www.phparch.com

December 2023
Volume 22 - Issue 12

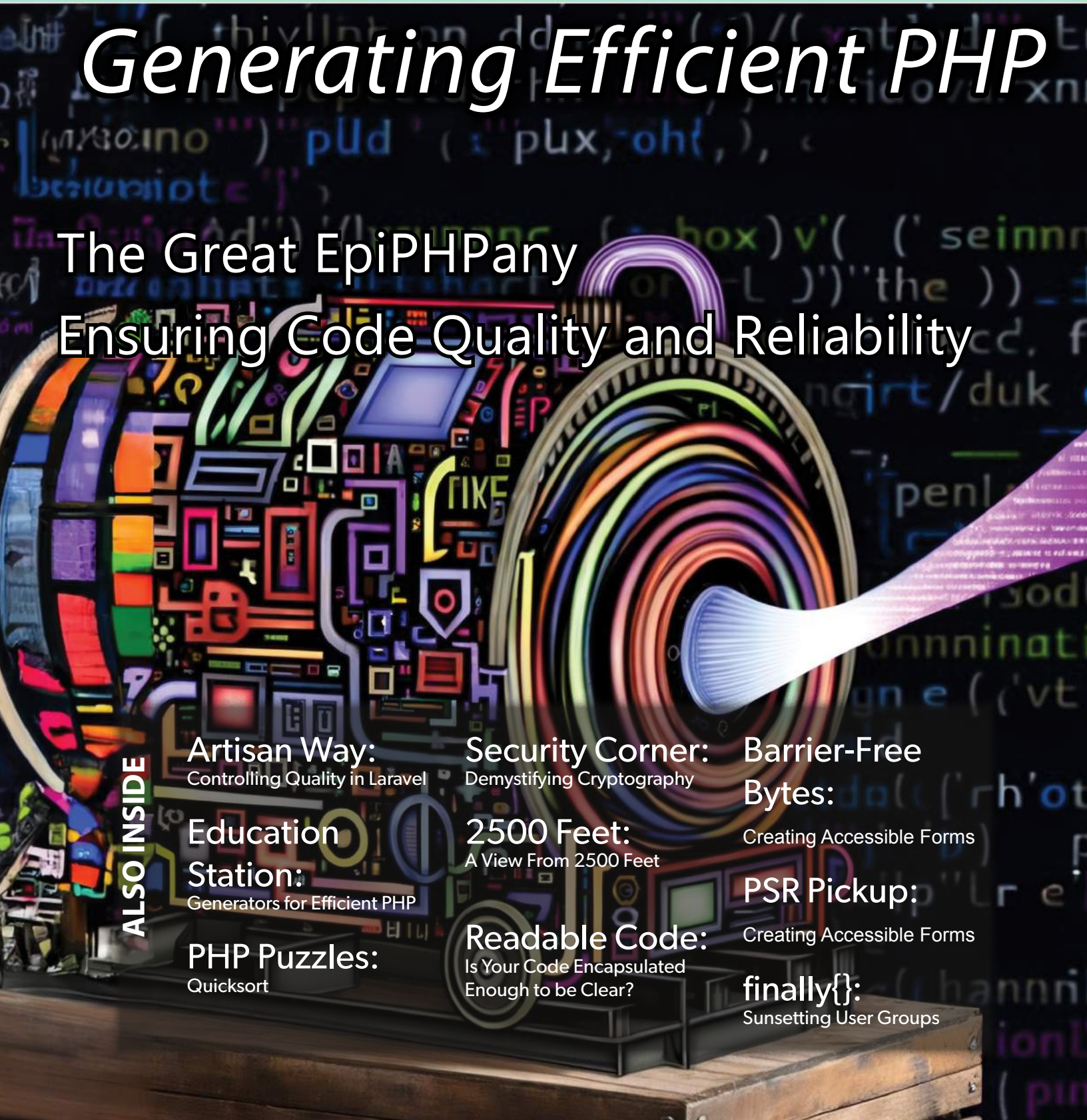
php[architect]

The Magazine For PHP Professionals

Generating Efficient PHP

The Great EpiPHPany

Ensuring Code Quality and Reliability



ALSO INSIDE


HAM Radio and Linux | DevOps | Cloud Native | Developer | Embedded | Home Automation | IoT | Kernel | Low Level Systems | Observability | Artificial Intelligence | Open Government | Kubernetes | Security | Systems and Infrastructure |
REGISTER NOW!
SCaLE
MARCH 14-17 • Pasadena, CA
21x
Don't miss North America's largest annual
community organized Open Source gathering!
50% OFF
Use Code ARCH
for an exclusive discount
socallinuxexpo.org
SCaLE
SOUTHERN CALIFORNIA LINUX EXPO
Want to Sponsor, Exhibit, or Volunteer at SCaLE?
Get in touch with us: sponsorship@socallinuxexpo.org

CONTENTS

Edited by Santa's Elves

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.

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

This month we have some very exciting news, the speaker lineup and schedule has been published for php[tek] 2024. We're extremely excited about the upcoming Tek lineup and think there is something there for everyone. Check it out for yourself at https://tek.phparch.com/schedule. We hope to see you there in April. Now, let's talk about this months release.

In this month's feature, Eric Ranner brings us *'The Great EphiPHPany'*. Eric takes us on his journey of discovering the world of programming, specifically PHP. He started in the field of hospitality and sought out programming in an effort to improve various social games he was using at his events. While researching PHP for web development, Eric discovered firsthand that the hate is real for PHP and thought maybe it wasn't the right solution for him. Check out this read to see how Eric realized that PHP has a way to do everything he could ever want to do while coding a website and more.

Our next feature is by Godstime Aburu and he's sharing some knowledge about testing in *'PHP Unit Testing: Ensuring Code Quality and Reliability'*. This thorough read will walk you through PHP unit testing and encourage you to fully embrace Test Driven Development (TDD). Godstime also discusses more progressive unit testing like mocking and stubbing as well.

This month's PSR Pickup by Frank Wallen brings us *'PSRs In Action: PHP League Event Package'*. Frank will explain the Event package and how it can easily handle events in your application. Read on to find out about installation and how this can work for you.

In our Artisan Way column, Steve McDougall is bringing you *'Controlling Quality in Laravel'* this month. Follow along as Steve gives you an inside look at the tools he uses to control code quality in writing Laravel applications. You'll learn about static analysis, code styling, and type coverage as various options for maintaining quality in your code.

Over in Education Station, Chris Tankersly has a good read for you, *'Generators for Efficient Code'*. Memory management is a common issue for developers and can definitely impact your efficiency. This is where PHP Generators come into play. Give this a read as Chris breaks down what exactly a generator is and what it does. He'll continue on to provide a little history of pre-generator time and the similarities and differences between generators and fibers.

Oscar Merida is delivering *'Quicksort'* in our PHP Puzzles column. Oscar left us with a challenge, as usual, this time to generate an array of N random numbers using Quicksort to order them from smallest to largest. Read on to see how Oscar solved this puzzle, and check out the latest challenge.

You'll want to check out Security Corner this month as Eric Mann is bringing us *'Demystifying Cryptography'*. You'll learn about symmetric and asymmetric encryptions. Additionally, Eric is going to break down envelope encryption, which is a hybrid between asymmetric and symmetric encryptions.

Christopher Miller is back in the column, Readable Code, with a great read, *'Is Your Code Ensapulated Enough To Be Clear?'*. Chris will take you through how to make your code clear, maintainable, and scaleable. All of this is possible with encapsulation, which Chris will outline by discussing the core principles, benefits, potential pitfalls, and real-world examples.

Our formerly DDD column, written by Ed Barnard, is now going to be called 2500 Feet where Ed will share his latest insights into how one of his new interests outside of coding has actually given him some useful insights into software development. This month, you'll get to check out Ed's first article, *'A View From 2500 Feet'*, in this new column. Ed is taking us with him on a personal journey of his latest chapter as a developer.

Maxwell Ivey shares some insight on forms this month in his Barrier-Free Bytes column with the article, *'Creating Accessible Forms'*. Max is giving us a great description of his experience utilizing various forms on websites. He goes on to outline some straightforward ways that improvements can be made, ultimately benefiting the user and developer in the end.

In our Finally column, Beth Tucker Long discusses user groups in *'Sunsetting User Groups'*.

Write For Us

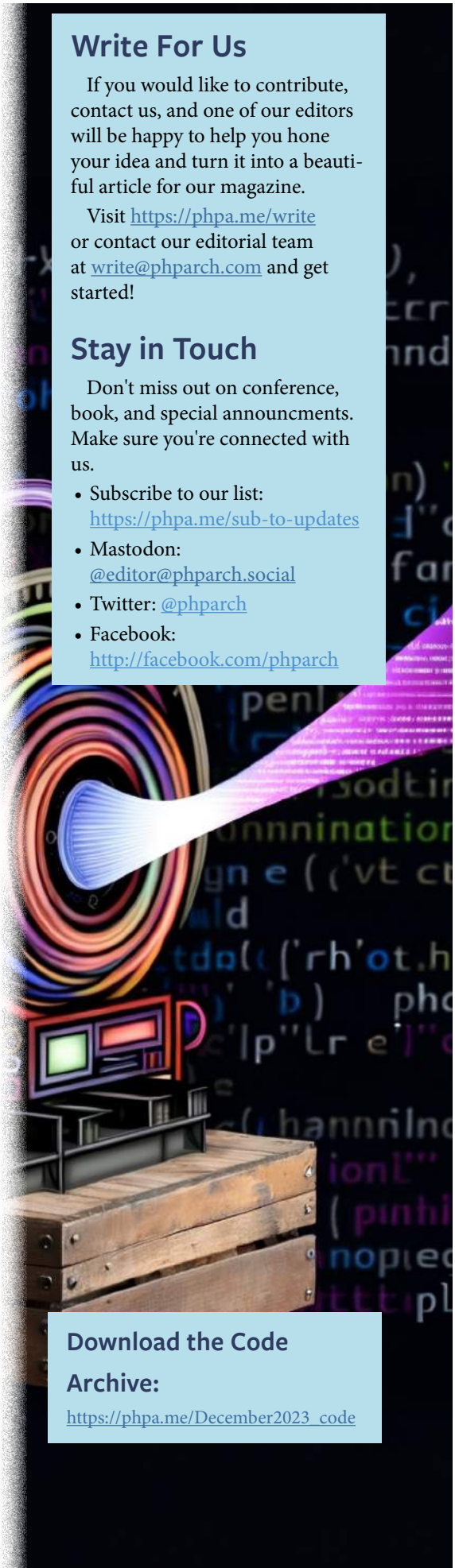
If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

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

Stay in Touch

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

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



Download the Code Archive:

https://phpa.me/December2023_code

Feature: PHP Unit Testing: Ensuring Code Quality and Reliability

Godstime Aburu

Welcome to your guide to becoming proficient in PHP unit testing! We're here to take you on a journey, starting with the basics and why they matter in the PHP development world. Along the way, you'll meet our trusty companion, PHPUnit, a well-loved unit testing framework.

As we delve into this article, you'll gain the skills to create unit tests and learn to embrace the power of Test Driven Development (TDD). We'll explore techniques like mocking, gracefully handling exceptions, and the wonders of parameterized testing. By the time you finish reading, you'll fully grasp the significance of unit testing, and we hope you'll be inspired to make it a cornerstone of your PHP development endeavors.

But that's not all! We'll even offer you a glimpse into the exciting future of unit testing in the realm of PHP development. Consider this your roadmap to coding with confidence and constructing stellar PHP applications. Let's embark on this journey together, one step at a time, and bring out the best in your PHP coding skills.

Introduction

Welcome to the ever-evolving world of web development, where PHP scripts fuel the websites that are part of our daily lives. In this dynamic landscape, a subtle yet powerful advantage sets experts apart from the rest – the reliability of their code. You might wonder why this reliability is so crucial. Well, join us as we dive into the world of PHP development and explore why code reliability matters. Discover the pivotal role that unit testing plays and get a sneak peek of what this article holds for you.

Imagine a world where your PHP-powered web application runs flawlessly, impresses users, and never falters. Sounds fantastic, right? But what happens when the unexpected occurs? Users become frustrated when errors pop up or their beloved web pages crash. In such situations, your application's reputation may suffer, and users might start searching for more dependable alternatives.

Code reliability is the cornerstone of consistent functionality. Think of it as the solid foundation of a house – without it, the entire structure could crumble. In web development, code reliability means having software that performs consistently under various conditions and during user interactions.

Think of it as the difference between a shaky bridge and a robust, well-built one. The shaky bridge might hold up most of the time, but you'd rather trust your journey to the solid one.

But we're not stopping there. Join us as we embark on a journey to uncover the hidden gem of web development – unit testing. It's like having a vigilant team of experts scrutinizing your code, ensuring it performs flawlessly.

Stay with us as we reveal the secrets to crafting effective unit tests, strategies for organizing your tests, and best practices for creating insightful test cases. We'll show you where to find testable sections in your PHP code, and we'll even introduce you to the concept of Test-Driven Development (TDD) using PHPUnit.

As we progress, we'll explore advanced unit testing techniques like mocking and stubbing to simulate and isolate dependencies, gracefully handling exceptions, and maximizing parameterized testing with data providers.

So, fellow PHP developer, fasten your seatbelt! By the end of this article, you'll be armed with the skills and resources to enhance the reliability of your PHP code. Together, let's embark on this journey, one unit test at a time.

Significance of Code Reliability in PHP Development

Picture this: You've crafted an amazing web application using PHP. It's a beauty, working flawlessly most of the time, sporting a sleek design, and blazing fast. But what happens when the unexpected strikes? Users get frustrated when encountering those pesky errors or witnessing their beloved pages crashing. Users might start hunting for a more dependable alternative if your app's reputation takes a hit.

That's where the magic word comes in – reliability. Ensuring the steadfast functionality of your PHP code is the golden ticket. It's like building a house with a rock-solid foundation; without it, the whole structure could crumble. In the world of web development, code reliability means having software that stays dependable no matter the circumstances or user interactions.

Think of it as the contrast between a creaky, unstable bridge and a well-engineered, robust one. You wouldn't risk your life on the wobbly one, right? Similarly, your users won't trust your PHP application if it's unreliable. Code reliability is the bedrock of user trust.

Role of Unit Testing in Ensuring Reliable Code

Now that we've emphasized the importance of reliable code, you might be wondering how to achieve it. This is where unit testing steps in with its magical toolkit to safeguard your PHP code.

Think of unit testing as your team of vigilant guardians, closely watching over your code's every move. It involves breaking down your code into manageable pieces and conducting thorough tests on each part. But this isn't just any testing; it's an in-depth examination of each code component to ensure it's doing its job flawlessly.

Imagine it as if you were baking a delicious cake. Unit testing is like checking that all your ingredients are fresh, that the oven is preheated to perfection, and that the cake rises just the way it should. This level of attention to detail ensures that your end product, in this case, your PHP application, performs as reliably as you intended.

Unit testing is a proactive approach, not something you do after writing your code. You can uncover issues early in the development process by writing tests in parallel with your code (or even before!). By tackling bugs in their infancy, you prevent them from growing into major headaches that could negatively impact your users' experience.

Overview of the Article's Content

Now that we've explored the vital concepts of code dependability and unit testing, let's dive deeper into the treasures hidden within this article.

First, we're laying the foundation by delving into the essentials of unit testing. You'll learn the heart of unit testing in PHP programming, what it's all about, and how to wield PHPUnit, a beloved framework for PHP unit testing.

But that's just the beginning! We're about to unlock the art of crafting effective unit tests. Not only will we provide you with smart strategies for organizing your tests and revealing the best practices for creating insightful test cases, but we'll also guide you to the gold mines within your PHP code where testing opportunities await.

But wait, there's more! We'll take you on a journey into the world of Test-Driven Development (TDD) using PHPUnit, where you'll witness how this approach can work magic, completely transforming your PHP development process.

We'll delve into more advanced unit testing strategies as we venture further. Get ready for a masterclass in using mocking and stubbing to simulate and isolate dependencies, gracefully handling exceptions, and unlocking the full potential of parameterized testing with data providers.

So fasten your seatbelt, fellow PHP developer! After you've devoured this article, you'll possess the skills and resources to enhance the dependability of your PHP code. Together, let's embark on this exciting journey, one unit test at a time.

Fundamentals of Unit Testing

Now, let's take a step back and revisit the fundamentals of our discussion about the critical role of code reliability and how unit testing is the unsung hero of the tale. In this section, we will break down the concept of unit testing and explore why it's a pivotal player.

Think of unit testing as the hero's introduction in this grand adventure. It's our trusty companion on this journey through PHP development. But what is it, and why is it such a big deal?

Unit testing is like the guiding light, illuminating your path in the world of PHP development. It's all about testing individual parts of your code to ensure they function as expected. It's not just testing; it's like a magnifying glass examining every detail to ensure everything runs smoothly.

Now, let's talk about the perks! Unit testing brings a host of benefits to the world of PHP development. It's not just about finding bugs; it's about preventing them from taking root in the first place. It's like having your PHP application wear a suit of armor to ward off potential issues.

And here's where our main character, PHPUnit, takes the spotlight. PHPUnit is your loyal sidekick in this journey, a beloved framework for PHP unit testing. It's the tool that'll be right by your side as you embark on your quest to create reliable, top-notch PHP code.

Definition and Purpose of Unit Testing

To put it simply, unit testing is like a chef carefully tasting each ingredient before adding it to the pot. It's the art of examining the tiniest building blocks of your code in isolation—those individual units or components. Each unit gets its own solo performance to make sure it's doing its job just right.

Imagine you're constructing a complex machine with a multitude of intricate parts. In this scenario, unit testing would involve testing each gear, wire, and circuit on its own to ensure that they operate flawlessly. This meticulous examination guarantees that every piece of your code is in perfect harmony.

Now, you might be wondering, why all this effort? Well, the goal of unit testing is refreshingly straightforward: it's all about catching those pesky bugs while they're still small and manageable. It's like shining a bright light on problems when they're lurking in the shadows rather than waiting for them to sneak up and wreak havoc on your application.

In essence, unit tests are the vigilant bodyguards of your code. They serve as a protective safety net, shielding your PHP application from regressions (those unintended side effects that can creep in with changes) and ensuring that your existing functionality remains intact as you introduce updates or new features. The end result? Code that's sturdy, reliable, and a breeze to maintain.

Benefits of Incorporating Unit Tests in PHP Development

Now that you understand what unit testing is all about, let's talk about why you should embrace it in your PHP development journey. The benefits are worth the effort, and here's why:

1. **Bug Prevention:** Unit tests are like your trusty bug detectors. They catch those pesky critters

early, preventing them from multiplying and causing chaos in your code.

