www.phparch.com

March 2021
Volume 20 - Issue 3

php[architect]

LAMBDA PHP

The New LAMP Stack is Serverless

Refactoring—Does This Code Spark Joy?

Fiendish Functions—Filtering Fact From Fiction

Free Sample Article

a php[architect] guide

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

In this book, Michael Akopov shares his experiences creating software solutions that can not only survive but thrive. If you're looking to take a project to the next level, he'll show you how to put it together without burning out. Includes a foreword by Taylor Otwell, the creator of Laravel.

Available in Print+Digital and Digital Editions.

Order Your Copy

phpa.me/beyond-laravel

Fiendish Functions —Filtering Fact From Fiction

Camilo Payan

Functional programming is a paradigm that has gained in popularity recently with the rise of several new functional programming languages and integration of new features in existing languages. All the same, there are many concerns about functional programming in the developer community. Is functional programming useful for the PHP developer? Will it add complexity to my codebase? This article will respond to some of these misconceptions.

To get the most out of this article, I hope you will take stock of what you currently know about functional programming, but complete knowledge isn't necessary. Also, I assume the reader has some knowledge of object-oriented programming such as familiarity with SOLID design principles, and knowledge of PHP functionality such as callables, anonymous functions, and the array methods that take callables, like `array_map()`[1], `array_filter()`[2], and `array_reduce()`[3]. With all that said, let's dive in with the first myth I'd like to dispel.

Functional Programming is the Same as Imperative Programming

The first myth about functional programming is that it is an older pre-object-oriented programming style, where instructions are written one after the other as in a WordPress template. This myth, unfortunately, persists due to a lack of knowledge of the different programming paradigms that exist. Without that knowledge, every paradigm is usually organized in a coder's mind as object-oriented or not object-oriented.

This myth is especially true in the PHP community with good reason. Those who worked in PHP in the early 2000s (and before) didn't have the object-oriented features—interfaces, namespaces, and abstract classes, to name a few—that have organized PHP 5+ projects since, and that led to many projects bogged down in spaghetti code without clear borders between responsibilities. That history leads to the clear delineation of coders' minds between the object-oriented approach and the "bad old way."

However, functional programming is not a return to that old way of doing things. Functional programming takes as its center not the description of objects and their behaviors, but the description of functions and their inputs and outputs. Then those functions are applied and composed to create our system.

The main issue that crops up with code from before PHP 5's release is that it lacked organization, focus, and clarity. Functional programming has an organizing principle that gives it focus and rules that make them explicit.

Functions in functional programming are fundamentally based on functions in mathematics. That is, something like f(x) where some math is done to a given input x, and the same result is returned every time for the same input. This goal in functional programming is known as writing pure functions. A pure function is any function where, given the same input, the same output is returned.

You might already be able to imagine some of the ramifications. Pure functions are necessarily protected from side effects since any reliance on a state outside the function would change the output—or they wouldn't be pure functions. Without side effects, the inputs and outputs themselves must be immutable, that is that the function cannot change the input since this would be a side effect.

For example, if we had a function that took in a date and returned the next date, we might implement something like Listing 1.

Listing 1.

```php
function nextDay(\DateTime $date) {
   $interval = new DateInterval('P1D');
   // Mutate the given date
   return $date->add($interval);
}

$date = new DateTime('August 12, 1791');
nextDay($date);
echo $date->format('F j, Y'); // August 13, 1791
```

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

2 `array_filter()`: *https://php.net/array_filter*

3 `array_reduce()`: *https://php.net/array_reduce*

Sample

However, this example mutates the input since PHP passes objects by reference, which could have ramifications in whatever code was using that date object in the first place. If we were to ensure that this function was pure, we would implement it as in Listing 2.

Here we create a new DateTime object and return the new date. The input is left intact, and the calling code doesn't need to worry about handling a side effect. Actually, PHP has a set of immutable `Date` objects that we could use instead to implement as in Listing 3.

Listing 2.

```php
<?php
function nextDay(\DateTime $date) {
   $interval = new DateInterval('P1D');
   $nextDay = clone $date;
   // Mutate the given date
   return $nextDay->add($interval);
}

$date = new DateTime('August 12, 1791');
$next = nextDay($date);
echo $date->format('F j, Y'); // August 13, 1791
echo $next->format('F j, Y'); // August 14, 1791
```

Listing 3.

```php
function nextDay(DateTimeImmutable $date) {
   $interval = new DateInterval('P1D');
   // DateTimeImmutable makes a new object.
   return $date->add($interval);
}

$date = new DateTimeImmutable('August 12, 1791');
$next = nextDay($date);
echo $date->format('F j, Y'); // August 13, 1791
echo $next->format('F j, Y'); // August 14, 1791
```

Functional Programming is Impossible Without a Functional Language

Functional programming is only possible in a functional programming language. Those that discard functional programming in PHP on these grounds will point out that PHP is not a functional language like Haskell or OCaml. Without the foundation of a functional language, the paradigm is doomed to failure in a PHP context. This myth is believable because it has a grain of truth! Functional programming and its theoretical background in the lambda calculus (which we will not get into in this article) are the foundation of popular functional languages. At the same time, PHP will allow you to mix in bits of the functional programming paradigm as you like.

