design context. This month, however, is quick and direct. I’ll show you a bit of code I’ve rapidly implemented countless times over the decades. See Listing 1.

Here is a sample result from my development environment:

Began timing sections:

Listing 1.

```
1. <?php
2.
3. $start = microtime(true) * 1000.0;
4.
5. /* initialization/bootstrap not shown */
6.
7. $timing = new Timing($start);
8. $timing->measure('Began timing sections');
9.
10. $service = RenderBodyFactory::create($timing);
11. $service->execute();
12.
13. echo PHP_EOL . $timing->show(PHP_EOL) . PHP_EOL;
```

```
76.478 ms, 76.478 ms total
Began service execution:
20.140 ms, 96.618 ms total
100 input rows processed, 100 emails sent:
16606.550 ms, 16703.168 ms total
Completed service execution:
0.019 ms, 16703.187 ms total
```

The above output is from a command-line PHP script. Class `Timing` works either within a webpage or with command-line PHP. I have not explained the context yet, but let’s look at what those four output lines tell us.

Line 1 tells me that it took 76 milliseconds to get through bootstrap and PHP startup. That’s a long time, but typical for my development environment with most caching options disabled.

Line 2 tells me it took another 20 milliseconds to get through PHP startup. This, again, is because I have caching disabled for such things as the Twig rendering engine. I’ll show you

in the code below how I know the difference between Line 1 and Line 2.

Line 3 tells me that my script executed successfully, although it took 16 seconds to do so. I expected it to process 100 database rows and send 100 emails into a sandbox account. The output confirms the code performed as expected.

Line 4 shows a very short time, 19 microseconds. It’s actually useful to note that something ran fast, even if it was just the difference between one method call and the next. If that time were *also* slow, I’d have an indication that something was wrong.

What Problem are We Trying to Solve?

I intentionally showed you a result, and my conclusions based on the output, without telling you the context. I wanted to point out that it’s important to be clear on what problem you’re trying to solve!

Generally speaking, I use this timing library (the `Timing` class of Listing 1) for two reasons:

- Verify that the code, webpage, or process did what I expected it to do. Did it follow the expected code path? Did it process the expected number of iterations or records? Was it off by one, perhaps?
- Obtain actual, numeric timing measurements. Was the code fast enough? It’s generally counterproductive to optimize code without first measuring its actual performance. It’s also counterproductive to optimize code that’s already fast enough.

In other words, this timing library is a “quick and dirty” way to understand the code at runtime. I’ve created observability. I created a means of observing code behavior when it runs (e.g., webpage loads).

What About Existing Libraries?

Most frameworks and other logging applications provide a way to obtain this same information. The “debug toolbar” on the webpage in your development environment may be precisely what you need.

However, in my own experience, that’s not always the case. Something simple and lightweight could be the quickest way to solve a problem or produce the answer you need. These measurements add mere microseconds of overhead.



Mere microseconds of overhead, incurred a billion times daily, become significant. That's why `Timing` is not a production solution for heavy traffic.

This tool is really just for yourself. You're creating observability on the most local, immediate level. I find this useful during the development of something new and during problem analysis. It's a simple tool, quickly implemented, for a simple purpose.

Example Usage

Listing 2 shows the relevant portion of the PHP script.

The script collects the current `microtime()` at the very top of the script. Then, later after bootstrap, we pass `$start` (the beginning time) into the `Timing` class constructor.

The next line, calling `$timing->measure()`, records the fact that bootstrap required 76 milliseconds. Our purpose here is data collection rather than judging performance or anything else.

The next line, `$service = RenderBodyFactory::create($timing);`, creates some sort of service and passes our `Timing` class into that service. After creating the service, we call `execute()` in that service.

Finally, we display results by calling `$timing->show()` and echoing that result to the console.

The service collects timing information as follows:

```
public function execute(): void
{
    $this->timing->measure('Began service execution');
    $this->walkInputQueue();
    $this->timing->measure('Completed service exec');
}
```

The inner method `walkInputQueue()` finishes by reporting:

```
private function walkInputQueue(): void
{
    /* processing not shown */
    $this->timing->measure(
        "$inputRows input rows processed, " .
        "$sendCount emails sent"
    );
}
```

Fine-grained Usage

The `Timing` class uses its own private array `$this->timings[]` to collect timing information. If, for example, I wanted more timing detail, I could insert `$this->timing->measure()` calls inside the processing loop.