2. **Improved Code Quality:** Crafting unit tests is like a friendly nudge towards writing cleaner, modular code. It's like a guardian angel for good coding practices, making your codebase clearer and a breeze to maintain.
3. **Confidence in Changes:** With those trusty unit tests in your corner, you can make changes to your code with peace of mind, knowing they'll be there to catch any unexpected surprises.
4. **Documentation:** Think of your code's unit tests as a living guidebook. They lay out how each piece should work, making it a breeze for you and your team to grasp and utilize your code.
5. **Time Savings:** While it might seem like a time investment upfront, writing tests often saves you loads of time in the long haul. It's like a shortcut that trims down the hours spent on debugging and hunting for problems.

Introduction to PHPUnit As a Popular Unit Testing Framework

Meet your trusty companion, PHPUnit, now that you're convinced of the benefits of unit testing. In the world of PHP unit testing, this is where the real magic unfolds.

PHPUnit, the go-to unit testing framework for PHP, is a real game-changer. It simplifies the creation and execution of unit tests with its array of powerful features and tools. Everything you need to ensure your PHP code is top-notch is right here in PHPUnit, which works just like a well-equipped test lab.

Now, let's dive into the world of PHPUnit with a quick example. Imagine you have a simple PHP function that can add two numbers: (See Listing 1)

Listing 1.

```php
function add($a, $b) {
    return $a + $b;
}

use PHPUnit\Framework\TestCase;
class MathTest extends TestCase {
    public function testAdd() {
        $result = add(2, 3);
        $this->assertEquals(5, $result);
    }
}
```

In this example, we take a step into the world of PHPUnit. We extend the `TestCase` class provided by PHPUnit, define a test method (`testAdd`) that calls our `add` function, and employ an assertion (`assertEquals`) to ensure the result meets our expectations. If the result doesn't align with our expectations, PHPUnit kindly raises a flag, signaling an issue in our code.

PHPUnit simplifies the whole process of crafting and running tests, making it a priceless addition to your PHP development toolbox. In the upcoming sections, we'll dive even deeper into the fantastic world of PHPUnit, uncovering its features and how it can be your best friend in the world of unit testing.

With the basics under your belt, you're well on your way to becoming a PHP unit testing pro. In the next part, we'll explore the art of creating effective unit tests, helping you hone your skills and create a solid testing arsenal for your PHP code.

Writing Effective Unit Tests

Welcome to the core of unit testing in PHP development. In this section, we're about to uncover the secrets of crafting powerful unit tests. We'll journey into the world of identifying those testable gems within your PHP code, chat about smart tactics for structuring and managing your tests, and dive headfirst into the realm of crafting meaningful test cases.

Identifying Testable Units in PHP Code

Before we dive into the world of crafting effective unit tests, let's get to the heart of what you're testing. In PHP, the gems you want to test are typically individual functions or methods within classes. These units should be like bite-sized nuggets, compact, self-contained, and handling a specific job.

Imagine you have a PHP class that's all about calculating the grand total of items in a shopping cart. In this scenario, a testable unit could be one of the methods inside that class, the one dedicated to calculating the subtotal. It's a focused piece of the puzzle, a little nugget of functionality you can extract and examine on its own.

Here's a simple example to illustrate the point: (See Listing 2)

Listing 2.

```php
class ShoppingCart {
    public function calculateSubtotal($items) {
        $subtotal = 0;
        foreach ($items as $item) {
            // Assuming each item has a
            // 'price' and 'quantity' property
            $subtotal += $item['price'] *
                         $item['quantity'];
        }
        return $subtotal;
    }
}
```

After pinpointing those testable gems, it's time to roll up your sleeves and create tests that put them through their

paces. This laser-focused strategy guarantees that every nook and cranny of your codebase does its job as intended.

Strategies for Structuring and Organizing Unit Tests

Now that you're aware of what's on your testing agenda, let's delve into the art of structuring and organizing your unit tests with finesse. A well-organized test suite isn't just a neat and tidy affair; it's your secret weapon for hassle-free management and maintenance, turning your testing suite into an invaluable ally instead of a cumbersome chore.

1. **Test File Structure:** Think of your test structure like a mirror reflecting your application's codebase. When it comes to organizing your test files, it's a smart move to keep them right alongside the code they scrutinize. You can keep things tidy and make life simpler by following a convention, such as naming a test file `ClassNameTest.php` for a class called `ClassName`.
2. **Test Classes:** Bring some order to the party by organizing your tests into neat classes, each one aimed at a particular component or job. PHPUnit lends a helping hand in this task with its trusty `TestCase` class, which you can use as a foundation for your test classes.
3. **Test Methods:** Inside your test classes, keep it simple with one method per test case. Give these methods meaningful names that tell the story of what they're putting to the test.

Here's a nifty example of a tidy test class: (See Listing 3)

Listing 3.

```
1. use PHPUnit\Framework\TestCase;
2. class ShoppingCartTest extends TestCase {
3.     public function testCalculateSubtotal() {
4.         // Create an instance of ShoppingCart
5.         $cart = new ShoppingCart();
6.         // Define a sample array of items
7.         $items = [
8.             ['price' => 10, 'quantity' => 2], // $20
9.             ['price' => 5, 'quantity' => 4],  // $20
10.            ['price' => 15, 'quantity' => 1], // $15
11.        ];
12.        // Calculate the expected subtotal
13.        $expected = 20 + 20 + 15;
14.        // Call the calculateSubtotal method
15.        //     and assert the result
16.        $subtotal = $cart->calculateSubtotal($items);
17.        // Assert that the calculated subtotal matches
18.        //     the expected subtotal
19.        $this->assertEquals($expected, $subtotal);
20.    }
21. }
```

By sticking to these tactics, you're building a well-structured and rational testing suite that becomes your trusty ally in spotting and tackling issues when they pop up.

Best Practices for Creating Meaningful Test Cases

Now, let's roll up our sleeves and explore the fine details of crafting test cases that truly matter. Think of these effective test cases as your trusty map, leading you to the heart of any issue lurking in your code. Here are some golden best practices to keep in mind:

1. **Clear Test Names:** When it comes to naming your test cases, be crystal clear and descriptive. A strong test name should spell out exactly what the test is up to and what it's on the lookout for. For instance, `testCalculateSubtotalWithMultipleItems`tells a much better story than just `testSubtotal`.
2. **Arrange-Act-Assert (AAA) Pattern:** When you're crafting your tests, follow a simple pattern called 'AAA.' First, set the stage by getting everything ready (like setting up objects or data). Then, get into action by running the code you're examining. Finally, check the results against what you expected.
3. **Use Assertions:** PHPUnit has your back with a toolbox of assertion methods tailored for different types of tests. When you're in the arena, pick the right tool for the job. For instance, reach for `assertEquals` when checking if the expected and actual values match.
4. **Test Boundary Conditions:** Don't limit your tests to everyday scenarios; go the extra mile and explore the boundaries and the edges. What unfolds when the input is as empty as a blank canvas or stretches to its absolute maximum? These tests shine a light on potential hiccups that might stay hidden in everyday routines.
5. **Isolate Dependencies:** 1. If your code relies on outside resources or services, think about bringing in some 'mocks' or 'stubs' to create a barrier around your tests. PHPUnit comes to the rescue with nifty features for this, letting you isolate your tests and keep your focus laser-sharp on the code unit you're examining.

Let's check out a real-world example of a test case that packs a punch using PHPUnit: (See Listing 4)

Listing 4.

```
1. public function testCalculateWithMultipleItems() {
2.     $cart = new ShoppingCart();
3.     $items = [
4.         ['name' => 'Item A', 'price' => 10],
5.         ['name' => 'Item B', 'price' => 15],
6.     ];
7.     $subtotal = $cart->calculateSubtotal($items);
8.     $this->assertEquals(25, $subtotal);
9. }
```

In this test case, we're setting the stage with the cart and items, taking action by crunching the numbers for the subtotal, and ensuring it lines up with our expected result.

By sticking to these tried-and-true practices, you're creating effective unit tests and building valuable tools that keep your PHP code's reliability on point. In the next section, we'll dive headfirst into Test-Driven Development (TDD) and uncover how this approach can be a game-changer for your PHP development journey.

Test-driven Development (TDD)

Step right into the world of Test-Driven Development (TDD), a revolutionary method in the realm of PHP development. In this section, we'll unravel the magic of TDD, reveal its countless advantages in the PHP universe, and hand you a step-by-step guide on how to put TDD into action with PHPUnit.

Understanding the TDD Approach

TDD might have a fancy ring to it, but it's actually a simple yet powerful concept. In a nutshell, TDD is a method where you start by crafting tests for your code even before you put a single line of code in place. It's like sketching out a map before setting off on an adventure.

The TDD cycle usually follows three easy steps:

1. **Write a Test:** 1. Kick things off by painting a picture of the functionality you aim to build with a test case. At first, this test will raise a red flag because you still need to write the code to make it work.
2. **Write the Code:** Now comes the exciting part: put your coding hat on and start crafting the magic that turns that red flag into a green one. Your mission is to whip up just enough code to meet the test's demands.
3. **Refactor:** After your test gets the green light, it's like having a canvas ready for you to refine the artwork. You can tweak the code's structure, make it more legible, or boost its performance without any worries about breaking things—thanks to your trusty tests that guarantee its correctness.

At first glance, this approach might seem a bit backward. Why in the world would you write tests for code that hasn't even been born yet? Well, the enchantment of TDD lies in its knack for pushing you to dive deep into what your code should achieve and how it should act even before you start slinging code around. Think of it as sketching out the plot of a story before embarking on that epic novel; you've got a crystal-clear vision of where it's all headed.

Benefits of Practicing TDD in PHP Development

Now, let's unveil the reasons behind TDD's surge in popularity among PHP developers:

1. **Faster Debugging:** TDD often lends a helping hand by nabbing those bugs at an early stage, which means less time down the road spent on hair-pulling debugging.
2. **Improved Code Quality:** TDD pushes you to kickstart your code journey with cleanliness and modularity in mind. The payoff? You have a codebase that's not only more reliable but also easier to keep in tip-top shape.
3. **Enhanced Collaboration:** Tests pull double duty as a kind of documentation, smoothing the path for team members to grasp the code and join forces on it.
4. **Confidence in Changes:** Whenever it's time for updates or fresh features, you can dive in with peace of mind, thanks to those trusty tests standing guard and ready to sound the alarm if anything goes awry.
5. **Better Design:** TDD has a knack for steering you towards beautifully structured code. It gently nudges you into crafting code that's all about owning a single responsibility and following the SOLID principles like a pro.

Step-by-step Guide to Implementing TDD with PHPUnit

Now, let's roll up our sleeves and get hands-on with TDD using PHPUnit. We're taking a practical dive, step by step, with a straightforward example: a function that does the nifty job of checking if a number is even.

Step 1: Write the Test

Begin by spinning the yarn with a test that sets the stage for what you expect from your code. Craft a test class and method, and give them a name that paints a picture of what you're putting to the test. (See Listing 5)

Listing 5.

```
1. use PHPUnit\Framework\TestCase;
2. class IsEvenTest extends TestCase {
3.     public function testIsEvenForEvenNumbers() {
4.         // Test for even numbers
5.         $this->assertTrue(isEven(2));
6.         $this->assertTrue(isEven(4));
7.         $this->assertTrue(isEven(100));
8.     }
9. }
```

Step 2: Run the Test (It Will Fail)

Now, hit the stage where you run your test. Brace yourself because it's supposed to raise a red flag since you haven't whipped up the code just yet. This step is a biggie because it's the moment of truth to make sure your test is doing its job.

Step 3: Write the Code to Make the Test Pass

Now, get your coding fingers moving and bring that test to a happy green state. In our case, we're crafting a function called `isEven`. (See Listing 6 on the next page)

Listing 6.

```
1. use PHPUnit\Framework\TestCase;
2. class IsEvenTest extends TestCase {
3.     public function
4.             testIsEvenReturnsTrueForEvenNumbers() {
5.         $this->assertTrue(isEven(2));
6.         $this->assertTrue(isEven(4));
7.         $this->assertTrue(isEven(100));
8.     }
9.
10.    public function
11.            testIsEvenReturnsFalseForOddNumbers() {
12.        $this->assertFalse(isEven(1));
13.        $this->assertFalse(isEven(3));
14.        $this->assertFalse(isEven(99));
15.    }
16.
17.    public function
18.            testIsEvenReturnsTrueForZero() {
19.        $this->assertTrue(isEven(0));
20.    }
21.
22.    public function
23.      testIsEvenReturnsFalseForNegativeEvenNumbers() {
24.        $this->assertFalse(isEven(-2));
25.        $this->assertFalse(isEven(-4));
26.    }
27.
28.    public function
29.      testIsEvenReturnsFalseForNegativeOddNumbers() {
30.        $this->assertFalse(isEven(-1));
31.        $this->assertFalse(isEven(-3));
32.    }
33. }
```

Step 4: Rerun the Test (It Should Pass)

Once your code is in place, hit that test once again. This time, you should see the sweet sight of a passing test, confirming that your code is right on the money for the specific test case.

Step 5: Refactor (If Needed)

If you feel the need, give your code a little makeover for better structure, readability, or performance. The charm of TDD? You've got those trusty tests that act like a safety net, making sure your tweaks don't trigger any nasty surprises down the road.

This simple example paints the picture of the TDD cycle. You rinse and repeat this dance for every piece of functionality you want to sprinkle into your PHP codebase. As time goes on, you'll assemble a solid collection of tests that wrap your code in a cozy blanket of confidence.

By diving into TDD with PHPUnit, you'll experience firsthand the perks of this approach in PHP development. It's not just about scribbling tests; it's about sculpting code with a top-notch design that's easy to maintain and as resilient as a fortress.

In the next section, we're delving into the deep end, exploring some high-level tricks in unit testing, like the art of mocking and stubbing dependencies, gracefully handling exceptions, and unleashing the power of parameterized testing. These techniques are your ticket to leveling up your PHP unit testing game.

Advanced Techniques in Unit Testing

Now that you've got the basics down and journeyed through the realm of Test-Driven Development (TDD), it's time to kick it up a notch and turbocharge your PHP unit testing prowess. In this section, we're diving into some advanced tricks that will arm you with the skills to craft more thorough and potent tests. We'll navigate the world of mocking and stubbing dependencies, unravel the subtleties of dealing with exceptions, and roll out the red carpet for parameterized testing with data providers.

Mocking and Stubbing Dependencies for Isolated Testing

In the real world of PHP applications, code frequently leans on external buddies like databases, APIs, or services. It's crucial to mimic these buddies instead of reaching out and shaking hands with them directly to keep your unit tests clean and tidy. And that's where the magic of mocking and stubbing takes center stage.

Mocking: Picture this: Mocking is like conjuring up phantoms that behave just like the real deal. Thanks to PHPUnit's wizardry, you can whip up these phantom objects that act as if they were the genuine dependencies. The perk? It lets you keep the limelight on the code unit you're putting through its paces without getting sidetracked by the actual dependencies.

```
// Creating a mock object of a database dependency
$mockDatabase = $this->getMockBuilder(Database::class)
                     ->getMock();
// Define expectations for the mock
$mockDatabase->expects($this->once())
    ->method('query')
    ->willReturn($fakeData);
// Now, use $mockDatabase in your test
```

Stubbing: Stubbing? Think of it as a cousin of mocking. It's all about prepping dependencies to belt out scripted lines. For instance, you can teach an HTTP client to croon the same tune every time, giving you a fixed response during testing, no matter what mood the actual external service is in.

```
// Stubbing an HTTP client
$httpClient = $this->getMockBuilder(HttpClient::class)
                   ->getMock();
$httpClient->method('get')
           ->willReturn($fakeApiResponse);
// Now, use $httpClient in your test
```

Embracing the art of mocking and stubbing secures a cozy cocoon for your unit tests. No matter how wild the dependencies might get, your tests stay shielded and unswayed, like a reliable compass in a complex wilderness.

Handling Exceptions and Assertions in Unit Tests

In the real world, code can stumble upon some surprise twists that throw exceptions into the mix. When it comes to robust unit testing, it's not just about navigating the usual path; it's about being prepared to tackle these curveballs and exceptional cases.

Handling Exceptions: PHPUnit comes to the rescue with some nifty tools tailor-made for handling exceptions. You've got the `expectException` and `expectExceptionMessage` methods at your beck and call. Use them to lay down the law, specifying the exact exceptions your code should toss and the error messages they should carry.

```
public function testDivideByZeroThrowsException() {
    $this->expectException(DivisionByZeroError::class);
    $this->expectExceptionMessage(
        'Cannot divide by zero'
    );
    // Your code that should throw the exception
    Divide(10, 0);
}
```

Assertions with Exceptions: PHPUnit goes the extra mile by dishing out some snazzy assertions for handling exceptions and giving them a good once-over. Take, for example, the `assertThrows` move, your trusty sidekick for confirming that a particular exception takes the stage. (See Listing 7)

Listing 7.

```
1. public function testExceptionMessage() {
2.     $this->assertThrows(
3.         MyCustomException::class,
4.         function () {
5.             // Code that should throw MyCustomException
6.             //    with specific message
7.         },
8.         'Expected exception message'
9.     );
10. }
```

By giving exceptional cases a run for their money and mastering the art of handling exceptions, you're building a safety net that lets your code gracefully bounce back from hiccups and serve up helpful error messages when the road gets rocky.

Utilizing Data Providers for Parameterized Testing

Let's face it: penning separate test cases for every possible input scenario can feel like a never-ending marathon. That's where parameterized testing and its trusty sidekick, data providers, sweep in to save the day. They let you take that same test routine and run it through different input data sets, making your tests streamlined and easy to manage. Here's how it works: (See Listing 8)

Take a peek at this example: the `additionProvider` method steps in with input data and expected outcomes for the `testAddition` test case. Then, like a diligent worker, PHPUnit takes the reins, running the test for each dataset without breaking a sweat. It's like testing a smorgasbord of scenarios with minimal fuss.

Parameterized testing is your secret weapon to streamline your test suite, ensuring your code gets the thorough once-over it deserves. It's the muscle behind code reliability.

With these advanced tricks up your sleeve, you're primed to tackle the wild world of complex scenarios and those sneaky edge cases in your PHP unit tests. In the final section, we'll circle back to the importance of unit testing, urging you to make it your trusty sidekick in PHP development, all while peering into the crystal ball for what lies ahead.

Conclusion

Give yourself a pat on the back for this fantastic voyage into the realm of PHP unit testing! As we close this chapter, let's take a moment to revisit why unit testing is such a crucial ally. It's the unsung hero in your PHP development toolkit, ensuring your code stands strong. So, let's make it a daily ritual, a steadfast practice in your PHP development journey.

And what lies ahead in the future of unit testing in the PHP world? Well, the path is ever-evolving, but one thing's for sure – it's an exciting road filled with opportunities to elevate your code's reliability and your development skills.