The issue here is one of definition. Every programming paradigm has vague definitions and fuzzy borders between them. If you ask five developers to define object-oriented programming, you'll get six answers, seven book recommendations, and one intense desire to move to the woods. The same holds true for functional programming. There isn't a strong definition.

We need to take a very loose definition for functional programming in PHP, which boils down to support for first-class lambda functions. We need functions that can be defined, passed around, applied, and composed at will.

"First class functions" refers to the ability to assign a function to a variable. This language feature supports the requirement that we be able to pass around a function. It has had first-class support in PHP since 2009 with the release of PHP 5.3.

```php
// Assign a function to a variable
// This creates a Closure, assigned to $sum
$sum = function($a, $b) {
    return $a + $b;
};

// Call the function, using the variable name
echo $sum(5, 4); // 9
```

A "lambda function" is an anonymous function, which is to say that your function can be defined and used in your code without being given a name. This fulfills the requirement that functions can be defined anywhere in your code and is most commonly used in PHP as callbacks. Callbacks predated PHP 5.3 in some ways, such as `call_user_func()` using named functions. However, they were formalized further as a type in 2009 with 5.3. This also points to the ability to pass a function into another function. Similarly, we can use lambda functions to return a function from a function. See Listing 4.

Listing 4.

```php
$names = ["John", "James"];

array_map(
   // Anonymous Lambda function passed in as parameter.
   function ($n) {
      echo "Hello; $n";
   },
   $names
);
```

While PHP is not a functional programming language, it is a general-purpose language that now supports multiple programming paradigms. We can use and apply functional programming principles to our code while using best practices to decide when to use which paradigm, much like we use experience and best practices to determine when to apply a design pattern.

Sample

Functional Programming Will Ruin My Object-Oriented Codebase

Since PHP 5, there has been a consistent push toward language features in PHP that support the best practices of object-oriented programming. It has led to many solid advances in PHP codebase organization and greater ease of library development. The PHP Framework Interoperability Group has worked to codify best practices in PHP while also supporting the various object-oriented frameworks like Symfony, Laravel, and Laminas.

When considering the functional programming paradigm's advantages, a developer might think that they can't combine the two. That choosing an object-oriented approach means that functional techniques are off the table. Since the world of PHP frameworks is entirely object-oriented, functional programming is a toy idea that we can't use in existing projects. Nothing could be further than the truth. The two approaches to thinking about and writing code can supplement and improve each other.

Some of the most influential ideas around object-oriented programming today are the SOLID design principles. The SOLID design principles are a set of code smells that make object-oriented code challenging to work with and change over time. They are the single responsibility principle, the open/close principle, the Liskov substitution principle, the interface segregation principle, and the dependency inversion principle. Let's go through these and see how a functional approach might help us accomplish their goals.

The Single Responsibility Principle

The idea of the single responsibility principle is that each class we write should have only one reason to change. If a class has more than one responsibility, that change becomes more burdensome. A change to one of your class's responsibilities could affect your class's other responsibilities, making your change more likely to introduce bugs.

Thinking in terms of lambda functions helps the single responsibility principle at the method/function level. Using lambda functions, we can break our code down into its smallest parts while pushing secondary responsibilities out to other objects or methods.

Let's look at some code (Listing 5) that finds the largest product in a series of digits.

We can see that there are several responsibilities mixed here. I build the array of factors from the given series of digits. Then I get the products of each of those factors. These can all be turned into individual functions, as you can see in Listing 6.

Now we've refactored a bit, and each of our refactored functions does only one thing. This solution makes extensive use of PHP's array functions, which are probably the most common use cases you'll find for a functional programming mindset.

Listing 5.

```php
function largestProduct(string $digits, int $len) {
   $factorsArray = [];

   for ($i = 0; $i <= strlen($digits) - $len; $i++) {
      $factorsArray[] = substr($digits, $i, $len);
   }

   $productsArray = [];
   foreach ($factorsArray as $factors) {
     $productsArray[] = array_product(str_split($factors));
   }

   return max($productsArray);
}
```

Listing 6.

```php
function largestProduct(string $digits, int $len) {
   return max(array_map(
      'getProduct',
      getFactors($digits, $len)
   ));
}

function getFactors(string $digits, int $len) {
   return array_map(
      fn ($i) => substr($digits, $i, $len),
      range(0, strlen($digits) - $len)
   );
}

function getProduct(string $factors) {
   return array_product(str_split($factors));
}
```