We know that 100 emails were sent in 16,606 milliseconds or 16.606 seconds. That means this particular script takes 166 milliseconds per email.

Is that really true?

This is where “observability” can help. Suppose I add a `measure()` call after each successful email send. I might be surprised! It might turn out that the *first* email send takes 12

Listing 2.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. /* namespace according to need */
6.
7. use function implode;
8. use function microtime;
9. use function sprintf;
10.
11. class Timing
12. {
13.     /** @var float */
14.     private $start;
15.     /** @var float */
16.     private $intervalStart;
17.     /** @var array */
18.     private $timings = [];
19.
20.     public function __construct(?float $start = null)
21.     {
22.         if (null === $start) {
23.             $start = microtime(true) * 1000.0;
24.         }
25.         $this->start = $start;
26.         $this->intervalStart = $start;
27.     }
28.
29.     public function measure(string $description): void
30.     {
31.         $current = microtime(true) * 1000.0;
32.         $total = $current - $this->start;
33.         $interval = $current - $this->intervalStart;
34.         $this->timings[] = "$description: " .
35.             sprintf(
36.                 '%.3f ms, %.3f ms total',
37.                 $interval, $total
38.             );
39.         $this->intervalStart = $current;
40.     }
41.
42.     public function show(
43.         string $separator = '<br>'
44.     ): string {
45.         return implode($separator, $this->timings);
46.     }
47. }
```

seconds to receive a response and that the rest go through much more quickly.

This means the performance profile is quite different from what I had assumed. The first was quite slow, and the rest were fast. This was going to a developer sandbox. Now I have to ask myself, is the production performance profile similar? Probably not!



That's why building-in this type of "quick and dirty" observability can be so helpful. I can quickly gather as much or as little information as I need to understand the problem at hand.

Production Safety

Suppose you are measuring "mainline" code, likely to be executed countless times over the course of a day or week. In that case, you'll want to be sure to remove the `measure()` calls once your investigation or feature development is complete.

On the other hand, if you're developing an administrative tool that's only used once a day, or even once a year, adding ten microseconds of overhead per day is not significant. Leaving the measurements in place might be useful. The next time it's used, a month or a year from now, you might need to examine the same sort of question all over again.

I currently lock down my administrative-tool measurements like this:

```
if ('devel' === OPERATING_ENV) {
    echo PHP_EOL . 'Timings only displayed in ' . PHP_EOL;
    echo 'DEVEL environment' . PHP_EOL;
    echo PHP_EOL . $timing->show(PHP_EOL) . PHP_EOL;
    echo PHP_EOL . 'Complete: ' . __FILE__ . PHP_EOL;
}
```

If the tool is run once a day (or even once every five minutes as a timed task), as much as a thousand microseconds a day is not significant. In other words, leaving the `measure()` calls in place doesn't matter. It might on high-traffic webpages, but not here.

However, at the same time, the information might be distracting to non-developers in the production environment. The above `if()` ensures nothing will be seen. But the timings are there if needed, and the `if()` can be commented-out if you know it's there. Leaving the above `if()` visible in the

main script ensures any developer *can* know it's there without having to dig deeply into the code.

Summary

First, clearly understand what problem you're trying to solve. Can creating some form of observability help you better understand the system in question or help you understand and solve the problem of the moment?

I find that creating immediate observability as I develop or investigate PHP code can be a quick and easy way to help find the answers I need. We created and implemented a timing library based on PHP's built in `microtime()` function. We then collected timing information.

Specifically, this technique can help answer two questions:

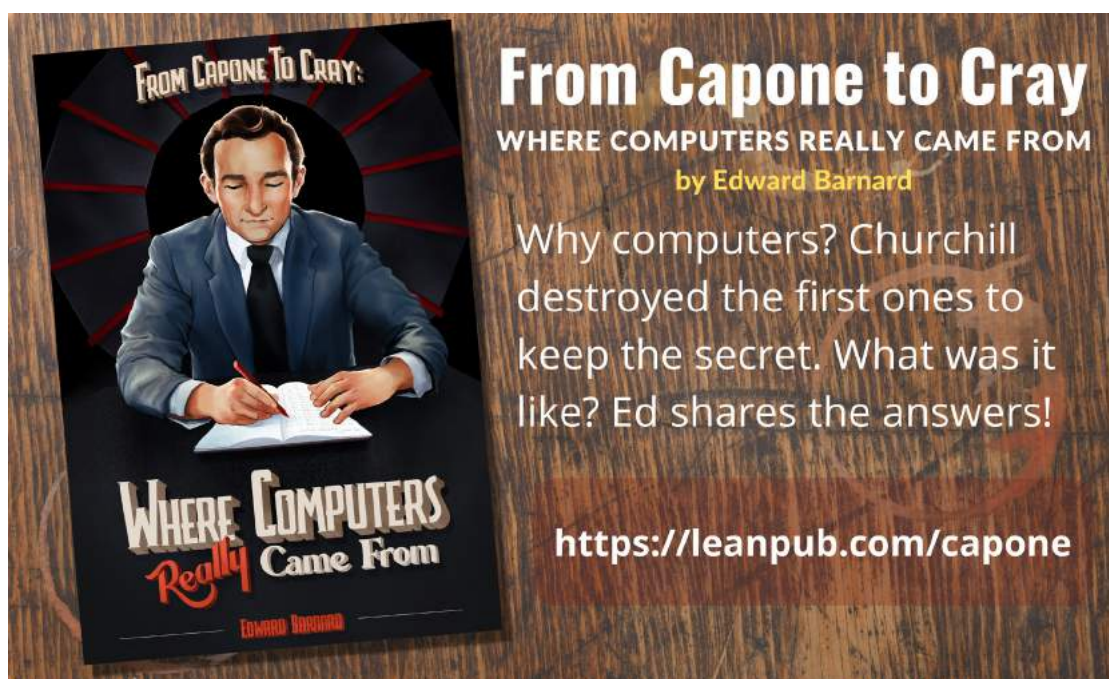
- Did the code do what I expected it to do?
- What was the actual numeric software performance, section by section, iteration by iteration?

I have found that answers to those two questions lead to a quick and deeper understanding of the software. I'm often surprised by the actual measurements, which means it's a good thing that I bothered to "run the numbers".



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

[@ewbarnard](https://twitter.com/ewbarnard)



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>



PER: Coding Style

Frank Wallen

No more PSR's to discuss at the moment...say it isn't so! Not to worry, we'll look at a new type of Recommendation: the PHP Evolving Recommendation, or PER, in the meantime.

Some of you may have been wondering, "What happens in *PSR Pickup* when we run out of PSRs?" We've reached that point, for now, and there still are four out in the field in Draft status:

1. PSR-5: PHPDoc Standard
2. PSR-19: PHPDoc Tags
3. PSR-21: Internationalization
4. PSR-22: Application Tracing

There is a PHP-FIG Discord server with discussion channels on PSRs in process, PHP in general, and FIG-related topics. Easily join the discussions by visiting this link: <https://discord.gg/php-fig>¹. As soon as those PSRs are released, we will discuss them here, but in the meantime, let's look at a new type of Recommendation: the PHP Evolving Recommendation, or PER.

What's the difference, one might ask? A PSR is considered to be static and unchanging, other than some tightening of the documentation due to changes in new PHP versions, like types, functions, etc. A PER, on the other hand, is intended to be a living document, growing with the PHP community at large. In my opinion, this shows how much PHP-FIG thinks about the PHP community and its needs and the serious efforts made to support it. The Discord discussions are lively, with many bright members and leaders of PHP.