Recap of the Importance of Unit Testing for Reliable PHP Code

Throughout our journey in this article, we've placed a spotlight on the pivotal role of unit testing in safeguarding the trustworthiness of your PHP code. Let's take a moment to underline why it's such a game-changer:

Unit testing, my friend, is the sentry at the gate of code reliability. It's your vigilant protector against the onslaught of bugs, regressions, and those sneaky issues that can sneak into your PHP applications. By scrutinizing each building block in isolation, you're like a detective catching culprits red-handed early in the game when they're mere mischief-makers.

Listing 8.

```
1. /** * @dataProvider additionProvider */
2. public function testAddition($a, $b, $expectedResult) {
3.     $result = add($a, $b);
4.     $this->assertEquals($expectedResult, $result);
5. }
6.
7. public function additionProvider() {
8.     return [
9.         [1, 2, 3],
10.        [0, 0, 0],
11.        [-1, 1, 0],
12.        [10, -5, 5],
13.    ];
14. }
```

Picture your PHP code as a vast tapestry of intricate threads. Unit testing lets you thread that needle one strand at a time, ensuring each thread is robust and dependable. Without these tests, your code could resemble a house of cards, teetering on the edge of fragility, ready to crumble at the gentlest breeze.

The importance of code reliability is paramount—no exaggeration there. It's the bedrock upon which the trust of your users, the stability of your application, and the ease of its upkeep are built. Reliable code paves the way for a smooth and untroubled user experience, free from unexpected hiccups and disturbances.

Encouragement to Adopt Unit Testing As a Standard Practice

Now that you've witnessed the magic of unit testing, we implore you to welcome it with open arms as an indispensable part of your PHP development voyage. It's not a frivolity; it's a must-have. Here's why it deserves a prime spot in your development routine:

1. **Quality Assurance:** Unit testing raises the bar for your code quality. It's the shield that guards against the pitfalls and slip-ups that can plague your code, giving you the confidence that your code always performs as intended.
2. **Time Savings:** Though crafting tests might appear to be a time investment at first, it's a wise time-saving move in the grand scheme. Spotting and resolving bugs early on is a far more efficient approach than the wild chase to catch elusive issues later in the development journey.
3. **Collaboration:** Think of your unit tests as a living, breathing documentation for your code, simplifying the understanding, collaboration, and effective contributions of your team members. It's your code's tour guide, always ready to show the way.
4. **Confidence:** Unit tests create a safety net, a reassuring cushion that lets you venture into changes, add new features, or embark on refactoring journeys with unwavering confidence. Those tests are the trusty safety harness, ensuring you stay on course and catch any unexpected twists along the way.
5. **Professionalism:** In the dynamic landscape of PHP development, unit testing distinguishes you as a true professional, marking your commitment to delivering rock-solid, dependable software. It's your badge of honor in the world of code craftsmanship.

Unit testing isn't reserved solely for big development teams or intricate projects. Whether you're a lone coder or part of a massive organization, crafting a modest script or a sprawling application, unit testing can substantially impact your development journey and the caliber of your code. It's the universal tool that fits in every developer's toolbox.

Final Thoughts On the Future of Unit Testing in PHP Development

As we cast our eyes on the horizon of PHP development, it's evident that unit testing will maintain its central role. Here are some parting reflections:

1. **Integration with Modern Tools:** PHPUnit and its testing counterparts are poised to evolve, seamlessly aligning with contemporary development tools and practices. Expect improved integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines, streamlining the testing process and enhancing code reliability. The future is all about harmony and efficiency.
2. **Greater Focus on Test Automation:** Automation will be the cornerstone. In the era of DevOps and automated testing, anticipate the emergence of more advanced testing automation tools and practices. The future is all about streamlining and efficiency, with testing leading the way.
3. **Community Support:** The PHP community has a rich heritage of knowledge-sharing and tool-building. As unit testing continues to gain momentum, brace yourself for a treasure trove of resources and libraries to support you on your journey. Together, the community thrives and advances.
4. **Shift Toward Test-Driven Development:** Test-Driven Development (TDD), as we've delved into, is poised for broader acceptance as developers experience its tangible advantages. It's a paradigm shift that can pave the way for more resilient and trustworthy PHP applications. The future is bright for TDD enthusiasts.

In conclusion, unit testing isn't a fleeting trend; it's an enduring practice that elevates the maturity and professionalism of PHP development. By embracing unit testing, you're making a valuable investment in the durability, stability, and success of your PHP projects. As you forge ahead in the realm of PHP development, keep in mind that every test you craft, every bug you intercept, and every regression you avert bolsters the reliability and dependability of your code. It's the path to code you can truly rely on.

References

Throughout this article, we've gathered insights and knowledge from a multitude of sources and references to offer you a comprehensive guide to PHP unit testing. Below is a list of these references, where you can dive deeper into the realm of unit testing:

1. **PHPUnit Documentation:** The official documentation for PHPUnit is a valuable resource for information on writing and executing unit tests in

PHP. You can access the latest documentation here[1]. It's an indispensable guide for all your PHPUnit needs.

2. **PHP Manual:** The PHP Manual is an essential resource for gaining a deep understanding of PHP, covering functions, classes, and core concepts. It's the definitive reference for all things PHP. You can access it online here[2]. Whether you're a novice or an expert, it's an invaluable companion in your PHP journey.
3. **Clean Code:** "A Handbook of Agile Software Craftsmanship" by Robert C. Martin is a highly recommended book that underscores the significance of writing clean and maintainable code, a fundamental principle of effective unit testing. It's a must-read for any developer committed to code quality and craftsmanship.
4. **PHPUnit GitHub Repository:** PHPUnit is an open-source project, and its GitHub repository serves as a central hub for the most recent updates, discussions, issues, and contributions. You can explore the PHPUnit repository here[3]. It's a great place to stay informed about the latest developments and to engage with the PHPUnit community.
5. **Online Tutorials and Blogs:** Websites like Stack Overflow, Medium, and various personal blogs are valuable sources of tutorials and articles on PHP unit testing. They offer practical insights and solutions to common challenges, making them excellent resources for developers looking to enhance their unit testing skills and knowledge.

These references have played a crucial role in shaping the content of this article, but it's important to recognize that the world of PHP development is constantly evolving. New tools, techniques, and best practices emerge regularly. I encourage you to remain curious and continue exploring to expand your knowledge in the dynamic field of PHP unit testing.

As you navigate your PHP unit testing journey, keep in mind that learning is an ongoing process, and the collective wisdom of the PHP community is an invaluable resource. Whether you're writing your first unit test or fine-tuning an extensive testing suite, these references will be reliable companions on your quest for code reliability and excellence.

Happy coding and testing!

Godstime is a skilled technical writer with 4 years of experience specializing in PHP development. He has a deep understanding of PHP, including frameworks like Laravel and Symfony, database integration, and server-side scripting. He is passionate about translating complex concepts into accessible content and staying up-to-date with the latest advancements in the PHP community. His goal is to contribute valuable content that empowers developers to build robust and scalable PHP applications.

1 https://phpunit.de/documentation.html
2 https://www.php.net/manual/en/
3 https://github.com/sebastianbergmann/phpunit

The Great EpiPHPany

Eric Ranner

Okay, so here is the "skinny". I am a 35-year-old Junior Developer. My road to the programming world is unorthodox, to say the least. Unlike many in the industry, I had no idea that I had it in me to be a programmer until I turned 33 years old. You see, I had an entire career before hitting the "reboot" button on my work life. Although it may seem trivial to give you my "work-life story", I think it's important for context.

A Mid-30-year-old's Journey Into Coding

My career started in the Hospitality industry at the age of 7. Yes, don't tell Uncle Sam that… You see, I have two brothers and three sisters. My Dad worked 60-70 hours a week to provide for us, and my Mom had the harder job of being a stay-at-home mom for six crazy kids! Naturally, we were quite the handful in the summertime when school was out, so my Dad decided he would bring me to work with him. I learned the business, and I learned it well. When I was old enough, I even got a paycheck to go along with all of the business knowledge!

When I graduated high school, the natural next step was studying Hospitality Management at college. All throughout college, I continued working at the same resort, climbing my way up the ladder. Eventually, I was offered a management position, but I had to switch from being a full-time student part-time worker to a full-time worker and "no-time" student. I figured I was in school to do Hospitality management anyway, so why not just jump into the workforce?

Things were going really well. I excelled at the position and gained more and more responsibility until, eventually, I worked my way up to the Corporate Social Director. This meant that I was responsible for the selling, planning, and execution of Corporate Retreats, with a specialty in events. To make these events extra special, I began to make interactive games like "The Price Is Right" and "Wheel of Fortune", all in Microsoft PowerPoint!

Oof… It hurts to "say" that out loud. I remember the agony of using the Animation Pane and trying to figure out which trigger needed fire, which event, and what order it needed to happen. The PowerPoint files ended up being several gigabytes in size, and well, I began to crash the program. Eventually, I wanted to add a "drag and drop" feature to a game I was making. The game was called TrivEmoji, and I wanted to be able to let our guests go up to a touch screen and drag different emojis from a selection board into an answer board (i.e., category music: pizza slice + American flag = American Pie). I broke into the developer tab and was introduced to the world of "programming".

I fought the good fight with VBA until I realized I just had to be free from the program altogether. I needed to be able to create whatever I wanted, wherever I wanted, and whenever I wanted it. That is when I discovered HTML5, CSS, and JavaScript.

From Powerpoint to the Web Stack

Now, mind you, I had NEVER written a single line of code in my entire life prior to this. I spent 17 years in Hospitality, so I was completely clueless on how to get started. I turned to the internet, but to be honest with you… that didn't help; if anything, it made things more confusing. There was TOO MUCH information, mixed with strong opinions and fanboys (and girls) galore… simply put, a whole lot of noise. So, I had to start somewhere. Since the internet was a mixed bag of resources (some things way over my head as a beginner), I turned to books. The very first one I found was a book on basic HTML and CSS[1]. Honestly, this was FANTASTIC! If you are out there and looking to get into coding, this is a great place to start.

I devoured the book. I followed along with the examples and was even able to make a few trivia games within a few weeks! I was so proud of myself. "It takes years for people to learn this, and I did it in a few weeks," I thought to myself. Little did I know, I really wasn't "coding". I feel like the community is mixed on this… Are HTML and CSS coding languages? I tend to fall into the camp of saying… no. Sorry for those of you out there just getting started. I'm proud of you for dipping your toe in the water, but really, HTML and CSS are mostly glorified markup. This isn't meant to downplay them. They are extremely important and very powerful in their own right. I mean, nobody wants to look at a website that contains no CSS, and without HTML, PHP and JavaScript would not exist. So, you both have my love and respect, but coding languages you are not.

That being said, I turned to the internet again to see what the next step was. Remember, I wanted to make a drag-and-drop board for my emoji game. So naturally, drag and drop in the browser means you NEED JavaScript—no ifs, ands, or buts about it. If you want to work as a web developer, you're going to need some JavaScript at some point. It has a monopoly on the browser, so you better get used to it. Love it or hate it, it was my next step. This time, I thought, since I had so much success with a book last time, let's try that again! I turned

1 https://phpa.me/css-quickstart

to Jon Duckett's book on Javascript & jQuery[2]. I thought to myself, "Okay, HTML and CSS took me less than a month… I'll have this JavaScript thing figured out in 3 months!" How naive I was.

First, although the book I just mentioned is exceptional, I realized why many coders don't work out of books. The book was written in 2014, and at the time I was teaching myself, it was 2021. The book contained nothing about arrow functions, the fetch api, and certainly nothing about server-side JavaScript via Node.JS. But of course, me at the time didn't realize any of this. I began to make divs change colors and could make counters and scoreboards. I was feeling pretty good! Look at all I had taught myself. Gold star for me! Woot woot!

Reality Hits

Alright, a few months in, and I can whip up a static website, link pages, and make things move and light up on the screen. I've made it! I'm a Web Developer… except, I'm not. No, this begins the downfall. Every coder has experienced this at one time or another. The realization that you don't know what you don't know—and what you don't know is a lot. I can still remember creating entire web pages to house a single question for a trivia game… and there were hundreds of questions. I hate to admit it to you, but the files were named "Question1.html", "Answer1.html", "Question2.html", "Answer2.html", and so on. I was in-lining JavaScript in the HTML because I had no idea what I was actually doing. Some guy on some forum shared a snippet that did what I needed… Copy, Paste, and ba-zing; I got myself an interactive, horribly coded, and disgustingly organized website.

I still remember writing the lines of Javascript:

```
var question1 = document.getElementById("question1");
var q1Btn = document.getElementByID("question1Button");
var answer1 = '1958';
q1Btn.addEventListener('click', (e)=> {
    e.preventDefault();
    question1.textContent = answer1;
 });
```

The worst part would be… the next line of code would start with `var question2 = document.getElementById('question2')`… Rinse and repeat a hundred times.

So that works, and to be honest, all of the people playing the games were having a great time and commenting about how great the new games looked and how well they functioned, but I knew it was just smoke and mirrors. I knew that there needed to be a better way to do this. It just so happened a guest and former employee had caught wind of what we were trying to do and offered his services (of course, with some bartering involved). He was a former PHP programmer and wanted to have a little side project and thought it would be fun to take our games to the next level. He told me that PHP was the way to really accomplish what we wanted. He said that we would be able to have a dashboard where we could have anybody put in questions and answers, and the games would automatically update. Competitive by nature, I knew I needed to upgrade my abilities as well.

PHP Refusal & Mern Bootcamp

After hearing that he was going to make our games in PHP, I went right back to the internet and googled "PHP for web development". As any PHP developer knows… The hate was real. Forum after forum, YouTube vid after YouTube vid, "PHP is dead!" or "Ranking coding languages, PHP goes straight into the trash tier." So naturally, I rebelled. Even though I was able to easily connect to a database and start up a browser session, the internet gurus knew better than me, and I would not be using a dying technology.

At this point in my journey (especially after getting my first taste of Full Stack Development with PHP), I knew the only way forward was to go back… to the backend, that is. I turned to the online community again, seeing how my JavaScript book was a few years shy of a decade old, and found that many people learn Full Stack Web Development from Udemy Courses and Bootcamps. Seeing how the Udemy courses were significantly cheaper than a coding boot camp, I decided to try those first. After they fell short of my hopes, I decided that the major problem I was having was just too much information to sift through. Too many opinions, and especially in the JavaScript world, way too many libraries and packages.

Trying to learn Node.JS with absolutely no understanding of the backend and how it works is like having a pilot fix an airplane engine. Just because you know how to fly the plane doesn't mean you have any understanding of how it works. I decided that it was time to bite the bullet and take a bootcamp. Not because I felt I needed it to understand how to code but rather because I had such a hard time trying to sift through the noise I keep mentioning. I thought for sure that structure was all I needed… it would also look good on my resume to have some kind of certification. I enrolled in a nine-month Full Stack Web Development boot camp focusing on the MERN stack. At first, the course was so easy for me—things like variables and if statements were cake. I figured I would surely cruise through this class and be a certified web developer in no time.

As the course went on, I remember all I could think about was how difficult this experience must be for absolute beginners. It progressed so fast that I eventually understood that the point of these bootcamps is not necessarily to make you an expert coder. Bootcamps, more or less, attempt to expose you to as many things as possible so you can get some level of familiarity with all of the things you will be doing daily as a Full Stack (in my case) JavaScript developer.

About halfway through the boot camp, I got my first job as a Software Developer! Now, it wasn't in the web stack (VBA… far from it), but it got me out of hospitality and into the right industry. The knowledge I have gained as a junior developer at my current company has made me a better coder than I could have ever imagined. It has also humbled me, tested me, and taught me the most valuable lesson that every coder has to learn: coders will NEVER KNOW IT ALL. It is simply

2 https://phpa.me/jquery

impossible and foolish to think you will be able to master all of the different coding languages and technologies out there. Instead, you need to find what problem you want to solve and use the best tools for the job, which brings us back to PHP.

PHP Adoption and Adoration

The job that I currently have is for a very small shop. It has its benefits and drawbacks (this is a topic for an article all on its own), but one of the benefits is you are exposed to so many technologies. If somebody hires us, we will find a way to make it work. Within the first two months at my company, a project came in to design and revamp a 100-question form submission system. Out of the entire company (like 4 of us), I was the one with the most web experience, so it fell to me. There was one problem, though… the client's codebase was in dirty old PHP…

Never one to back down from learning something new, I enthusiastically jumped and began to hammer through the PHP basics. Ah, ok, this is a loop… Variables are declared with a "$" interesting…. Eventually, finding the syntax was rather easy to pick up, especially since I had some exposure to it already and, of course, my JavaScript knowledge. Once I had the confidence, I felt like I was ready to jump into the codebase. WHAT A MESS! I was looking at ONE FILE with 10,000 lines of code! I remember thinking… this HAS to be why people hate PHP!

As I continued looking at the codebase, I noticed something very different from JavaScript… "Wait! PHP can integrate directly with HTML?" I said out loud to myself. "No EJS or Handlebars needed?" "Wait! To get data from a form, all I need to do is access the `$_POST` array? I don't have to grab the elements by id, give them event listeners, grab their values, parse the json, send them to an api, etc.?"

Then came the realization that all I needed to do was install the latest version of PHP, give a file a ".php" extension, put in my `<?php` tags, and all of this functionality was at my fingertips. No, I don't have to install Node.JS (to be fair, I did have to install XAMPP)! No, I don't have to run "npm install express" and what feels like 50 other libraries to do basic routing. And this next one was beautiful… No, I don't need to generate a JSON web token, sign it, return it, and manage state for a basic login system… I simply had to write one line of code… "session_start();" and a good amount of that authentication functionality was available (obviously, managing the database and assigning session variables came later, but this approach was much more beginner-friendly than server-side JavaScript's).

Over the next month, I discovered that PHP had a way to do everything I could ever want to do while coding a website and more. I even began to venture into OOP! The language is flexible enough to support procedural or object oriented programming styles. What wasn't to like? It is used in over 70% of websites and is the language that backs the most popular CMSs in the world. If you want to be a web developer, why wouldn't you want to know PHP?

From MERN to MAMP

As the months went by, I discovered not only did I like PHP, but I LOVED PHP and preferred it over any other language or web development tool. I even wrote some back-of-house programs in PHP to help migrate and clean up our in-house repositories. Don't get me wrong, I will always need JavaScript, and it holds a special place in my heart for being my first REAL coding language, but when it comes to backend web development, PHP is king.

If I could go back in time to before I wrote my first line of code and have one piece of advice I could share with my past self, it would be, "Learn PHP!" If I had known that back then, I would be so much farther in my development and understanding of not only web development but software development and computer science in general. Most coders recommend JavaScript or Python as a first coding language. Although I can see where they are coming from (JavaScript is very visual when you manipulate it, and Python reads almost like English), I wholeheartedly disagree. I think the case should be made more often for PHP as a first language, especially if your focus is web development.

- It has so many great features baked into the language and a friction-free integration with HTML.
- WordPress and other popular CMSs are built in PHP, so you can get an easy entry into working with existing codebases.
- Utilizes HTTP to help with understanding how data is sent over the internet

All I can say is that as a 35-year-old junior developer, time is the most important thing that I don't have on my side. I need to learn web development quickly and efficiently, and PHP is the best tool for the job. It is tried and true and will be around for many years to come. PHP is alive and well. This is my Great EpiPHPany!

Eric Ranner is a Junior Software Developer at Grandjean & Braverman, Inc. He is passionate about web development and all things PHP. He is the owner and lead developer of EJ Web Design in Honesdale, PA. He is also the lead developer and main consultant of BeesNestDesign.com.

Demystifying Cryptography

Eric Mann

One of the more advanced topics handled by modern developers is cryptography. It's the stuff of science fiction to many, but frankly, it doesn't have to be a mystery to any of us.

We shared roughly the same course schedule and had many of the same professors and textbooks. That made it easy for Ben and I to share secret messages via our college-monitored email. Rather than sending notes in the clear, we'd merely reference a professor and a page number, then append a fully encrypted message in the rest of the message.

The professor reference would direct each of us to the primary textbook for the class. We'd then turn to the correct page and use the first sentence on that page as an encryption key for our messages. It wasn't the most secure of communication methods, but it was entertaining to the two of us nonetheless.

Symmetric Encryption

This is also a great example of one of the ways cryptography works in the real world. We had a trusted means of communication—our class schedule and an in-person agreement on how to arrive at encryption keys. This allowed us to leverage the same secret key for both encryption and decryption. So long as no one else figured out how we agreed on keys, our messages were safe from prying eyes.

Not that rants about the coursework we didn't like were the most critical of messages in the first place.

At its core, symmetric encryption is one of the most common forms of cryptography you'll directly encounter within software. Your machine is likely protected by full-disk encryption, which leverages a symmetric encryption key (likely tied to your password) to encrypt and decrypt the primary hard drive. Your mobile devices leverage similar mechanisms. Servers and databases hosted by cloud providers like AWS use the same methods and algorithms as well.

In each of these cases, the encryption scheme relies on some secret information shared between the parties involved through some other secure means. This could be a textbook on a shelf, a passphrase conveyed out-of-band between participants, or a password you keep to yourself. In any case, the algorithms involved are strong, secure, and well-documented. It's the security of the means of exchanging the "pre-shared secret key" that makes or breaks the overall safety of the system.

Asymmetric Encryption

In some situations, you cannot trust another means of communication and have no way to convey a pre-shared secret. In those environments, you might use an asymmetric encryption algorithm to agree upon a key, exchanging otherwise *public* information over a potentially untrustworthy medium to broker that agreement. A solid example of this exchange is the entirety of the TLS protocol (which provides support for HTTPS in the browser). Your browser and the server convey some public details, do a bit of math on their own, and come up with an agreed-upon secret key known only to them and never exchanged over the wire.

In the meantime, no one who eavesdrops on this initial handshake can derive the same secret information on its own! Once the browser and server have agreed upon the secret, they can use it to communicate privately between themselves.

Unfortunately, asymmetric encryption algorithms are very limited in the amount of data they can protect. Specifically, the RSA algorithm (one of the most widely used and studied asymmetric encryption schemes) can only protect data up to the same size as the key being used[1]. Said another way, a 2048-bit RSA key can protect at most 256 bytes of data.

This is not nearly enough room to house any meaningful conversation. Practical implementations, including TLS, must then resort to some fancy tricks to protect larger messages.

Hybrid (envelope) Encryption

An envelope encryption scheme is a hybrid of both asymmetric and symmetric encryption. It leverages an asymmetric algorithm (i.e., RSA with 2048-bit keys) to derive a shared secret between two parties. It then uses that secret to encrypt or decrypt an ephemeral (i.e., entirely random) key; this separate key is then used with a symmetric algorithm (i.e., AES256) to encrypt larger amounts of data.

You can see an example of this directly with phparch.com. At the time of this writing, the website uses a TLS certificate issued by Lets Encrypt that expires in December 2023. The certificate signs a 2048-bit RSA key that can be used by the client browser to negotiate an encrypted TLS connection.

Further inspection of the server[2] shows that it's configured to support *multiple* cipher suites within the TLS 1.2 specification—the strongest being `ECDHE-RSA-AES256-GCM-SHA384`. This cipher mode supports[3] elliptic curve Diffie-Helman with ephemeral keys for the key exchange, RSA for

1 https://phpa.me/rsa-encryption
2 https://phpa.me/mozilla-analyze
3 https://phpa.me/ciphersuite-info

authentication (i.e. validating the authenticity of any ephemeral keys provided by the server), AES256 in Galois/Counter Mode for symmetric encryption, and SHA-384 for hashing.

It's a multi-layer envelope allowing you to quickly and transparently leverage multiple forms of encryption to ensure your communication with the server is safe from prying eyes.

Looking Forward

In the land of PHP, we have the option of using either OpenSSL (with appropriate extensions enabled) or Libsodium (part of PHP core since version 7.2) for encryption. Both libraries support all of the styles of encryption above—symmetric, asymmetric, and envelope encryption. However, Libsodium is quite a bit easier to use.

The `sodium_crypto_box_*` family of functions[4] supports asymmetric encryption by way of an envelope scheme natively. You leverage public/private keys everywhere, and the library automatically takes care of the key agreement, ephemeral symmetric key, and underlying symmetric encryption. It's a so-called "pit of success" because the default way to use things is the safe way. You'll have to go out of your way to do things wrong.

The `sodium_crypto_secret_box_*` family of functions[5] instead supports symmetric encryption. Both families of functions work extremely well and handle secure defaults on your behalf, so you don't need to have a Ph.D. to implement encryption the right way.

Whatever you need to encrypt, the primitives provided by PHP make it safe and easy to do so. Knowing which *type* of encryption is appropriate for your needs depends highly on the threat model you've defined for your application and the risks you aim to mitigate.

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

4 https://phpa.me/sodium-crypto-box

5 https://phpa.me/sodium-crypto-secretbox

Controlling Quality in Laravel

Steve McDougall

I want to give an overview of the tools I like to use for controlling the quality of my code, how I configure them, and how I like to use them. Many people have "their" approach to this, but I find discussing it openly is how we learn from one another and start to adapt.

It's 2023, and there are many different options for controlling the code quality of your Laravel applications—from static analysis to type hinting, code linting, and more.

So, firstly, what do I mean by controlling the quality of your Laravel application? I'm talking about the code style, type hinting, return types, and any linting or automated checks you may wish to run. The beauty of this is that you set the bar for what you deem acceptable to you—you can be as strict or lax as you like. There is no wrong answer. Some people prefer super strict code, like myself, whereas others prefer to embrace the dynamic nature of the PHP language and roll with the punches, so to speak.

Something to bear in mind, however, is that the stricter your code is, the fewer dynamic checks and tests you need to run. I want to transport you back in time a second, back to PHP 5.4 or, as I like to call it, the wild west. We didn't have type hinting or return types, and our current static analysis tools were quite new to us as PHP developers. We used to have loads of checks in the logic of our code, like the following: (See Listing 1)

This may be slightly exaggerated, but—it isn't far off how we used to do our code! We had so many logic steps that we used as break out clauses so that logic operations further down our code wouldn't break by default.

Static Analysis

My first stop when it comes to maintaining the quality of my code is to add static analysis to my project, whether that is a Laravel project or an Open Source practice. I like to use PhpStan for my static analysis needs; having used psalm and PhpStan, my preference leans towards PhpStan. This is mostly because it suited my needs, was easy to use, had good debugging output, and had some great extensions available. This is my default configuration for PhpStan:

```
parameters:
    paths:
        - app/

    level: 9

    checkGenericClassInNonGenericObjectType: false
```

Listing 1.

```
1.  public function someMethod($argument)
2.  {
3.    if (is_nl($argument)) {
4.      throw new InvalidArgumentException(
5.        "Argument cannot be null"
6.      );
7.    }
8.
9.    if (is_string($argument)) {
10.     throw new InvalidArgumentException(
11.       "Argument must not be a string"
12.     );
13.   }
14.
15.   if (! is_array($argument)) {
16.     throw new InvludArgumentException(
17.       "Argument must be an array"
18.     );
19.   }
20.
21.   // so some logic here
22. }
```

For those who aren't used to it, I will walk you through it quickly. We start with the parameters of the configuration; I usually start with the **paths** I want to pay attention to. In a standard Laravel application, this is `app`, but I usually have a `src` directory available that I also include. The next check is **level**—I typically keep this at level 9, which is the maximum level it supports. This keeps my code as type-aware as it can be, enforcing it to be strict and making me think through any issues that may come up. The final thing I add is `checkGenericClassInNonGenericObjectType` and set it to false, this makes sure that when I am defining relationships in Eloquent models, etc., I do not get static analysis failures that force me to add too many type notations in docblocks. There is nothing wrong with this, but it doesn't add enough value for the way I like to write Laravel and PHP code in general.

When working with PhpStan, I add this as a composer script so that I can quickly run static analysis checks and have an easy-to-use way to run it in CI. This looks like this:

```
{
  "scripts": {
    "stan": [
      "./vendor/bin/phpstan analyse --memory-limit=3g"
    ]
  }
}
```

I like to manually set the memory limit in the composer script, as I have encountered problems in the past where it has run out of memory in larger projects. I find it easier to do here than remember to add the option to the console command itself. In a GitHub Action, this would look like the following:

```
jobs:
    quality:
        name: "Code Quality Checks"
        steps:

            - name: 'Run PhpStan'
              run: composer run stan
```

I tend to have a separate GitHub Action that I can run on certain events, as I do not want it to run on every part of the development lifecycle—for example a draft PR or a long-running PR.

Code Styling

I will move on to code styling once I am comfortable with the static analysis checks. I used to use PHP CS Fixer for this; however, since the release of Laravel Pint, I have converted. The main reason for this is the ease of configuration and the friendly output that it gives me. I like to keep my code at a consistent PSR-12, but add additional rules to match the way I write PHP. I won't dive into extreme detail here as it is a long file with many options. Here is the raw file as a GIST[1]: (See Listing 2 on the next page)

1 *GIST: https://phpa.me/2023-12-steve-king-gist*

To break it down, a few of the **must have** configuration options are:

- Declare Strict Types: this is always set to true. If you know, you know. If you don't, do it anyway - trust me.
- Final Class: this is always set to true. I **like** final classes. It forces me as a developer to think about whether I should extend a class or look at composition instead. I favor composition over inheritance, so this helps me make smarter decisions without having to question it.
- Fully Qualified Strict Types: this is always set to true. If I am importing a class, I want to import it with a `use` statement. It keeps my code clean, and my IDE automatically folds these nicely out of the way.
- Method Chaining Indentation: this is also set to true. We want our code to read well to avoid potential cognitive overload when reading our code. Indenting these method calls, at least to me, highlights that the logical calling is done in an order, and it is easy to change the order or add a new method call.
- Ordered Traits: this is always set to true. PSR-12 allows me to have them on each line, but I also want them ordered properly, like my imports are.
- Protected to Private: this is also set to true. Along with final classes, this forces me to think about composition first. Another way to make sure I am writing code I am happy with and can maintain more easily in the long run.
- Yoda Style: this is always set to true. This is where we have `null !== $variable` instead of `$variable !== null`, aka as if Yoda would say it. If I am completely honest, this is mostly because I like Star Wars and think Yoda belongs in my code. Do I really need a better reason than that?

This is only a small handful of the rules I like to configure in my code style, but as you can see from the small list above, it focuses on strict conventions and forces me to think about inheritance so that I write better code.

Much like with static analysis, I tend to add this as a composer script to make executing it easy—it also has the side benefit of being chainable.

```
{
    "scripts": {
        "pint": [
            "./vendor/bin/pint"
        ]
    }
}
```

Being able to run `composer pint` is much easier to type than `./vendor/bin/pint` and is a great proxy to use in your GitHub Actions, too. (see code example under listing 2)

Listing 2.

```
{
  "preset": "per",
  "rules": {
    "align_multiline_comment": true,
    "array_indentation": true,
    "array_syntax": true,
    "blank_line_after_namespace": true,
    "blank_line_after_opening_tag": true,
    "combine_consecutive_issets": true,
    "combine_consecutive_unsets": true,
    "concat_space": {
      "spacing": "one"
    },
    "declare_parentheses": true,
    "declare_strict_types": true,
    "explicit_string_variable": true,
    "final_class": true,
    "fully_qualified_strict_types": true,
    "global_namespace_import": {
      "import_classes": true,
      "import_constants": true,
      "import_functions": true
    },
    "is_null": true,
    "lambda_not_used_import": true,
    "logical_operators": true,
    "mb_str_functions": true,
    "method_chaining_indentation": true,
    "modernize_strpos": true,
    "new_with_braces": true,
    "no_empty_comment": true,
    "not_operator_with_space": true,
    "ordered_traits": true,
    "protected_to_private": true,
    "simplified_if_return": true,
    "strict_comparison": true,
    "ternary_to_null_coalescing": true,
    "trim_array_spaces": true,
    "use_arrow_functions": true,
    "void_return": true,
    "yoda_style": true,
```

Listing 2. (cont.)

```
    "array_push": true,
    "assign_null_coalescing_to_coalesce_equal": true,
    "explicit_indirect_variable": true,
    "method_argument_space": {
      "on_multiline": "ensure_fully_multiline"
    },
    "modernize_types_casting": true,
    "no_superfluous_elseif": true,
    "no_useless_else": true,
    "nullable_type_declaration_for_default_null_value":
          true,
    "ordered_imports": {
      "sort_algorithm": "alpha"
    },
    "ordered_class_elements": {
      "order": [
        "use_trait",
        "case",
        "constant",
        "constant_public",
        "constant_protected",
        "constant_private",
        "property_public",
        "property_protected",
        "property_private",
        "construct",
        "destruct",
        "magic",
        "phpunit",
        "method_abstract",
        "method_public_static",
        "method_public",
        "method_protected_static",
        "method_protected",
        "method_private_static",
        "method_private"
      ],
      "sort_algorithm": "none"
    }
  }
}
```

```
jobs:
    quality:
        name: "Code Quality Checks"
        steps:

            - name: 'Run Laravel Pint'
              run: composer run pint
```

Typically, I would add additional steps in this workflow so that it commits any changes it makes. However, I like to make sure that I run this before I commit. It is a kind of ritual I follow when writing code:

- Write code
- Check Static Analysis
- Ensure code style is consistent
- Commit to version control

Type Coverage

Type coverage is relatively new for my code quality checks, mostly because it was a pain to use beforehand. But as I use PestPHP for my testing, I can use the type coverage plugin and run this really easily. There is no configuration needed on this one; all of it is configured as command line arguments that I add to the composer script. Typically, I don't add a minimum coverage argument because I use it mostly to inform me, and using it alongside PhpStan allows me to make sure nothing has slipped the net at all.

```
{
    "scripts": {
        "types": [
            "./vendor/bin/pest --type-coverage"
        ]
    }
}
```

This will run through all the code in my project, ensuring that I have a good level of type coverage, which is important to me. Knowing what types are returned and what types are required is vital—even if I have to use a union type because it isn't completely predictable. This is the biggest issue with something like Laravel; it will often return a series of different types depending on scenarios. While this is great for flexibility, it can lead to bigger issues with being strict with types. It's nothing we can't work around, though.

Using these tools on their own obviously isn't enough, but they put up what I would call good "guard rails" for your application. You still need to write proper tests and make sure your logic is sound. But, with good quality code that has stronger type safety, you can mitigate a vast majority of what I would call "silly" issues. Where you pass the wrong variable or try to do something you know you can't.

Steve McDougall is a conference speaker, technical writer, and YouTube livestreamer. During the day he works on building API tools for Treblle, and in the evenings spends most of his time writing content, or contributing to the PHP open source community. Whatever you do, don't ask him his opinion on twitter/X @JustSteveKing

A View From 2500 Feet

Edward Barnard

My new column begins with a mental exercise. It only takes 24 seconds, and I hope you do play along. With "A View From 2500 Feet", I expect every month will include airplanes, algorithms, and abstract thinking. Airplanes? Yes! Airplanes solved my problem in software development. This month, I'm sharing that story with you.

Twenty-four Seconds

I have a 24-second exercise for you. Do it out loud. Do it without any assistance, looking things up or writing things down. It's a question designed to help evaluate brain function, so you'll definitely want to pass this test!

> *Count backwards from 100 down to zero, by sevens. Count down as quickly as you can. Count "100", "93", and so on. GO.*

Did you take those 24 seconds? If not, please do so before you continue reading here.

I needed you to take on that exercise because how you do it mentally is probably quite different from how you might implement this challenge in PHP code. Listing 1 shows my PHP solution. This is *not* how I thought through the verbal exercise.

Listing 1.

```
<?php

declare(strict_types=1);

$counter = 100;
do {
    `/usr/bin/say $counter`;
    $counter -= 7;
} while ($counter >= 0);
```

I found that, in my own case, each calculation was easiest if I broke it down into multiple steps. Here, for example, is how I can most quickly calculate 72 minus seven in my head:

1. 72 minus 2 is seventy.
2. Seven minus 2 is five, meaning I still need to subtract 5 to reach the answer.
3. 70 minus 5 is 65.

In other words, I use the multiple of ten as my boundary. I can rapidly perform single-digit subtraction in my head, but I'm not so fast with multi-digit subtraction. Knowing that limitation, I break the calculation into a series of single-digit problems:

1. With 65, I break the number 7 into 5 and 2 because subtracting 5 gets me to my preferred boundary.
2. 65 minus 5 yields 60. That's a rapid calculation for me.
3. 60 minus 2 yields 58. That's also a rapid calculation for me.

You likely used a different thought process than I did. You might not have needed to break the calculations down into multiple steps like I did. What I'd like you to do, though, is think through how you *did* accomplish the task.

Could you write down your process as a series of steps, like I just did? When you do, you've written down an algorithm[1]:

> *An algorithm is a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation. Broadly, "algorithm" is a step-by-step procedure for solving a problem or accomplishing some end.*

(See "Additional Reading" below.)

Crossover Skill

Why am I making such a big point of stating the obvious? We all know how to develop software, which means we all have a working knowledge of developing and implementing algorithms. For example, when I take on a bugfix, I don't announce, "I'm going to implement an algorithm".

Step back and take a look at the larger picture. There's a crossover skill hiding in this exercise: documenting a business process. Discovering the business process often involves "picking the expert's brain". This might come from a casual conversation, drawing things out on a whiteboard, or the formal "workshop" structure of EventStorming[2].

1 https://phpa.me/mw-algorithm

2 https://www.eventstorming.com

I find it's hard to write down a thought process. It's usually just as difficult to document a business process based on tribal knowledge[3]. How do you articulate something that's automatic, no longer requiring thought? As a result, I find it becomes difficult to see any reason for writing down the business process. I often see strong pressure to skip this step (of documenting the business process as such) and move directly to writing the code.