Let's get started on the first **PER: Coding Style**². This PER builds on and expands two previous PSRs: "Basic Coding Standard" (PSR-1³ and "Extended Coding Style" (PSR-12⁴), and effectively replaces PSR-12. However, it does not change anything described in PSR-1 and PSR-12. As PHP adds new functionality, the Coding Style document will also change to support it. In fact, it will be using semantic versioning to track changes and is currently at version 2.0.

PSR-12 was first released in 2019, and it wasn't long after that PHP had significant changes and improvements, especially in versions 8 and 8.1. Although it was comprehensive and established excellent guidelines on style, it was obvious that the document would have to grow. This PER essentially starts where PSR-12 left off but also adds additional clarity on previously established style guidelines.

I considered describing the PER's rules here but realized this would present a long article, and it would likely be best to read the rules in the document itself. Instead, I want to present some of the discussions they were having and the challenges of deciding whether something should be present in the document. Strict types declaration (`declare(strict_types=1)`), for example, was ultimately decided to be out of the scope of the document. It came down to recognizing that it falls under a Coding Standard rather than a Coding Style. Style should not impact the performance of the code, which strict types definitely will. In my opinion, it is good practice to use strict types. However, using short forms over long ones (`bool` over `boolean`, `int` over `integer`) *does* fall into scope.

The FIG group conducts numerous surveys before completing discussions and making decisions. In the meta-document for **PER: Coding Style**, they offer the results of their surveys in two groups: PHP-FIG Project Representatives⁵ and General Non-Representatives⁶. It is interesting to see how project representatives vote on each issue or question, considering how each could impact their project. In most cases, it's overwhelmingly positive as each point has

⁵ <https://phpa.me/fig-cs>

⁶ <https://phpa.me/git-cs-nrv>



¹ <https://discord.gg/php-fig>

² <https://www.php-fig.org/per/coding-style/>

³ <https://www.php-fig.org/psr/psr-1/>

⁴ <https://www.php-fig.org/psr/psr-12/>



likely been deeply discussed, but there were a couple that were “contentious” at votes of 13 *for* and 3 *against*. Those were “Compound namespaces with a depth of two or more MUST not be used” and “Declare statements in PHP files containing markup.”⁷ Among non-representative voters, the questions were primarily positive, with only two coming in below 90%, and only one reaching as low as 88.5%. Again, I believe that has a lot to do with the continued discussions as the community opinions form. Some of these discussions started in 2015! It’s like a new river that meanders as it establishes its presence in the terrain; it was 2022 when the PER was concluded as accepted!

Conclusion

This PER is a significant document to review, but there should be little need for debate on its relevance to coding in

⁷ <https://phpa.me/fig-cs-declare>