Here's the observation I mentioned at work just yesterday:

> *One's options ALWAYS are: Design it now or design it later.*

Crossing the Mississippi

I need to tell you a story. As of March 2023, my software development efforts were not going well. I chose to stop learning, stop exploring, stop innovating. Why? Because my efforts were being ignored. I was making our mess of legacy code worse because my code didn't look like anything else in the codebase. My code even had thousands of unit and integration tests.

I had considered this type of situation before as "The Train Wreck" (see "Related Reading" below), but I had proposed "The Train Wreck" as an ethics problem: what can you do when you're asked to do something that you shouldn't? The answer is that very few people have the means to walk away from their current employer.

To me, my situation of March 2023 was a related use case: the situation felt untenable, but I did *not* want to walk away. The first step in my solution was to stop heading in a different direction. Stop making things worse. "Better" code, if nobody else understands it, is not better code.

My situation was not unique. The Jargon File identifies a Real Programmer[4]:

> *A Real Programmer's code can awe with its fiendish brilliance, even as its crockishness appalls… because someday, somebody else might have to try to understand their code in order to change it. Their successors generally consider it a Good Thing that there aren't many Real Programmers around anymore.*

Dwelling on the frustration doesn't help the situation. What's the answer? I needed to take a step back. I needed to look at the situation one step removed, looking for the context, the larger picture. I'm calling this back-step "the view from 2500 feet" because I mean that literally! To continue the "play on words", my "crossover skill" became crossing the Mississippi River at 2500 feet in a World War II "warbird", Figure 1. To my surprise and delight, I found that this skill directly relates to software development. I'll explain below.

3 https://phpa.me/wikipedia

4 https://phpa.me/real-programmer

Figure 1.

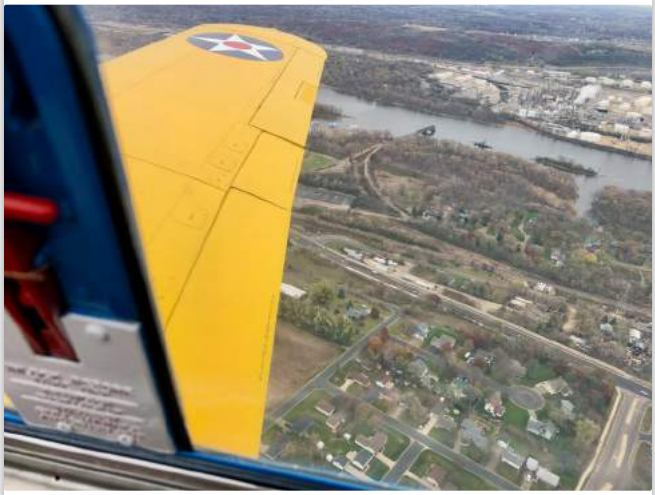


Figure 1. Crossing the Mississippi

How *does* crossing the Mississippi relate to software development? It took me months to find that answer, and now I can have all kinds of fun sharing those discoveries. Software development can and should be *fun*. That's always been true.

The Abstract View

As of March 2023, I had nothing left to write. Not about coding, anyway. I've continued to write PHP code; therefore, I continued to write the "DDD Alley" column here. I'm now done writing about code, at least for a while. What can I do? I can talk in the abstract. I can show how this skill over "here" can be applied to that situation over "there". There's more to software development than writing code, and I have experiences to share.

What do I mean by "abstract", and does it mean more than PHP's `abstract` keyword?

Here's my favorite example. Picture being at a small construction-project site. Figure 2 shows bags of cement.

Figure 2.

Figure 2. Cement

Cement[5], in this sense, is *abstract*. It's not directly useful by itself. The sidewalk in Figure 3, by contrast, is a *concrete implementation*. The bags of cement hold potential for many different purposes, and the sidewalk implements one specific purpose. The sidewalk is an example of applying the abstract potential.

Figure 3.

Figure 3. Concrete implementation

Expectation of Learning

The next software project, bugfix, or exploration, will usually require learning something new. I expect that I won't yet know enough to "solve this" or "develop that". This is why my choice in March to "stop learning" led to endless frustration.

Learning, however, is a crossover skill. It's important to know how *you* best learn. I often find myself exploring *how to learn*. I prefer physical books, for example, and dislike electronic books and PDFs. The small screen size and low resolution, as compared to a physical book, slow me down. I've been trying newer devices, such as a relatively large Kindle Fire, and that's an improvement. Always be learning.

Not Software

Frustration's not healthy. We all know that! The answer came from a weird direction. Eight months later, I can see it was a good answer, so it's time to share the story.

My wife Susan teaches high school, including study skills. In the winter of 2022-23, one of her (graduated) former students asked if Susan would consider tutoring him. He was studying for his Federal Aviation Administration (FAA) aviation mechanic[6] certification. Susan knows nothing about airplane mechanics, but she knows about study skills and how to break down the curriculum to be most beneficial for her former student.

She was applying an abstract skill, you will note, to a novel situation. They met in a World War II-era airplane hangar. Come mid-March, her student suggested that I might be interested in a "Living History" flight in one of their warbirds. She could purchase the flight as an anniversary gift.

I stopped by and crawled into the nose of a small World War II bomber that still flies. Little did I know that I'd later be riding that front blue plane (Figure 4) across the Mississippi River (Figure 1), in an open cockpit, in November, in Minnesota!

Figure 4.

Figure 4. The Blue Plane

I ran the numbers. I figured out that if I *joined* the Commemorative Air Force for a year, I'd have the chance to learn far more than if I were a passenger for one ride. Learning is important! And at that moment, learning about something that wasn't software was also important.

Finding a Place

I'm not a pilot. I'm not a mechanic. That presents an obvious problem: how could I be useful to an organization centered on keeping historical military aircraft ("warbirds") flying? Luckily, there's an obvious answer. It's the reason we "keep 'em flying", to share the history, to help people touch, feel, and relate to what it was like. The families often don't know, and finding out can be a deep emotional experience.

Bear with me a moment longer, and then I'll bring the story back to software development. Figure 5 (on the next page) shows a gentleman, 98 years old, who stopped by with his daughter for the Veterans Day event at Delta Air Lines in Minneapolis/St Paul (MSP airport).

5 https://phpa.me/wikimedia-cement

6 https://www.faa.gov/mechanics/become

Figure 5.

Figure 5. Doolittle Airplane Mods

Standing next to our B-25 Mitchell aircraft, he began describing to me how a certain group of B-25 aircraft were prepared for the Doolittle Raid[7] in April 1942. He said they took all the guns out to lighten the load; how they created wooden broomsticks to look like guns pointing out the rear. I was agreeing with each point, that Yes, this was correct, and that was correct, and so on.

The Doolittle Raiders *were* in Minneapolis. That *is* where the airplanes were modified, so this all made sense. He stated adamantly that the modifications were in Minneapolis, *not* St Paul. That is correct, but not well known. This gentleman clearly knew his stuff. I confirmed to his daughter that everything he'd said was correct.

His voice was soft and whispery. I finally realized he was saying, "I" did this, and "we" did that. To be clear that I understood, I asked him directly, "Were you personally helping modify the Doolittle airplanes?" He confirmed, "Yes". His daughter was in tears; she didn't know. He took part in a significant (but top secret at the time) piece of history, but his family didn't know.

We find this is often the case. The veterans and those who participated don't share things with their families. But standing next to an operational B-25, 81 years later, he began to talk about his part in the event.

History and Context

Back in March, I was looking for a way to be useful. I gave myself a project goal of learning everything I could in one month and started the clock—One month.

The first problem with that goal should be obvious. This is too nebulous a goal! It takes in too much territory. What might a "minimum viable product" be? Can I narrow this down to a minimum set of features? Note that, once again, I'm using a crossover skill from software development. I'm trying to bring some sanity to an insane, unreachable set of requirements.

I decided to focus on bombers and ignore learning about fighters. I further narrowed it down to the aircraft itself as opposed to campaigns, strategies, adventures, and politics. I further narrowed the goal to a single aircraft type, the B-25 Mitchell, since I was standing next to one at the time. I was surrounded by subject-matter experts.

A bit of project discovery brought me more information. We needed tour guides and people available to share the history and context, answer questions, and so on. Perhaps I could be one of those people. I also had two months rather than just one. That's because our flying season starts in mid-May. It would be a day of Living History Flight Experiences, and my job would be one of those providing ramp tours.

Wait. What? I couldn't even tell the difference between the blue plane and the yellow plane. I could recognize the bomber every time because it had *two* propellers.

In software development, we call this point "fake it until you make it". My advice to you is, "Do your homework"! The time spent gaining deeper knowledge of a software system will usually pay off.

Figure 6 shows the notebook I made myself in sheer panic. I included a small image of each airplane I was describing because, at that point, I really couldn't tell them apart!

Figure 6.

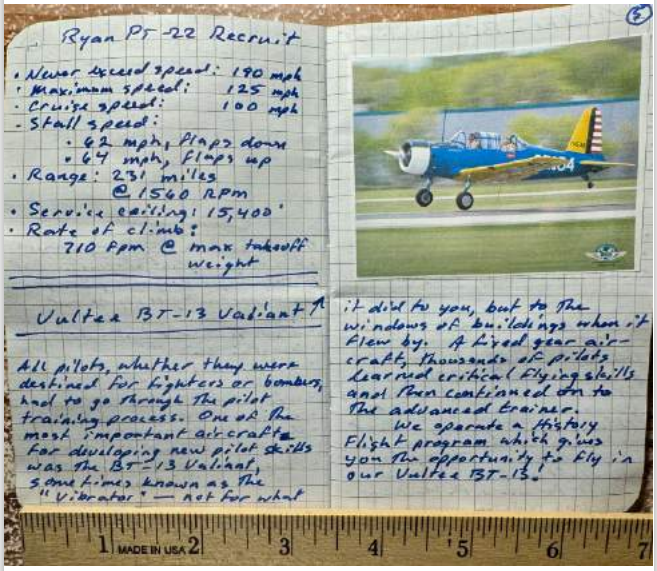


Figure 6. "The Gen" Notes

There I was, on the airport ramp, that first day. As I walked people around and crawled inside the aircraft, I explained things, shared the context, and told some of the stories. (I'm the photographer for Figure 7, so I'm not in the photo.) After a while, I noticed something interesting.

7 https://phpa.me/doolittle-raid

Figure 7.

Figure 7. Pre-Flight Briefing

Our own crews were quietly standing behind the visitors, listening to what I was saying. I figured they were listening in and evaluating. Feedback would be great! I was relieved not to have any corrections on the spot.

Later, however, one of our aircraft restoration experts told me privately that I really need to get my stories written down so that our other tour guides know what to say. I thought that was pretty good for being the new guy!

Summary and Looking Ahead

This month I changed the column to "A View From 2500 Feet". I discussed algorithms and airplanes. I showed how contrasting two wildly different topics can lead to identifying "crossover skills" that apply to both situations. Such skills have become abstract, with multiple implementations.

I gave an example of the difference between "fake it until you make it" and "do your homework". I find that diving more deeply into a given context or problem at hand usually pays off.

Next month, I'll explain that you can't solve a process problem by writing code. This month, we discussed airplanes and algorithms. Next month, we'll address airplanes and people–and process. It turns out that studying people can be as valuable to developing software as studying algorithms.

Additional Reading

- Chapter 7, "Design an Algorithm in Your Head", *Living Amongst the Wizards of Cray Research*[8] by Edward Barnard
- "The Train Wreck: When Safety is Discretionary", July 2017 php[architect]
- Chapter 16, "Imposter Syndrome", *Living Amongst the Wizards of Cray Research*, explains my expectation of lifelong learning

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.

8 https://leanpub.com/dragons

Generators For Efficient Code

Chris Tankersly

One of the reasons that developers migrate toward higher-level languages and scripting engines like PHP is because of all the nice things they bring to day-to-day development. PHP was born out of the need to not have to write raw C programs to generate dynamic content. You can still program in C for web development if you want, but PHP has some nice features that make building web applications faster.

There is a cost to this. When you write in a language that compiles down to a specific CPU, like x64 or ARM, you lose some portability, but you gain a lot of performance as the compiler knows how to work with that CPU architecture. When you start to move up the chain to languages like Java or .NET, they compile down to an intermediary system that then runs on the host. For PHP, the scripts we write are converted to opcodes, and those opcodes are interpreted are run through a C engine—both those intermediary systems, like the JVM or PHP's Zend Engine, slow things down some.

With modern computers, this is rarely an issue. We can easily give up a few CPU cycles, knowing that, at the end of the day, we are far more bottlenecked by things like databases or calls to third-party services. We can service thousands of requests on hardware that would restrict JVM or CLR-based applications. If PHP is the slowest part of the overall application, you have other problems.

There will be times when performance does become an issue. One of the most common performance issues developers encounter is memory management. PHP has a decent garbage collector, which has continued improving with each release, but what happens when you need to process a large amount of data?

The last thing we want to do is waste a bunch of memory by reading entire datasets into memory. We also want to reduce the amount of times we need to process a bit of data. We are totally fine with small inefficiencies in our code, but we want to make sure we do not introduce large inefficiencies.

PHP generators are a powerful feature that can be used to improve the memory efficiency of your applications. Generators allow you to iterate over a sequence of values without having to store the entire sequence in memory at once. This can be especially useful when working with large datasets or when you need to process data in a streaming fashion.

What is a Generator?

A generator[1] is a function that can yield multiple values over time. When a generator function is called, it returns a Generator object. This Generator object can then be iterated over using things like a `foreach` loop.

When the `foreach` loop encounters a yield statement in the generator function, it will pause the execution of the generator function and return the yielded value. The generator function will then resume execution when the foreach loop needs the next value.

This allows the generator function to produce a sequence of values without having to store the entire sequence in memory at once.

How Things Worked Before Generators

Before PHP generators were introduced, the only way to iterate over a large dataset was to store the entire dataset in memory at once. This could lead to high memory usage and performance problems, especially for applications that need to process large amounts of data.

For example, to read a large file line by line, you would need to read the entire file into memory at once before you could start processing it. This could lead to memory problems if the file was very large. (See Listing 1)

Listing 1.

```
1. class DataService {
2.     private array $data = [];
3.     public function __construct(string $filename) {
4.         $fh = fopen($filename, 'r');
5.         while (($row = fgetcsv($fh, 1024)) !== false) {
6.             $this->data[] = $row;
7.         }
8.         fclose($fh);
9.     }
10.
11.    public function getData(): array {
12.        return $this->data;
13.    }
14. }
15.
16. $dataService = new DataService('really-big-file.csv');
17. foreach ($dataService->getData() as $row) {
18.     // Do something with $row
19. }
```

Similarly, to process streaming data, you would need to store the entire dataset in memory before you could start processing it. This could also lead to memory and performance

1 https://phpa.me/generator-overview

problems since you would have to wait for the entire dataset to be received before you could start processing it.

The performance impacts of not having generators can be significant. The high memory usage can lead to out-of-memory errors and crashes for applications that need to process large amounts of data. Additionally, high memory usage can also lead to performance problems, as the garbage collector will have to work harder to free up memory.

Why Use Generators?

The largest reason to use a generator is to reduce memory usage. This is accomplished through a combination of how the generator returns results, how your application processes the data, and by letting the garbage collector better clean up memory as it is no longer needed.

In general, generators should be used whenever you need to write memory-efficient and scalable code.

Reducing Memory Usage

If we rewrite our little `DataService` class to use generators, the largest benefit is that we are no longer reading the entire file. As the `foreach` loop iterates of the `\Generator` that `getData()` has returned, we only ever load one row of data (up to 1024 characters) into memory at a time. As the loop finishes and moves on to the next row, the previous rows can be garbage collected as the loop discards them. (See Listing 2)

Listing 2.

```
class DataService {
    public function __construct(
        private string $filename
    ) {
    }

    public function getData(): \Generator {
        $fh = fopen($this->filename, 'r');
        while (($row = fgetcsv($fh, 1024)) !== false) {
            yield $row;
        }
        fclose($fh);
    }
}

$dataService = new DataService('really-big-file.csv');
foreach ($dataService->getData() as $row) {
    // Do something with $row
}
```

This can be hugely beneficial for large amounts of data, allowing you to process much larger files without any changes to your PHP installation's memory settings. We may open the file more often if we have multiple calls to `getData()`, but overall, we are probably more efficient than storing the huge dataset in memory.

Processing Streaming Data

Another common use case is dealing with streaming information. In this case, we may want to read the information in chunks rather than wait for an entire transmission to finish. We may also be dealing with a situation where the connected party is constantly sending data to us, so we never reach the "end" of its message. (See Listing 3)

Listing 3.

```
class SocketDataGenerator {
  public function __construct(private $socket) {
  }

  public function readData(): \Generator {
    $sock = $this->socket;
    while (($buffer = fread($sock, 1024)) !== false) {
      yield $buffer;
    }
  }
}

$host = '127.0.0.1';
$port = 5000;
$socket = stream_socket_client(
    "tcp://$host:$port",
    $errno,
    $errstr,
    30
);
$dataGenerator = new SocketDataGenerator($socket);

foreach ($dataGenerator->readData() as $data) {
    // Process the data
}
```

Here, we connect to another program that is listening for connections to port 5000. We then wait until we fill up 1024 bytes of data. That buffer is then yielded back to our `foreach` loop and can be processed. This allows us to continue to read the data no matter how large the amount of data the other program sends to us over an indeterminate amount of time. It also allows us to process the data sooner rather than waiting for the entire transmission.

In some cases, you may need to wait for multiple buffers to be yielded back, but you could adjust the `fread()` length to create a larger buffer or just wait for some marker that helps you know when an overall chunk is "finished."

This also has a nice side effect of making the code easier to read and maintain. You could use some sort of `Iterable` class to do something similar, but here we have reduced everything down to one main method for reading, and a `forloop` for processing the data.

Lazy Loading of Data

Lazy Loading is a general programming paradigm of only fetching data when needed, compared to eager loading, which

fetches all potentially needed information upfront. A good example is ORMs, which handle multiple object relationships. Many of them allow you to request an object with everything loaded, while others allow you to lazy load relationships only when you call them.

We can do something like this with generators. (See Listing 4)

Listing 4.

```
class LazyDatabaseUserIterator implements Iterator
{
    private mixed $row;
    private int $index = 0;
    private mixed $statement;

    public function __construct(private \\PDO $pdo) {
    }

    public function current(): mixed
    {
        return $this->row;
    }

    public function next(): void
    {
        $this->row = $this->statement->fetch(
            PDO::FETCH_OBJ
        );
        $this->index++;
    }

    public function key(): int
    {
        return $this->index;
    }

    public function valid(): bool
    {
        return !empty($this->row);
    }

    public function rewind(): void
    {
        $this->statement = $this->pdo->prepare(
            "SELECT * FROM Users"
        );
        $this->statement->execute();
        $this->row = $this->statement->fetch(
            PDO::FETCH_OBJ
        );
    }
}

$users = new Users(/** PDO connection **/);
foreach ($users as $user) {
    // Do something with the user
}
```

`LazyDatabaseUserIterator` is an example of how we would apply this using the `Iterable` interface. The basic idea is that as we start to iterate over the `$users` object, it internally calls `rewind()` to kick things off. This fires off a prepared statement to our database server via PDO. While this is a `SELECT *` call that we expect to return multiple results, we use `fetch()` to return just a single result from the statement.

As the `foreach` loop continues, `next()` is called, which fetches another row. This continues until we have no more rows left, and `next()`'s `fetch()` call returns null.

How would this look with a generator? (See Listing 5)

Listing 5.

```
class UserLoaderGenerator {
    public function __construct(private \PDO $pdo) {
    }

    public function getUsers(): \Generator {
        $statement = $this->pdo->prepare(
            "SELECT * FROM Users"
        );
        $statement->execute();

        while (
            $row = $statement->fetch(PDO::FETCH_OBJ)
        ) {
            yield $row;
        }
    }
}

$userLoader = new UserLoaderGenerator(
    /** PDO connection **/
);
foreach ($userLoader->getUsers() as $user) {
    // Process each user
}
```

Generators cut out much of the code we needed for the iterator. While we can no longer iterate directly on that `$users` object, we can wrap everything up into a method that returns a simple generator that does the same thing as all that iterator code. The only difference is that we no longer need to worry about keeping state and maintaining all that iterator code.

Sending Data Back to a Generator

Generators are great for getting data from places, but did you know that you can also send data back to a generator? The object that is `yield`ed can actually have data pushed back into the logic that generated the `yield` call for further processing, allowing an almost coroutine-like handling of logic.

Let's jump back to our socket example. Since we are using a generator, we can send a signal back that says to disconnect when we receive something specific that indicates we should close the connection. Specifically, if the data contains

the string "–TERMINATE–", the generator will be signaled to stop processing and close the socket.

First, let's revise the `SocketDataGenerator` class: (See Listing 6)

Listing 6.

```
1. class SocketDataGenerator {
2.     public function __construct(private $socket) {
3.     }
4.
5.     public function readData(): \Generator {
6.         while (
7.             ($buffer = fread($this->socket, 1024))
8.                     !== false
9.         ) {
10.            $command = (yield $buffer);
11.            if ($command === 'disconnect') {
12.                $this->isActive = false;
13.                socket_close($this->socket);
14.                break;
15.            }
16.        }
17.    }
18. }
```

Inside the `readData` method, we loop as before as long as data is successfully read from the socket. After yielding the buffer, the generator waits for a command from the caller. If it receives the 'disconnect' command, it closes the socket and breaks out of the loop.

Now let's look at how we interact with this generator in our calling code: (See Listing 7)

Listing 7.

```
1. $host = '127.0.0.1';
2. $port = 5000;
3. $socket = stream_socket_client(
4.     "tcp://$host:$port", $errno, $errstr, 30
5. );
6. $dataGenerator = new SocketDataGenerator($socket);
7.
8. foreach ($dataGenerator->readData() as $data) {
9.     // Process the data
10.
11.    if (str_contains($data, '--TERMINATE--')) {
12.        $data->send('disconnect');
13.    }
14. }
```

As before, we loop over the data yielded by the `readData` generator. Inside this loop, we check each chunk of data for the presence of the string "–TERMINATE–". If this string is found, we use the `send` method to send a 'disconnect' command back to the generator. This command is received by the generator immediately after the `yield` expression that produced the current chunk of data. When the generator receives this command, it closes the socket and breaks from the loop.

If you are interested in diving deeper into how generators can be used for things such as this, Nikita Popov, an Internals contributor, goes into much more detail on using `send()` and getting cooperative multitasking working using generators. Check out his article on Cooperative multitasking using coroutines (in PHP!) (npopov.com)[2] for more information.

What About Fibers?

Generators and fibers, both available in PHP, serve different purposes and are used in distinct scenarios, though they share some similarities in terms of yielding control back to their caller.

Generators, introduced in PHP 5.5, are a way to implement simple iterators. As shown in this article, they allow you to write functions that can pause execution and resume where they left off, maintaining their state (variables, counters, etc.) between pauses. Generators are primarily used for iteration—they yield a series of values, one at a time, each time they are iterated over.

Fibers[3], introduced in PHP 8.1, are more about concurrency than iteration. A fiber is a low-level mechanism that allows creating multiple code paths that can be executed somewhat simultaneously. Unlike threads, fibers are not executed in parallel and do not offer true multitasking.

Instead, they are executed cooperatively: a fiber explicitly yields control back to its caller, at which point another fiber can start or resume its execution. This feature is particularly useful for asynchronous programming, enabling non-blocking I/O operations, and handling multiple tasks that wait for resources without blocking each other. Fibers can suspend and resume their execution at any point with a similar `yield` keyword, but they are not tied to the concept of iterating over data. They are more about splitting the execution flow in a manageable and less resource-intensive way than threads.

While both generators and fibers in PHP use a form of the `yield` keyword to pause and resume execution, their primary uses are quite different. Generators are designed for producing sequences of data efficiently, often in the context of iteration. In contrast, fibers are a tool for managing concurrent tasks within a single thread, suitable for asynchronous programming and non-blocking operations. The choice between using a generator or a fiber depends largely on whether the task at hand is iterative data processing or concurrent task management.

2 https://phpa.me/npopov-multitasking

3 https://phpa.me/coroutines

Clean Up Code with Generators

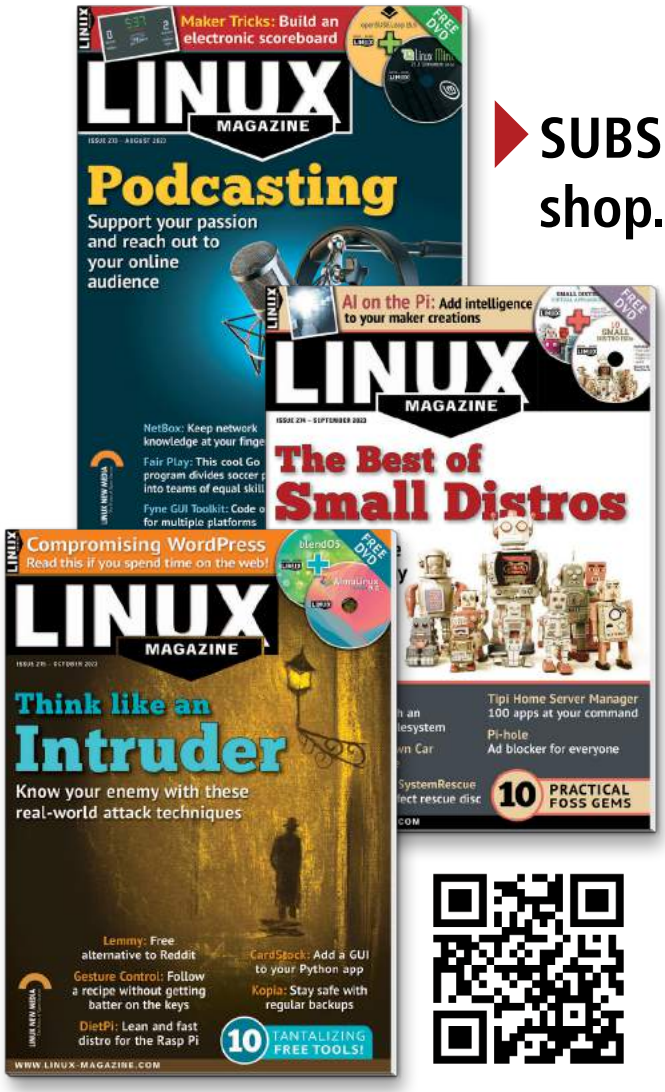
PHP's introduction of generators marks a significant advancement in the language's capability to handle complex data processing and task management. Generators, with their efficient handling of data sequences and memory management, are indispensable for applications dealing with large datasets or needing to process streams of data iteratively.

The next time you need to work with large sets of data, or even maybe just iterate over data more efficiently, take a look at generators. The classic way of loops will never go away, but generators might make things easier to read and maintain.

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.

Linux Magazine Subscription

Print and digital options
12 issues per year



SUBSCRIBE
shop.linuxnewmedia.com

Expand your Linux skills:

- In-depth articles on trending topics, including Bitcoin, ransomware, cloud computing, and more!
- How-tos and tutorials on useful tools that will save you time and protect your data
- Troubleshooting and optimization tips
- Insightful news on crucial developments in the world of open source
- Cool projects for Raspberry Pi, Arduino, and other maker-board systems

Go farther and do more with Linux, subscribe today and never miss another issue!

Follow us

@linux_pro
Linux Magazine
@linuxpromagazine
@linuxmagazine

Need more Linux?

Subscribe free to Linux Update

Our free Linux Update newsletter delivers insightful articles and tech tips to your inbox every week.

bit.ly/Linux-Update

Quicksort

Oscar Merida

Quicksort is a popular sorting algorithm first published in 1961. It works quicker than most other algorithms with randomized data. It uses a "divide-and-conquer" approach to sort an array recursively. Let's look at how to use it in PHP and learn why this could be a very short article.

Recap

> *Generate an array of N random numbers and use Quicksort to order them from smallest to largest. Pick as large a range as you want to work with, but don't make the range too small, and use your comb sort function to order it.*
>
> *Again, generate many such arrays, then collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.*

Algorithm

The quicksort algorithm begins by picking a pivot element and then swapping elements so that elements greater than the pivot are in one sub-array and elements less than the pivot are in the other sub-array. In the worst case, it's no better than the average complexity of bubble sort (O^2). On average, it takes O(n log n) comparisons.

Big-o Notation

> *When reading about algorithms, you're likely to run into "Big-O Notation." This measure approximates the performance of an algorithm for comparison with others. For sorting algorithms, the n represents this of the array we are sorting. O^2 means an algorithm requires, on average, a number of swaps equal to the square of its size.*

Tony Hoare developed quick sort because he needed to sort words in Russian sentences for a machine translation project. He was asked to write Shellsort code and instead bet his boss he knew of a faster algorithm—a bet he ultimately won. Both the C standard library and java use quicksort in their built-in sorting functions. A straightforward approach to selecting the pivot element is to use the last one in the array. Picking a "good" pivot element is crucial (pivotal?) for quicksort performance. As a result, most improvements to the original algorithm focus on how to select the pivot.

Pseudo-code

Quicksort relies on two functions. First, we have a base function. It calls a partition function to move the pivot to its correct position, with smaller elements before it and larger ones after. It then recursively calls itself to sort each partition. Pseudocode in Listing 1 is from nie.edu[1]

Implementation

Recall how C and Java use quicksort in their standard library? PHP's `sort()`[2] function also uses quicksort. It selects a pivot point in the middle of the array. `sort()` should be faster than any user land implementation we could write, so if you used it for this puzzle, technically, you are done. But let's implement it to see how it would be done and to compare it to this core function.

Listing 2 shows a PHP implementation of the pseudocode shown earlier. Not shown is the `swap_elements()` function from other algorithms. Also, one thing that tripped me up—and might get you too—is that we're sorting the arrays in place, so make sure you're passing it by reference to the functions by prepending an `&` in the function signature.

HackerEarth[3] has an excellent, interactive visualization of quick sort and other algorithms.

Benchmarks

The table below shows how long our implementation of the quick sort took to sort arrays of various sizes.

User land	1000	5000	10000
Fastest	0.01934	0.1185	0.2580
Slowest	0.04587	0.2326	0.3694
Mean	0.02707	0.1450	0.3063

Similarly, we can measure the built-in quicksort's performance. We should not be surprised that the Zend Engine's C-implementation is orders-of-magnitude faster than any of our sorting algorithms. See Figure 1.

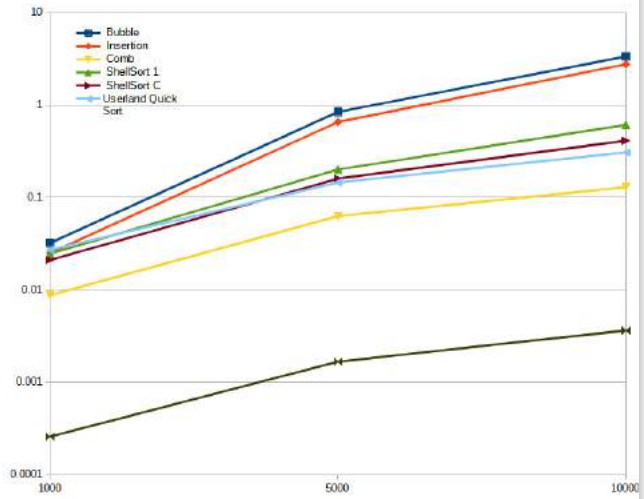
1 https://phpa.me/quick-sort

2 https://php.net/sort

3 https://phpa.me/quick-sort-visualized

sort()	1000	5000	10000
Fastest	0.0002099	0.001361	0.00307
Slowest	0.0005871	0.002790	0.0059
Mean	0.0002572	0.001667	0.00363

Figure 1.



Playing Cards

For next month, come up with a way to represent the cards in a typical poker deck, excluding jokers. Cards can be one of four suits—hearts, diamonds, spades, and clubs. They can be a number from 2 to 10 or a face card like a jack, queen, king, or ace. Implement some way to shuffle a standard 52-card deck.

Some Guidelines And Tips

- *The puzzles can be solved with pure PHP. No frameworks or libraries are required.*
- *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
- *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
- *Refactoring is encouraged.*
- *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
- *Go ahead and try many solutions if you like.*
- *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH[4] 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.*

Listings:

Listing 1.

```
1. procedure quick_sort(
2.     array : list of sortable items,
3.     start : first element of list,
4.     end : last element of list
5. )
6.     if start < end
7.         pivot_point ← partition(array, start, end)
8.         quick_sort(array, start, pivot_point - 1)
9.         quick_sort(array, pivot_point + 1, end)
10.    end if
11. end
12.
13. procedure partition(
14.     array : list of sortable items,
15.     start : first element of list,
16.     end : last element of list
17. )
18.     mid ← (start + end) / 2
19.     swap array[start] and array[mid]
```

Listing 1. (cont.)

```
20.
21.     pivot_index ← start
22.     pivot_value ← array[start]
23.
24.     scan ← start + 1
25.     while scan <= end
26.         if array[scan] < pivot_value
27.             pivot_index ← pivot_index + 1
28.             swap array[pivot_index] and array[scan]
29.         end if
30.         scan ← scan + 1
31.     end while
32.
33.     swap array[start] and array[pivot_index]
34.
35.     return pivot_index
36. end procedure
```

4 https://psysh.org

Listing 2.

```php
<?php

/**
 * We're assuming sequential integer keys
 * @param array<int, scalar> $list
 */
function quicksort(
    array &$list,
    int $first = 0,
    int $last = null
): void {
    if ($last === null) {
        $last = count($list) - 1;
    }

    if ($first >= $last) {
        return;
    }

    $pivotIdx = partition($list, $first, $last);
    quicksort($list, $first, $pivotIdx - 1);
    quicksort($list, $pivotIdx + 1, $last);
}

/**
 * @param array<int, scalar> $list
 */
```

Listing 2. (cont.)

```php
function partition(
    array &$list,
    int $first,
    int $last
): int {
    // select pivot
    $mid = floor(($first + $last) / 2);
    // move the pivot to the start of the array
    swap_elements($list, $first, $mid);

    $index = $first;
    $value = $list[$first];

    $scan = $first + 1;
    // loop through the partition and swap
    // if an element is less than the pivot
    while ($scan <= $last) {
        if ($list[$scan] < $value) {
            $index++;
            swap_elements($list, $index, $scan);
        }
        $scan++;
    }

    swap_elements($list, $first, $index);

    return $index;
}
```

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


CREATING A PHP LIBRARY WITH PACKAGIST
GETTING OUT OF PHP DEPENDENCY HELL WITH COMPOSER
php[podcast] Episode 2023.1.1
QUEUES WITH PHP'S SPLQUEUE
THE PHP DATETIME CLASSES
WHAT'S CHANGED IN PHP 8.2
WRITING CLEANER MORE RESILIENT CODE WITH VALUE OBJECTS
DEPLOYING OUR CODE WITH DEPLOYER 7.0
php[architect]
STAY UP-TO-DATE
WITH THE LATEST PHP TRENDS
AND TECHNOLOGIES
Parallelize
Decrypting Cryptography
About PHP's Git Server
PHP[WORLD]
LIVE STREAM
THE FUTURE IS COMMUNITY
PHP[TEK]
SUBSCRIBE
www.youtube.com/@Phparch

Creating Accessible Forms

Maxwell Ivey

I want you to think about forms in a different way. They aren't just a routine part of your website. They are, in fact, the lifeblood of any business that sells things online. Whether that is a full-blown e-commerce site or simply someone who offers services from their website. Forms can either be a smooth, beautiful bridge or a creeky, old draw bridge designed to keep robbers out.

Let's talk more about forms—more specifically, creating accessible forms.

So, let's explore some aspects of what an accessible form would look like and how to avoid some of the most common pitfalls.

To start, I want to make a general comment about building forms. And this statement is in no way meant to offend this community. After all, you may be a great coder at many things, but you may not understand the vagaries of creating accessible forms. Additionally, you may not have the time to educate yourself or to ensure you follow the guidelines I will share with you when rushing to complete a project on deadline.

But, if you have any doubts about your ability to create an accessible form, then please just use one of the established form builders.

Google Forms[1], Typeform[2], and Gravity Forms[3] are all platforms where someone can build highly accessible forms.

Personally, I prefer Google Forms the most, but the backend of their site is not accessible for me to create my own forms. So, I usually have to use one of the others.

Please only use a survey form from MailChimp or one of the other survey sites if you have had it tested. Quite often, the latest, coolest form builder or survey creator will turn out to be inaccessible.

Now, let's start with something most people would not consider part of accessibility. When designing a form, it is much easier for screen readers and screen magnification users to navigate forms when the questions are divided into multiple pages with a minimum number of items on each screen.

This is one area where designing for accessibility will also improve things for all your users. A large number of internet users are visiting your sites on mobile devices, so this benefits them as well since fewer questions per page will fit better. In fact, I have come to the conclusion that one question per page is ideal for everyone unless you have a huge number of questions to ask. Then, use your own judgment so that no one screen overwhelms your users.

It is also very helpful to squeeze the screen down so that the only thing being displayed is your form. This can be especially helpful to people with attention deficit issues.

After limiting the number of items per page, the next issue will be screen navigation. As you remember from my article on navigation limitations, most adaptive tech users will navigate a form using the tab or shift-tab keys depending on whether they are moving from beginning to end or in reverse.