PHP today. They are only recommendations, but those standards improve our communication, workflow, integrations, and more. You will find the link to the document⁸ in the footnotes below, and one to the Github repository⁹, where you can review the votes and the contributors involved.



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)

⁸ <https://www.php-fig.org/per/coding-style/>

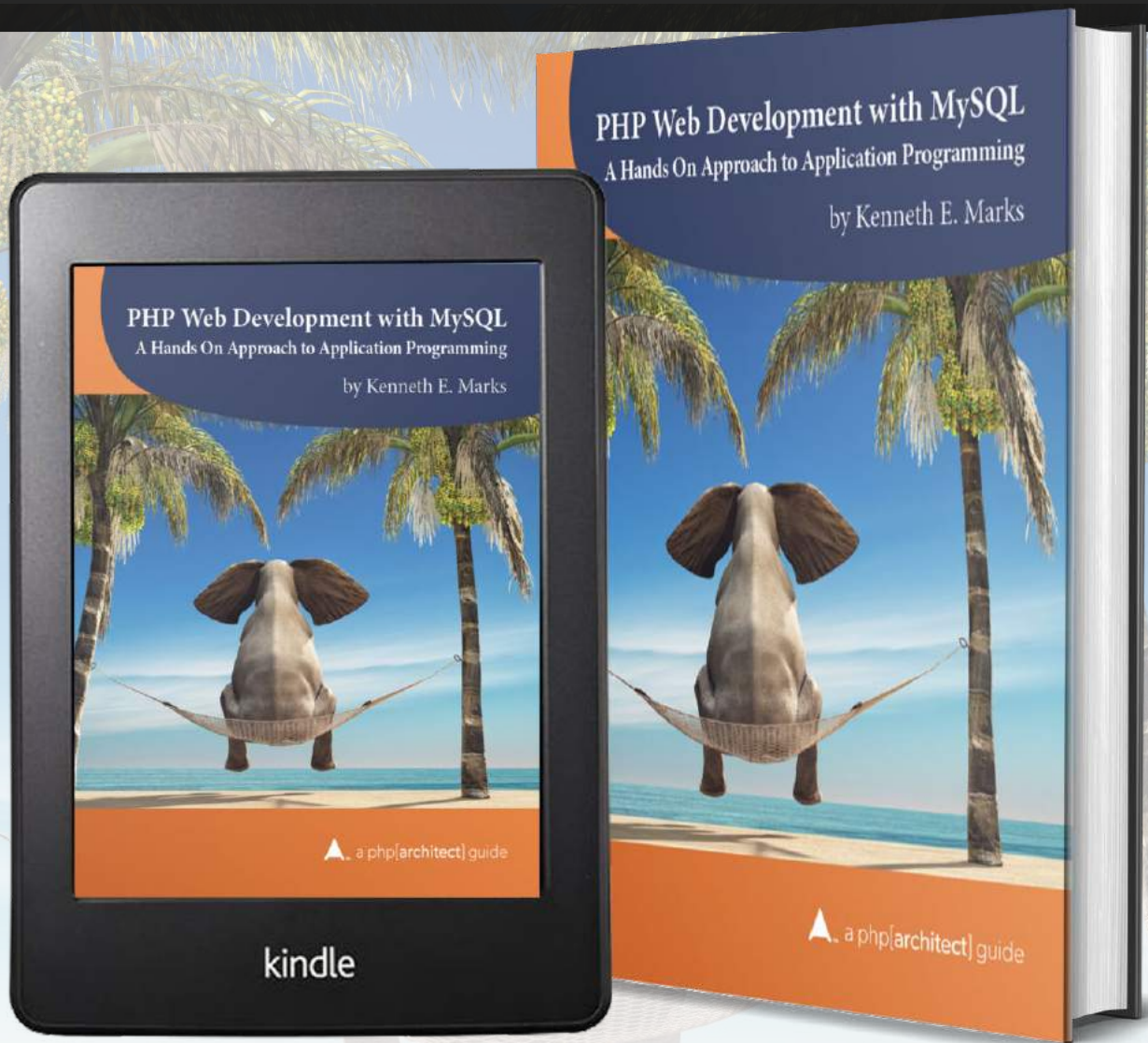
⁹ <https://phpa.me/fig-cs-releases>

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>





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.

Coming Soon
<https://phparch.com/books>



Defensive Programming For Laravel

Matt Lantz

Building Laravel applications can be a wonderful experience. As a full-stack framework, it provides numerous avenues for creating a well-functioning scalable application. Laravel is, by default, set with layers for security measures, and there are multiple packages to add extra standards, ensuring your application will remain as secure as possible.

Despite these additional tools and built-in security measures, every Laravel developer should practice some defensive programming and ensure that they align with the top ten recommendations from OWASP. Even following these cannot guarantee your application will be 100% secure, but it can dramatically reduce your vulnerabilities to an attack. Regardless of your preferred front-end framework or vanilla Blade templating structure, there are practices on the Front-End and Back-End of your application that can help to avoid uncomfortable phone calls with clients or customers.

Defensive programming is a practice where developers presume their users will not enter valid safe data. It's a philosophy of anticipating the worst situations and most aggressive bad actors using the application rather than users who do not want to break your application. There are six main conventions to ensure you're following defensive programming patterns.

Never Trust User Input

It's better to allow-list rather than deny-list. Allow lists ensure you know who enters data into your application and what they submit. Furthermore, developers should ensure that all inputs are validated and stripped of tags or script injections where possible.

Use Database Abstraction

There may be some debate over ORMs, but it is best to use toolings like your framework's ORM, such as Eloquent, PDO, or Doctrine. These

database abstraction tools assist in removing potential SQL injections and any holes that can end up in hard-written SQL statements as database engines change over time.

Framework Over Custom

Identify optimal places for commonly used community-driven components rather than custom-build every component in your application. However, it is possible to have too many dependencies, which can create a maintenance problem. If possible, utilize your framework's core components rather than third-party packages. While ensuring the third-party packages you do use are well battle-tested.

Don't Trust Developers

You can trust other developers in your team and yourself or optimize your tooling and processes to assume developers are writing insecure code. Ensure you have aggressive QA team members and are implementing penetration testing for your application because developers will make mistakes. It also means do not trust any other developers, including the authors of your dependencies. Ensure you inspect your dependencies for flaws and confirm they are adequately tested.