If the form is embedded in a page with many other items, placing a heading at the beginning of the form is very helpful as it will allow people like me to navigate directly to the beginning of the form fields on each page.

Key elements in accessible forms Here are some things that are vital as I'm navigating a form:

- As I move to the next field with my tab key, my cursor should be placed in the edit field.
- The question should announce what it is.
- I should be given any instructions that will affect how I enter my answer.
- I should hear any suggestions for how to complete a question.
- As I enter my information or after I have entered it, the site should alert me if I have entered my answer incorrectly.
- Then, it should tell me how to correct my answer.
- Also, it would be great to have an item to the right of each question that displays the word Okay or a Green Check with appropriate alt text (alternative text: describes an image on a page, helping people with visual impairments.). This becomes more important the more items you put on each screen. When I reach the bottom of the page or the form, I need to be able to press that button without the use of a mouse. Because, as you know, adaptive tech users cannot always exercise a mouse click.

Quite often, I will get to the end of a form and press submit only to find that an error was undetected while I was completing the form. If this happens, the form should tell me what items are incorrect and advise me how to fix them.

1 https://www.google.com/forms/about

2 https://www.typeform.com

3 https://www.gravityforms.com

- Once I have successfully completed a form, there should be no doubt that this is the case. There should be a notification that is spoken, or there should be a header that says thanks, congrats, or some other positive confirmation.
- Additionally, I should receive an email notification right away that my submission has been received. No matter how good a person with disabilities is with their tech, many of us will fear the effects of getting a form wrong. Or, we think we sent it but never heard anything from the site owner.

Now, let's talk about a couple of special examples:

Most people get checkboxes right, but they may not give clear enough instructions. It is important to know if you only want us to pick one, multiple, or all that apply. And if you do want us to pick up to three out of five, for example, then please have an alert that will tell us if we have exceeded the maximum number.

With *select* boxes, it depends on which format you are using. If you have a list of possible answers, those are best. They are usually triggered by pressing the space bar and then arrowing up or down through the list. The challenging ones are where you type in a word or a few characters and it returns a list of possible options. These are more difficult because selecting the item you want the first time is often next to impossible. I often spend many minutes with one of these kinds of select boxes. And more often, I have to contact the site owner for help.

I recently visited a site where they used *select* boxes with a great twist. After I typed in a few characters, it displayed a list of linked text. I then clicked the link that worked best. I may have seen this on Linked In. But it may have been on a UK marketing site, too.

Then there are buttons:

This is where there will be a list, usually of three or four, but it can be up to 10 or 11 options, and you press a button. You can usually also navigate this by using your arrow keys. With these, it's best that each time I move up or down, it announces the selection I would be making and the related effects on my answers. For example, at checkout, Walmart has one for delivery orders where you can choose when you want your groceries delivered. As you move through the options, it tells you how much the delivery fee will be at that time and when you have to be checked out in order to qualify for your chosen time.

Then there are grids:

These are usually found in calendars. The grid needs to be designed so I can move up and down and left to right with my arrow keys. It needs to be able to tell me quickly which dates are available or not available. I personally prefer calendars that only show available dates and times. This allows me to spend much less time determining what is available. Calendly[4] is now doing a great job with this. When someone sends a scheduling link, I can tab through the page, but it will only stop my cursor on days that are available. Once I select a day, the system repeats for available times.

Then there are captchas:

You wouldn't think these things would still be a problem in 2023 or almost 2024. And I'm told by my sighted friends that they have just as much trouble solving security questions as I do. It's great when people use the Google-affiliated one because then, usually, all I have to do is click the check. Sometimes, after I click the check, I still have to solve a security question. But there is always an audio option. The non-visual option doesn't have to be audio. You could also have a visitor check boxes based on similar shapes, colors, countries, etc., as long as the images have alternative text attached to them. You could even use something like an email confirmation, as many email subscription forms employ, to make sure you want to sign up for that mailing list. Another favorite non-visual captcha option of mine is the simple match question or the question that says leave this blank if you are a human. **An additional thought on captchas:** Sometimes, when I have trouble solving a security question, I go to a site's contact form to request help, only to find that their contact form has the exact same captcha. I hope you aren't doing that. And if you are, now you know how to improve it. The easier we make it for people to reach us to address problems, the earlier we can solve them, avoiding frustration and anger that lead to less productive results.

Next up are photos:

I know I touched on this in my article about *done for you*; quite often, submitting images is a key part of completing a form. Now, one way to avoid this is to use a Google form and give the user the option of having the form access their profile. Then, the image is already there. I have had someone help me upload it, so it looks good too. Making the process of selecting photos as easy as possible is important. Here, instructions are important again. You want to tell us what formats to use and what sizes are permitted in dimensions or storage space. And a notification that our photo has been uploaded is appreciated. I often have to go back and forth using tab and shift-tab until I hear the announcement of the file name from my image.

If the image's display on a site is critical, please consider offering the option of having the user email you the image. Or, suggest they upload the photo and click a box or mention in a notes section that they need help editing their image for display purposes. I have some 'what's your excuse' merchandise on a community site for authors. The site invites me to email images instead of trying to post them myself, and they will also write the alt text description for them when being added to the site. This reminds me: please be sure to include a space to add alt text. Especially if the submissions will become profiles or some other content on your website—by having a space to add alt text, you can hopefully get most of your sighted users to add their own descriptions. However, I want you to know that many

4 https://calendly.com

will simply say it's their company logo, headshot, or product name.

Now, assuming that you have done everything right in the information-gathering section of a form, the next step will be to make purchases. Ensuring that the checkout section is accessible is vital. You want it to be as explanatory as possible while being highly accessible. This part of the process results in the highest fear for a user that is disabled. Did I select the right products? Is the final price what I thought it was? Are there any fees I missed? Are there add-ons that can drastically increase my final cost? Am I going to be able to return this if necessary? And, if so, what will I have to do to make that happen? This is quite often a place in a form where people will use unlabelled images. You want to create confidence in decisions made by customers who happen to have a disability.

When entering credit card or third-party payment option information, like PayPal, you want this area to be really stable. Too often, when I get to the checkout, the screen will jump around as I'm trying to enter my card info and verify it before completing my transaction.

Another aspect of the checkout process is entering your shipping address if you purchase physical items. To me, the best way to handle this is to have a button that lets me choose to use my mailing address as my shipping address when both are the same. Entering information once instead of twice helps to avoid mistakes, which is helpful to both the buyer and the seller.

Again, you want a very clear confirmation that an order has been placed, both on the screen and via email. Additionally, the email should detail what has been ordered, the cost, and the final bill, including which payment method was billed.

I understand that most of you will not design checkout or purchasing systems from scratch. Most will use a code provided by the credit card processing company or third-party payment options, like PayPal, Stripe, etc. But in most cases, you can determine how those third-party apps will behave. So, you want to choose the options that will make their work as accessible as possible before including it on your site.

Trust me, if someone has a problem with a purchase, they will come to you before going to the credit card company. And if they go to the credit card company or payment processing company, then you run the risk of a confused customer becoming one who cancels an order or files a fraud report.

Like I said in the beginning. We want our forms to be a big, beautiful, smooth, high-speed bridge to doing business with your site. The point still holds even if the site's sole purpose is information collection or content sharing.

Creating an accessible form will help you devise one that is easier for all your customers or potential buyers to use. It needs to be simple in design, with limited elements per page. It must address buttons, checkboxes, select boxes, grids, and security questions. It needs a way to contact you should the customer need help completing your form. When your form is designed for e-commerce, you should verify that any third-party sites or plug-ins you use are optimized and tested for accessibility. And finally, you should test everything. I'll speak to the subject of how to best do that in a future post. For now, let me just say that I have made it my mission to become familiar with the needs of people with disabilities, which includes outside the arena of vision loss. I have been an accessibility advisor for years; I am currently doing quality reviews for Audio Eye. I would be more than happy to check out your forms and give you some quick feedback. If necessary, I would be willing to do a formal review of your website or any aspect of it that concerns you. And I welcome follow-up questions on this topic. I always tell my friends that we can't expect website and app developers to consider our needs if we aren't willing to tell them what our needs are and why they exist. I hope this post leads to more conversation around forms and other common processes of most websites. Thanks, 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

@phptek

@phptek@phparch.social

https://tek.phparch.com

PHP[TEK] 2024

CONNECT WITH THE COMMUNITY, LEVEL UP YOUR SKILLS, LEARN THE LATEST TRENDS, AND MAYBE EVEN FIND YOUR NEXT JOB. RESERVE YOUR SPOT TODAY.

PHP[TEK] 2024
16TH ANNUAL PHP[TEK] CONFERENCE FOCUSED ON PHP AND WEB DEVELOPMENT.

HTTPS://TEK.PHPARCH.COM/

Is Your Code Encapsulated Enough To Be Clear?

Christopher Miller

In this journey, we won't merely scratch the surface. We'll dive headfirst into the world of encapsulation—a fundamental concept within PHP development. Encapsulation is your partner in crafting code that is not only clear but also highly maintainable and scalable within the PHP ecosystem.

In the ever-evolving realm of PHP development, clarity and maintainability stand as unwavering pillars upon which successful projects are built. Whether you're just setting foot on your PHP journey or, like me, have traversed its landscapes for over two decades, one thing remains indisputable: the importance of code that not only functions but also communicates its logic with crystal-clear precision. Crafting PHP code that achieves its objectives and does so with elegance and lucidity is an art that transcends coding paradigms.

Guiding you through the intricacies of code encapsulation, dissecting its principles, exploring its benefits, and delving into practical implementation. Together, we'll unravel the secrets of well-encapsulated PHP code and understand how it profoundly influences PHP software development.

Whether you're a budding programmer eager to refine your coding skills or a seasoned expert aiming to fortify your understanding, this article caters exclusively to PHP enthusiasts. It's a tribute to the art of crafting PHP code that functions efficiently and narrates a compelling story of logic, elegance, and clarity rooted in PHP's unique ecosystem.

So, fasten your seat belts as we embark on a journey to answer a pivotal question: Is your code encapsulated enough to be clear?

Understanding Code Encapsulation

In the dynamic world of PHP development, understanding code encapsulation is akin to wielding a powerful tool for crafting clear, maintainable, and efficient applications. Code encapsulation is not just a buzzword but a fundamental concept deeply rooted in Object-Oriented Programming (OOP) principles. It's the practice of bundling data and the methods that operate on that data into a single unit—a class in PHP's context.

Let's start by exploring the core principles of code encapsulation in PHP:

1. Classes and Objects: In PHP, code encapsulation primarily revolves around classes and objects. A class serves as a blueprint for objects, defining their structure and behavior. You create a logical and self-contained unit by encapsulating data and methods within classes. (See Listing 1)

In this example, we've encapsulated the model property and its accessor methods within the `Car` class, ensuring data integrity and access control.

2. Access Modifiers: PHP offers access modifiers like public, private, and protected to control the visibility of class members. Encapsulation often involves making data properties private and providing controlled access through methods. (See Listing 2)

Listing 1.

```
1. class Car {
2.
3.     private $model;
4.
5.     public function setModel($model) {
6.         $this->model = $model;
7.     }
8.
9.     public function getModel() {
10.        return $this->model;
11.    }
12. }
```

Listing 2.

```
1. class BankAccount {
2.
3.     private $balance = 0;
4.
5.     public function deposit($amount) {
6.         if ($amount > 0) {
7.             $this->balance += $amount;
8.         }
9.     }
10.
11.    public function getBalance() {
12.        return $this->balance;
13.    }
14. }
```

Here, the balance property is encapsulated as private, allowing controlled access via the `deposit` and `getBalance` methods. Interestingly, it appears our Bank Account is a bad implementation—we can only add money if we have money—an interesting approach; I might build it one day to try it out.

3. Encapsulation and Abstraction: Encapsulation goes hand in hand with abstraction. Abstraction involves hiding complex implementation details and only exposing what's necessary. In PHP, this means encapsulating the underlying complexity and providing simplified interfaces.

(See Listing 3)

In this example, the `Database` class encapsulates the intricate database connection and query logic, exposing only the essential methods for interaction. I'm sure you understand this is just for examples, but just in case, don't set up your database stuff like this! There's little to no control here!

Understanding code encapsulation in PHP is the foundation for building robust and maintainable applications. By embracing OOP principles and leveraging PHP's class-based structure, you empower yourself to write code that not only functions but does so with clarity and efficiency. In the following sections, we'll delve deeper into the benefits of code encapsulation in PHP and practical examples of its implementation within PHP applications.

Benefits of Code Encapsulation

Understanding the significance of code encapsulation in PHP extends beyond mere theoretical knowledge; it directly impacts the quality and maintainability of your PHP applications. Here, we delve into the tangible benefits that encapsulation brings to PHP development, illustrated with real-world code examples.

1. Improved Code Organization: Code encapsulation promotes a structured organization of your PHP codebase. You create modular and organized components by encapsulating related data and methods within classes. (See Listing 4)

In this PHP class `Product`, data and methods related to a product's properties are neatly encapsulated within a single unit, enhancing code organization. Of course, there can be some great improvements to this—but this is again, as with everything else, showing you the encapsulation in a simple way—it would probably look more like this if I were to do it for a full setup: (See Listing 5)

2. Enhanced Code Reusability: PHP code encapsulated within classes can be reused across your application or even in other projects. This reusability simplifies development and ensures consistent behavior. (See Listing 6 on the next page)

The `Logger` class can be easily reused throughout your PHP application or across multiple projects to maintain a consistent logging mechanism.

3. Easier Maintenance and Debugging: Encapsulated PHP code is easier to maintain and debug. When an issue arises, you can focus on the encapsulated unit without sifting through the entire codebase.

Listing 3.

```php
class Database {
    // Private implementation details

    public function connect() {
        // Implementation details hidden
    }

    public function query($sql) {
        // Implementation details hidden
    }
}
```

Listing 4.

```php
class Product {
    private $name;
    private $price;

    public function __construct($name, $price) {
        $this->name = $name;
        $this->price = $price;
    }

    public function displayInfo() {
        return "Product: {$this->name}, " .
               "Price: {$this->price}";
    }
}
```

Listing 5.

```php
class Product {

    public function __construct(
        private ?string $name = null,
        private ?int $price = null
    ): void {}

    public function setName(string $name): void {
    $this->name = $name;
    }

    public function setPrice(int $price): void {
        $this->price = $price;
    }

  public function getName(): ?string {
        return $this->name;
    }

    public function getPrice(): ?int {
        return $this->price;
    }

    public function getDisplayInfo(): string {
        return "Product: {$this->getName()}, " .
               "Price: {$this->getPrice()}";
    }
}
```

Imagine, if you will, a 3000 line "god function" that does everything—in that function, you have an error that's not giving you anything useful about where it is, and it's a fatal error. It could take you HOURS to find it.

Now imagine that same 3000-line function, broken into 20 smaller classes, with a few methods—it becomes a breeze to find out where the error is happening in most cases.

In the PHP development landscape, code encapsulation isn't just a theoretical concept; it's a practical approach to building maintainable, scalable, and efficient applications. By embracing encapsulation and its benefits in PHP, you pave the way for code that is clear and organized, resilient to change, and adaptable to evolving requirements. In the subsequent sections, we'll delve into practical examples of implementing code encapsulation within PHP, showcasing its real-world applications.

Principles of Code Encapsulation in Php

To harness the full power of code encapsulation in PHP, it's essential to grasp the underlying principles that guide this practice. In this section, we'll explore the fundamental principles of code encapsulation within the PHP context and illustrate them with concrete code examples.

1. **Data Hiding with Private Properties**

One of the core principles of code encapsulation in PHP is data hiding. By marking properties as private, you restrict direct access to these properties from outside the class, enforcing controlled access through methods.

2. Abstraction with Methods

Encapsulation in PHP promotes abstraction, which means exposing only essential functionality while hiding complex implementation details. Methods serve as the interface to interact with encapsulated data.

3. Encapsulation of Behavior

Code encapsulation in PHP doesn't solely apply to data but also to behavior. Methods within a class encapsulate specific functionality, allowing you to organize and modularize code effectively.

4. Access Modifiers for Control

PHP provides access modifiers like `public`, `private`, and `protected` to control the visibility of class members. These modifiers play a crucial role in fine-tuning encapsulation.

Understanding these principles of code encapsulation in PHP empowers you to craft well-structured and maintainable PHP code. By enforcing data hiding, promoting abstraction, encapsulating behavior, and leveraging access modifiers, you lay the foundation for PHP applications that are not only clear and organized but also robust and adaptable. In the next sections, we'll delve into practical PHP examples and explore how these principles translate into real-world coding scenarios.

Code Encapsulation in Practice with PHP

In this section, we'll take a hands-on approach and explore real-world code encapsulation scenarios in PHP. By examining practical examples, you'll gain a deeper understanding of how to implement code encapsulation effectively within your PHP applications.

- *1. Encapsulating User Data**:

One common use case for code encapsulation in PHP is managing user data. By encapsulating user-related properties and methods within a class, you create a structured and secure way to handle user information.

Imagine, if you will, a setup where you don't have any control over the user information. All you do is use a `$_SESSION` to hold your user details: (See Listing 7)

Listing 6.

```php
class Logger {
    public function log(string $message) {
        // Log message to a file or database
    }
}

// In different parts of your PHP application or projects
$logger = new Logger();

$logger->log("Error: Something went wrong.");
```

Listing 7.

```php
session_start();

// Suppose we want to store user information
$_SESSION['user_id'] = 1;
$_SESSION['user_name'] = 'Chris';
$_SESSION['user_email'] = 'chris@example.com';
$_SESSION['user_role'] = 'admin';

// Later in the code, we may update user data directly
$_SESSION['user_role'] = 'user';

// Retrieve user data directly from $_SESSION
$user_id = $_SESSION['user_id'];
$user_name = $_SESSION['user_name'];
$user_email = $_SESSION['user_email'];

// Deleting user data when it's no longer needed
unset($_SESSION['user_id']);
unset($_SESSION['user_name']);
unset($_SESSION['user_email']);
unset($_SESSION['user_role']);

// Destroy the entire session when the user logs out
session_destroy();
```

Now whilst this doesn't seem too bad, it's a relatively simple session setup—what about if it gets more complex? That kind of engineering feels wrong in PHP, right?

Let's start by building out a bit of a better system.

```
class User {
    public int $id;
    public string $name;
    public string $email;
    public string $role;
}
```

Sure, this is better because we have a class we can serialize into our session, but let's improve on it for usability and encapsulation. (See Listing 8)

So now we've encapsulated not too bad, but let's go a step further: (See Listing 9)

This gives us decent isolation of session and user, but let's encapsulate the authentication better: (See Listing 10)

Now, yes, we have more code, but it's encapsulated better and heading in the right direction. Obviously, there would be more to do around security and management of session data, but this was a good example of encapsulation.