Write SOLID Code

Follow a healthy approach to writing SOLID code. Avoid the use of uninstantiated object properties, and ensure objects are never in an inconsistent state. Where possible, use immutable

objects. Types are valuable and can help ensure that your code isn't allowing overly complex inputs from the user.

Write Tests

Regardless of your main coding patterns, it would be best if you always wrote tests. Whether you follow TDD or write tests after you get primary functionality is up to you and your team. Good quality tests are structured to have low coupling and test a variety of possible outcomes. Rather than focusing solely on the happy path, it's in your best interest to write a handful of unhappy path tests that will ensure you prevent as many attacks as possible.

By adhering to these concepts, you have a proper mindset for securely architecting your code base. Regardless of how many users your application will have or how many potential bad actors may infiltrate your application, you will likely have written your code in a way that protects both the data and the business. However, we cannot review defensive programming for Laravel without exploring the ten most common attacks defined by OWASP. We can easily explore how using a defensive mindset and some standard community packages, we can address each of the OWASP concerns.

Cross-Site Scripting

XSS is an attack where the attacker injects malicious scripts into the user's browser. One of the main challenges with this attack is that the scripts are automatically executed when received. Perpetrators do this by injecting scripts into your application. Step one in protecting against this is sanitizing your



inputs, ensuring you're not disabling CSRF. You can create middleware which performs further sanitization of inputs, for example, removing all possible HTML or scripting tags. Adding Content-Security-Policy headers can also help protect against this attack method, but it should be considered a secondary measure. Lastly, and this applies to all of the below attacks, avoid using `request()->all()` in all possible cases. It's also important to be mindful of your uses of `$model->update([])` as this can create holes in your security practices.

SQL Injection Attacks

Similar to what we discussed earlier, SQL injections can compromise a server's cookies, web forms, or HTTP posts to manipulate data out of the database. Attackers exploit this through poor protection measures in input fields. By sticking to Laravel's ORM, Eloquent, you reduce the probability of enabling SQL injection attacks, as it handles some sanitization for you. Writing custom SQL statements and injecting request inputs directly into the strings will increase your chances of enabling an SQL injection attack.

Broken Authentication

Compromised credentials are a typical attack pattern, whether by brute force, credential stuffing, or dictionary attacks; many applications are vulnerable to this attack. The simplest way to ensure your Laravel app is protected by these is by implementing rate limiting on logins and denying the use of compromised passwords for your users. You can do this by adding `uncompromised()` to your default password rules like this:

```
Password::defaults(fn () => Password::min(8)
->mixedCase()
->letters()
->numbers()
->symbols()
->uncompromised()
);
```

You can further enhance this by reducing remember me to 30 days rather than the default 400. You can do this by modifying the web guard to match this:

```
'web' => [
    'driver' => 'session',
    'provider' => 'users',
    'remember' => 43800 // Set the duration here (minutes)
],
```

Otherwise, you can add MFA to your application, which many of the Laravel starter kits make very easy to reduce the chances of making your authentication system susceptible to attacks.

Drive-by Download

This attack method occurs when a bad actor infects your application through security flaws in the server operating system, web browser, or even your application. The attack method is more of an attack against your users than your application. However, the best way to protect against this is to keep your OS updated, not just as an OS but all security patches as well. Developers must also keep track of their dependencies on the OS or language levels and install only what is necessary on their servers. Dependency pruning—removing all components not needed for your application—can be another pathway to help protect against this attack method. Overall, there is little you can do in a Laravel app that will assist with this, but a good DevOps maintenance policy will protect your application.

Password-based Attacks

Similar to the broken authentication attack concern, implementing components like MFA and denying the use of compromised passwords can ensure you are less vulnerable to a password-based attack. Using rate limiting on any routes that could be brute forced is also a handy solution to these attacks. You can implement more complex rules for default passwords in Laravel instead of denying the compromised passwords.

Fuzzing

A fuzzing attack consists of bad actors impeding your application with large amounts of random data to get it to crash. The Fuzzing tools these bad actors use can then identify an application's "weak" spots. This type of attack also helps perpetrators identify if they can exploit known security issues. Similar to Drive-by Download, ensuring your servers are up to date and ensuring your codebase's dependencies are up to date are your best modes of defense. Lastly, rate limiting can also help defend this attack method.

Using Components with Known Vulnerabilities

Though significant progress has been made in addressing this problem with tools like composer and automated scanners for vulnerabilities in dependencies, developers should not assume that third-party components have extraordinary security compliance measures. Though the automated systems can help identify out-of-date components, not all third-party developers correctly identify packages as abandoned. When in doubt, utilize tools like `https://endoflife.date/` to help ensure that your application is not relying on code that no longer has security support or is at its end of life.

DDoS

Very few developers have yet to hear of a DDoS attack, especially since the days of Anonymous running their very public attacks. A DDoS attack is overwhelming your server by botnet systems making many requests. Ensuring you have a CDN configured can be helpful; tools like CloudFlare's mitigation system for DDoS attacks can also be cost-effective. Though it's convenient to think that autoscaling will mean that a bombardment will not hinder your application, it will significantly impact your business's operating costs. Outside of network configurations and tooling, you can implement an Application Firewall, either in your Cloud Provider or, in some cases, in your application directly with middleware. This approach can be helpful when DDoS attacks are concealing injection or XSS attacks.

MiTM

Man-in-the-middle attacks are common in applications that are not using HTTPS. The perpetrator intercepts the data being transferred between two parties. Without encryption, the attacker can extract data from the application, which can be personal data. Given the ease of Let's Encrypt certificates, there is almost no reason to have HTTP-based traffic. Furthermore, ensuring cookies and session data are encrypted can help avoid any other extraction efforts that MiTM attackers may attempt to exploit. However, these protections go beyond user-facing components. Ensure you have SSL configured for any external components in your Laravel application, be it databases, caches, search indexers, or any other platforms you connect with.

Directory Traversal

Overall this attack is where the attacker attempts to move up and down the path hierarchy on the server to access files and content. Laravel's default structure helps deter this by

maintaining a public folder with nearly nothing but public assets if you build them there. Other ways to protect against this are by always keeping S3 buckets private, placing a session-based route in front of any assets you need to load, or using a temp URL for file uploads.