In the realm of PHP development, code readability and code encapsulation are inseparable companions. In this section, we'll explore how these two concepts intersect and why they play a pivotal role in creating PHP code that is not only functional but also easily understood and maintained.

Listing 8.

```
<?php
session_start();

class User {
    private $userId;
    private $userName;
    private $userEmail;
    private $userRole;

    public function __construct() {
        // Check if user is already logged in
        if (isset($_SESSION['user_id'])) {
            $this->userId = $_SESSION['user_id'];
            $this->userName = $_SESSION['user_name'];
            $this->userEmail = $_SESSION['user_email'];
            $this->userRole = $_SESSION['user_role'];
        }
    }

    public function login($id, $name, $email, $role) {
    // Store user data in session and class properties
        $_SESSION['user_id'] = $id;
        $_SESSION['user_name'] = $name;
        $_SESSION['user_email'] = $email;
        $_SESSION['user_role'] = $role;

        $this->userId = $id;
        $this->userName = $name;
        $this->userEmail = $email;
        $this->userRole = $role;
    }

    public function logout() {
        // Clear session data and class properties
        unset($_SESSION['user_id']);
        unset($_SESSION['user_name']);
        unset($_SESSION['user_email']);
        unset($_SESSION['user_role']);

        $this->userId = null;
```

Listing 8. (cont.)

```
        $this->userName = null;
        $this->userEmail = null;
        $this->userRole = null;
    }

    public function getUserId() {
        return $this->userId;
    }

    public function getUserName() {
        return $this->userName;
    }

    public function getUserEmail() {
        return $this->userEmail;
    }

    public function getUserRole() {
        return $this->userRole;
    }
}

// Example usage:
$user = new User();

// Log in a user
$user->login(1, 'Chris', 'chris@example.com', 'admin');

// Access user data
echo 'User ID: ' . $user->getUserId() . '<br>';
echo 'User Name: ' . $user->getUserName() . '<br>';
echo 'User Email: ' . $user->getUserEmail() . '<br>';
echo 'User Role: ' . $user->getUserRole() . '<br>';

// Log out the user
$user->logout();

// After logout, accessing user data should return null
echo 'User ID after logout: ' .
        $user->getUserId() . '<br>';
?>
```

Listing 9.

```php
<?php
session_start();

class Session {
    public function __construct() {
        session_start();
    }

    public function destroy() {
        session_destroy();
    }

    public function set($key, $value) {
        $_SESSION[$key] = $value;
    }

    public function get($key) {
        return $_SESSION[$key] ?? null;
    }

    public function remove($key) {
        unset($_SESSION[$key]);
    }
}

class User {
    private $userId;
    private $userName;
    private $userEmail;
    private $userRole;

    public function __construct($session) {
        // Check if user is already logged in
        if ($session->get('user_id')) {
          $this->userId = $session->get('user_id');
          $this->userName = $session->get('user_name');
          $this->userEmail =
                          $session->get('user_email');
          $this->userRole = $session->get('user_role');
        }
    }

    public function login(
        $session, $id, $name, $email, $role
    ) {
    // Store user data in session and class properties
        $session->set('user_id', $id);
        $session->set('user_name', $name);
        $session->set('user_email', $email);
        $session->set('user_role', $role);

        $this->userId = $id;
        $this->userName = $name;
        $this->userEmail = $email;
        $this->userRole = $role;
    }

    public function logout($session) {
        // Clear session data and class properties
        $session->remove('user_id');
        $session->remove('user_name');
        $session->remove('user_email');
        $session->remove('user_role');

        $this->userId = null;
        $this->userName = null;
        $this->userEmail = null;
        $this->userRole = null;
    }

    public function getUserId() {
        return $this->userId;
    }

    public function getUserName() {
        return $this->userName;
    }

    public function getUserEmail() {
        return $this->userEmail;
    }

    public function getUserRole() {
        return $this->userRole;
    }
}

// Create a session instance
$session = new Session();

// Create a user instance with access to the session
$user = new User($session);

// Example usage:
if (!$user->getUserId()) {
    // Log in a user if not already logged in
    $user->login($session, 1, 'Chris',
                    'chris@example.com', 'admin');
}
```

Code encapsulation in PHP naturally promotes a clear and organized code structure. The code's structure becomes self-explanatory when data and behavior are encapsulated within classes. This inherent organization enhances code readability, allowing developers to quickly grasp the purpose and functionality of different parts of the codebase.

Encapsulated PHP classes act as self-contained units of functionality. This characteristic simplifies code readability by isolating specific tasks and responsibilities within individual classes.

In collaborative PHP development projects, encapsulation fosters effective teamwork. Each class encapsulates a specific aspect of the application, allowing different team members to work on separate components simultaneously. This parallel development is facilitated by the clarity and readability of encapsulated code.

Listing 10.

```php
<?php
session_start();

class Session {
    public function __construct() {
        session_start();
    }

    public function destroy() {
        session_destroy();
    }

    public function set($key, $value) {
        $_SESSION[$key] = $value;
    }

    public function get($key) {
        return $_SESSION[$key] ?? null;
    }

    public function remove($key) {
        unset($_SESSION[$key]);
    }
}

class User {
    private $userId;
    private $userName;
    private $userEmail;
    private $userRole;

    public function __construct($session) {
        // Check if user is already logged in
        if ($session->get('user_id')) {
          $this->userId = $session->get('user_id');
          $this->userName = $session->get('user_name');
          $this->userEmail =
                        $session->get('user_email');
          $this->userRole = $session->get('user_role');
        }
    }

    public function getUserId() {
        return $this->userId;
    }

    public function getUserName() {
        return $this->userName;
    }

    public function getUserEmail() {
        return $this->userEmail;
    }

    public function getUserRole() {
        return $this->userRole;
    }
}

class Authentication {
    public function login(
        $session, $id, $name, $email, $role
    ) {
        // Store user data in session
        $session->set('user_id', $id);
        $session->set('user_name', $name);
        $session->set('user_email', $email);
        $session->set('user_role', $role);
    }

    public function logout($session) {
        // Clear session data
        $session->remove('user_id');
        $session->remove('user_name');
        $session->remove('user_email');
        $session->remove('user_role');
    }
}

// Create a session instance
$session = new Session();

// Create an authentication instance
$authentication = new Authentication();

// Create a user instance with access to the session
$user = new User($session);

// Example usage:
if (!$user->getUserId()) {
    // Log in a user if not already logged in
    $authentication->login($session, 1, 'Chris',
                    'chris@example.com', 'admin');
}

// Access user data
echo 'User ID: ' . $user->getUserId() . '<br>';
echo 'User Name: ' . $user->getUserName() . '<br>';
echo 'User Email: ' . $user->getUserEmail() . '<br>';
echo 'User Role: ' . $user->getUserRole() . '<br>';

// Log out the user
$authentication->logout($session);

// After logout, accessing user data should return null
echo 'User ID after logout: ' .
                $user->getUserId() . '<br>';

// Destroy the session
$session->destroy();
?>
```

Readable code reduces the cognitive load on developers. Encapsulation simplifies understanding and troubleshooting because each class has a well-defined purpose and encapsulated behavior. Developers can focus on specific classes when debugging or extending functionality without the need to comprehend the entire codebase.

In PHP development, code readability and encapsulation are not mere ideals but practical necessities. The union between them results in PHP code that is not only clear but also maintainable, collaborative, and adaptable. As we progress through this article, we'll further explore how code encapsulation enhances PHP code optimization, making it both efficient and readable.

Code Optimization and Encapsulation in PHP

Code encapsulation in PHP is not only about improving readability and maintainability but also a pathway to code optimization. In this section, we'll explore how well-encapsulated PHP code can lead to optimized, efficient, and high-performing applications backed by practical examples.

1. Modular and Efficient Design

One of the immediate benefits of code encapsulation in PHP is the creation of modular and efficient code units. Encapsulating related data and methods within classes naturally promotes a more organized and efficient code structure. (See Listing 11)

Listing 11.

```
1. class Calculator {
2.     public function add($a, $b) {
3.         return $a + $b;
4.     }
5.
6.     public function subtract($a, $b) {
7.         return $a - $b;
8.     }
9. }
```

In this PHP class, the encapsulated methods `add` and `subtract` create a modular and efficient design for performing basic arithmetic operations. This approach allows you to optimize and extend each method independently.

2. Code Reusability

Code encapsulation facilitates code reusability in PHP. Encapsulated classes and methods can be reused across your application or even in other projects, reducing code duplication and promoting efficient use of resources. (See Listing 12)

Listing 12.

```
1. class FileHandler {
2.     public function read($filename) {
3.         // Encapsulated file reading logic
4.     }
5.
6.     public function write($filename, $data) {
7.         // Encapsulated file writing logic
8.     }
9. }
```

The `FileHandler` class encapsulates file reading and writing logic, making it reusable across various parts of your PHP application or in different projects.

3. Performance Optimization

Well-encapsulated PHP code often leads to performance optimization. By encapsulating data and operations, you can implement efficient algorithms and data structures within classes, because you're isolating to a single action.

4. Reduced Code Coupling

Code encapsulation in PHP promotes reduced code coupling, where classes and modules are loosely connected. This decoupling enhances code maintainability and facilitates easier performance optimization.

Code encapsulation in PHP is not just a practice for enhancing code readability but also a pathway to creating high-performance applications. The modular and efficient design, code reusability, and performance optimization benefits of encapsulation make it a powerful tool in your PHP development arsenal. As we continue, we'll explore common pitfalls in code encapsulation and how to avoid them, ensuring that your PHP code remains both efficient and maintainable.

Common Pitfalls and How to Avoid Them in Code Encapsulation in PHP

While code encapsulation in PHP offers numerous benefits, there are common pitfalls that developers may encounter. In this section, we'll explore these pitfalls and provide guidance on how to avoid them, ensuring that your PHP code remains efficient and maintainable.

Over-Encapsulation

One common pitfall is over-encapsulation, where developers create excessive classes and methods for minor functionalities. Doing so can lead to an overly complex codebase. Prioritize encapsulation for significant, related functionalities. Avoid creating small, highly specialized classes unless they genuinely improve code organization and readability.

Lack of Encapsulation

On the flip side, some developers may neglect code encapsulation, resulting in sprawling and disorganized code that is challenging to maintain. Embrace encapsulation for data and behavior that logically belong together. Encapsulate related functionality within classes to promote organization and maintainability.

Insufficient Documentation

Failure to document encapsulated code can hinder collaboration and future maintenance. It's essential to provide clear documentation for your encapsulated classes and methods. Include inline comments and PHPDoc annotations to describe the purpose and usage of encapsulated classes and methods. Document input parameters, return values, and any exceptions thrown. Although your functions should control

the types and return values—it will be very helpful to have the exceptions

Tight Coupling

Overly tight coupling between classes can be a pitfall. When one class relies heavily on the internal details of another, changes in one class can have unintended consequences on others. Use dependency injection and interfaces to decouple classes. Encapsulate dependencies and use constructor injection to make classes more flexible and easier to maintain.

Neglecting Unit Testing

Failure to write unit tests for encapsulated code can lead to undetected bugs and make it challenging to validate changes in the future. Implement unit tests for encapsulated classes and methods. Test various scenarios to ensure encapsulated functionality works as expected and continues to do so after future modifications.

Ignoring Performance Considerations

While encapsulation promotes maintainability, it's essential to consider performance implications. Overly encapsulated code may introduce unnecessary overhead—balance encapsulation with performance requirements. Use profiling tools to identify bottlenecks and optimize critical parts of your codebase while preserving encapsulation where it matters most.

By being mindful of these common pitfalls and taking proactive steps to avoid them, you can ensure that your code encapsulation practices in PHP lead to maintainable, efficient, and robust applications. As we conclude this article, we'll reflect on emerging trends in code encapsulation and their implications for the future of PHP development.

Emerging Trends in Code Encapsulation for the Future of PHP

The world of PHP development is continually evolving, and code encapsulation is no exception. In this final section, we'll take a glimpse into emerging trends in code encapsulation practices and how they are shaping the future of PHP development.

Microservices and Encapsulation

Microservices architecture, gaining popularity in recent years, emphasizes encapsulating business functionality into small, independent services. Each microservice encapsulates a specific domain or feature, fostering modularity and scalability.

PHP developers are adopting microservices, encapsulating code into discrete units that communicate through APIs. This trend promotes code encapsulation on a macro level, creating highly modular and maintainable PHP applications.

Containerization and Encapsulation

Containerization technologies like Docker encapsulate entire environments, including PHP applications and dependencies, into portable containers. This approach enhances consistency and reproducibility.

PHP developers are leveraging containerization to encapsulate PHP applications, ensuring that code runs consistently across different environments. This trend streamlines deployment and maintenance while enforcing encapsulation.

Serverless Computing and Encapsulation

Serverless computing abstracts server management, allowing developers to encapsulate code into functions or serverless components. This trend simplifies infrastructure concerns and promotes code modularity.

PHP developers are exploring serverless platforms, encapsulating code into serverless functions that scale automatically. This approach enables fine-grained encapsulation and cost-efficient execution of PHP code.

Stronger Type Systems

Modern PHP versions introduce stronger type systems and static analysis tools. Encapsulation practices are adapting to leverage type hints and enforce strict typing, reducing runtime errors.

PHP developers are embracing stricter typing to enhance encapsulation. They create more robust and self-documenting code by specifying data types in method signatures and class properties.

Code Generators and Annotations

Code generators and annotations are emerging as tools to automate code encapsulation tasks, such as generating boilerplate code or configuring aspects of encapsulated components.

PHP developers are exploring code generation and annotation libraries to streamline encapsulation tasks. These tools can reduce repetitive work and enhance code consistency.

As PHP development continues to evolve, embracing these emerging trends in code encapsulation can lead to more efficient, maintainable, and scalable PHP applications. By staying informed about these developments, PHP developers like yourself can adapt and harness the power of encapsulation to meet the challenges of the future.

Conclusion

In the ever-evolving realm of PHP development, the practice of code encapsulation stands as a beacon of clarity, maintainability, and efficiency. From the early days of PHP to the dynamic landscape we navigate today, the principles of encapsulation have remained steadfast, shaping the way we craft PHP applications.

Throughout this comprehensive exploration, we've embarked on a journey through the nuances of code encapsulation in PHP. We've discussed its fundamental principles, its benefits to PHP development, and practical examples illuminating its real-world applications. We've delved into the symbiotic relationship between code encapsulation and code readability, showcasing how they work in harmony to create exceptional PHP code.

We've also uncovered the power of code encapsulation to drive code optimization, making PHP applications not only clear but also high-performing. By exploring common pitfalls and strategies to avoid them, we've equipped

ourselves to navigate the challenges that come with encapsulating code effectively.

As we've looked ahead to emerging trends, we've seen how code encapsulation continues to evolve. Microservices, containerization, serverless computing, stronger type systems, and automation tools are reshaping the landscape of PHP development. Embracing these trends ensures that PHP remains a vibrant and relevant language in the ever-changing world of software development.

As you, fellow PHP developers, continue your journeys in the world of PHP, remember that code encapsulation is not just a practice; it's a philosophy. It's the art of crafting code that not only performs its tasks but does so with elegance, clarity, and efficiency. It's a testament to our dedication to creating PHP applications that stand the test of time.

So, whether you're embarking on your PHP journey, have traversed its paths for years, or are adapting to the winds of change, embrace the power of code encapsulation in PHP. Let it be your guiding light, your tool for creating code that not only functions but also tells a story—a story of logic, elegance, and clarity, in the dynamic world of PHP development.

Thank you for joining me on this journey through the world of code encapsulation in PHP. May your PHP code always encapsulate the brilliance of your ideas and the clarity of your logic. Until we meet again in the realm of PHP development, and look at Is Your Code Tested Enough To Allow You To Be Confident, happy coding, fellow encapsulators.

In 1983, Christopher was introduced to computers by his dad, at the tender age of 3. now, over 40 years later, he has been working in the industry for over 20 years making an impact across multiple sectors of the industry. Starting with launching the first web development company in Staffordshire, Christopher dealt with the web - when the web was little more than just pretty text. He established the websites for many different businesses in their first inception, before moving onto web applications a little while later. Illness prevented Christopher from working in the industry full time for some considerable time - but recovery meant he could tackle once again the joys of code - but he soon found that his skills had become out of date, so thanks to the School of Code in the UK he was able to return to the workplace with revitalised skills ready to tackle the next wave. Specialising since the School of Code in Readable Code, He has worked with a large number of languages, specialising in supporting businesses to grow standards for their code base, and now he is ready to share his processes with the world.
@ccmiller2018

Sunsetting User Groups

Beth Tucker Long

Are you still a member of a user group? If so, is your user group still actively meeting? Many of these groups no longer regularly meet, if they meet at all. Why have these groups faded over the years?

Twenty years ago, growing your coding skills was hard. Libraries and frameworks lacked good documentation. It was hard to hear about new ideas, design patterns, and security tips. Groups of people began congregating to help each other navigate these challenges. It was the birth of the PHP user group community.

Fast forward to ten years ago, and PHP user groups were thriving and making a huge difference in our community. They were a great place to learn about PHP, hear about new ideas, and meet other people using PHP. We wanted everyone to have access to one. We made online maps to help you find them and mentored people to speak at them, and if you made the mistake of talking to me at a conference, you would suddenly find yourself tasked with organizing your own local user group. They were contagious and so much fun!

Yet time marched on, and changes came. Frameworks became more specialized. Good documentation became a standard. Development shifted from one language into a mixture of languages based on what worked best for each portion of the application. Attendance at PHP user groups began to stagnate and then dwindle. Fewer people considered themselves PHP developers because applications now required them to program in multiple languages. Fewer people needed help with PHP because the documentation and standardization had made things clearer and easier. Meanwhile, some of the newer languages were still in that wild, confusing phase, so people began switching to other communities where they needed more help. PHP user groups began disappearing.

And then the pandemic shut them all down.

Many people admirably strove to keep things going, holding video meetings or keeping discussions going online. Online user groups popped up to try to pull in people whose local user groups could not handle the requirements of running virtual meetings. It was a noble goal, and on some level, it helped, but it didn't stop things. User groups keep shutting down.

I have heard people lament that this is proof that PHP is dying. No one wants to come to PHP events anymore. Is this really what is happening? Looking back 20, 10, or even just 5 years ago—I am not using the same operating system, I am not using the same frameworks, and I am definitely not using the same PHP. Why are we trying to use the same user groups?

It's time we sunset the old user group model and spin up a fresh version with updated features and a modern UI. Everything may have changed, but having a community is still an incredibly beneficial part of being a developer. We still need user groups, just in a new way. So what does that look like? I can't wait to find out.

Beth Tucker Long is a developer and owner at Treeline Design[1], a web development company, and runs Exploricon[2], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[3] and Full Stack Madison[4] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 http://www.treelinedesign.com
2 http://www.exploricon.com
3 http://madwebdev.com
4 http://www.fullstackmadison.com