Whether or not you expect your application to scale and have hundreds of users or even thousands, you should anticipate that an attacker could hit you when you have as few as one user. If you have customers in your application, you should have clear documentation on contacting them and informing them of any data breaches or system attacks. It would be best to have a plan for restoring your system or data when an attack occurs. Rather than assuming you will not likely get attacked, it's better to be prepared for tomorrow's attack. Following some defensive programming concepts, you can implement solutions within Laravel applications to address the common OWASP concerns and deliver a high-quality application to your users. It's a complex process to determine how to architect software to address business needs, but it's significantly harder to deal with a business having its systems destroyed by an attack. Hopefully, with a little extra effort in the former, you can avoid the latter.



Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. @MattyLantz

Beyond Laravel


An Entrepreneur's Guide to Building Effective Software

by Michael Akopov

Harness the power of the Laravel ecosystem to bring your idea to life.

Written by Laravel and PHP professional Michael Akopov, this book provides a concise guide for taking your software from an idea to a business. Focus on what really matters to make your idea stand out without wasting time on already-solved problems.

Order Your Copy
<https://phpa.me/beyond-laravel>





Do We Deserve to Be Here?

Beth Tucker Long

For more years than anyone cares to count, programmers have made jokes at PHP's expense, shuddered when someone mentions being a PHP programmer, or written long rants about how PHP isn't a real option for professionals. How has this affected us?

It is exhausting trying to prove you measure up to other people's standards. For years, I have heard developers repeatedly need to justify their use of PHP. "I can build things so quickly with it." "The performance is really good." "The security issues are caused by bad developers, not by the language." "It is used on so many sites; you need to be able to write PHP to be hireable."

And my favorite: "PHP used to be terrible, but now that we have , it's now just as good as ."

Here's the thing, PHP isn't any good at pretending to be other languages. And it shouldn't be. PHP is amazing in its own right.

Yet, for a while, discussions about core development were very focused on adding in all of the new features of competing languages. It caused many arguments about what "modern" languages have to contain to be worthwhile. It led to many toxic comments about how you could not call yourself a developer if you didn't use a certain language construct. Not our finest moment as a community.

Looking back, it really feels like this was our moment of burnout. We were so used to fighting to justify our use and love of PHP that we lost track of who we were fighting against. We were too tired and frustrated to notice that we were attacking our own community. We splintered. We weakened. We faltered and had to take a step back.

Luckily, we have wonderful, dedicated people who found ways to glue us back together. Collaboration invigorated the community. Renewed interest in internals brought new ideas and fresh talent. We rebuilt slowly and steadily.

We have more than a decade of steady releases every year. We have a standards group evaluating ways to keep us all working together seamlessly. We have a foundation to ensure the continued development of the language.

PHP is thriving.

We have survived years of having to justify our existence. We have grown up and earned the right to stop caring what other people think. We don't need to be the most popular language or the most modern. We don't need to mimic other languages to be relevant, and we definitely don't need to waste time and effort trying to prove that PHP is "good enough". We have an absolutely amazing community, a plan for the future, and a solid language we can rely on. PHP rocks, and we are going to focus our efforts on being authentically PHP. All the haters can just mind their own business. Long live PHP!



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

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

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

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

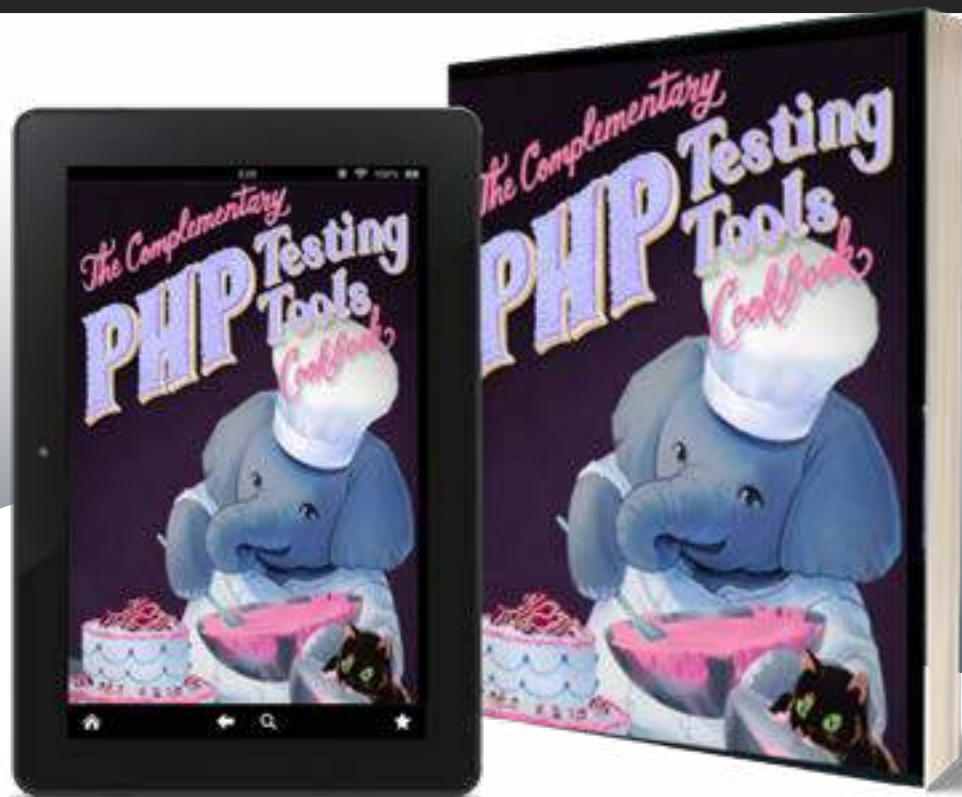
4 Full Stack Madison: <http://www.fullstackmadison.com>

The PHP Foundation

The PHP Foundation is a collective established with the non-profit mission to support, advance, and develop the PHP language. We are a community of PHP veterans, community leaders, and technology companies that rely on PHP as a critical digital infrastructure. We collaborate to ensure PHP language long-term success and maintenance.

DONATE TO OUR COLLECTIVE





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



PhpStorm

Enjoy
productive
PHP

jetbrains.com/phpstorm