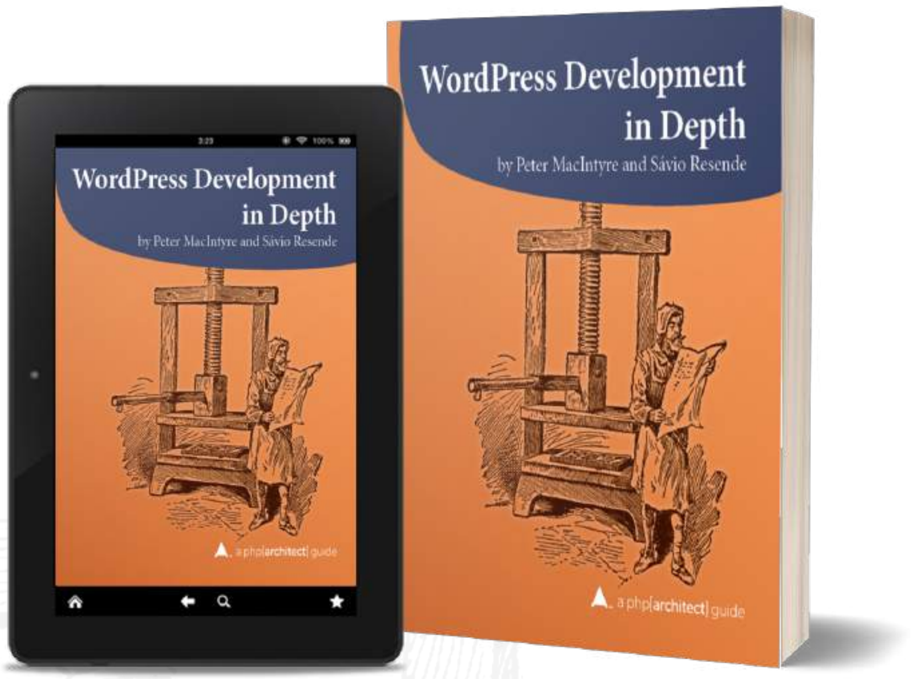
The Open/Closed Principle

The open/closed principle in object-oriented programming dictates that objects should be open for extension but closed for modification. The idea here is to help code changes by avoiding the brittleness that results from codebases where classes depend directly on other concrete classes, so a modification in a class to finish a feature request can result in unforeseen bugs in another one.

In practice, developers take this to mean that classes should depend on abstract classes or interfaces. Instead of needing new implementation details in your class for each concrete class that is supported, your class would rely on an abstraction. Then it could take in concrete classes that extend that abstraction or implement that interface without needing to change. In this way, you extend your class's functionality without needing to be modified to support new feature requirements. However, if we expand our thinking on the open/closed principle to include functional programming concepts, we can see other ways to apply the open/closed principle.

Higher-order functions, that is, functions that can take other functions as inputs or return functions as outputs, are also open for extension while being closed for modification.

Sample


WordPress Development in Depth
by Peter MacIntyre and Sávio Resende
a php[architect] guide

Build a custom, secure, multilingual website with WordPress

WordPress is more than a blogging platform—it powers one-fifth of all sites. Since its release, enthusiastic contributors have pushed the envelope to use it as a platform for building specialized social networks, e-commerce storefronts, business sites, and more. This guide helps you sort through the vast number of plugins available to find the ones that work for you. You'll also learn how to use CSS and PHP code to tailor WordPress to your needs further.

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

Read a sample and purchase your copy at the link below.

Order Your Copy
http://phpa.me/wpdevdepth-book

Some of the best examples of this are already native PHP functions. The suite of array functions that take a function callback is all higher-order functions whose functionality is extended by the function you pass in. At the same time, you also are not able to modify the inner workings of that function.

Immutable objects are, by definition, completely closed to modification after their creation. Their internal state can't be changed, and any method that does make a "change" actually creates a new immutable object. This is pretty far afield from the original definition of the open/closed principle, or even its interpretation as a dependence on abstraction. Still, immutability maintains an essential role in functional programming. Immutability helps support the creation of pure functions that don't have side effects because immutable objects protect the programmer from creating side effects in the objects they're using. In this way, we could see it as a natural alignment of goals under the open/closed principle.

The Liskov Substitution Principle

The Liskov substitution principle means that child classes should maintain the behavior of their parent classes. One of the byproducts of the Liskov substitution principle in current best practices is to avoid the inheritance hierarchies that this principle is meant to help you navigate entirely. You can see this in the proliferation of dependency injection libraries and the frameworks that use them.

Inheritance isn't a part of functional programming. As stated earlier, functional programming is based on function composition: building behavior by combining other behaviors/functions. This is remarkably similar to the idea of dependency injection that programmers have turned to resolve the LSP's demand.

Dependency Inversion Principle

The dependency inversion principle states that abstractions should not depend on details, and details should not depend on abstractions. In the object-oriented paradigm, we take this to mean that classes should separate high-level business logic from glue code by depending on interfaces or contracts instead of on concrete classes.

Again, we can look to higher-order functions, which can take a function as input and apply it. In this case, the interface is the function itself, and the implementation details are in the function. This is possible with first-class functions. The suite of `array_*` functions in the PHP standard library again shows a concrete example, where the implementation details of your filter or map function are left to you. In contrast, the higher-level functionality of applying that function across an array is contained in the standard library function.

How Can Functional Programming Fit Into My Object-Oriented Codebase?

So what does functional programming look like in action in a codebase? Especially in an object-oriented codebase? In a different environment, like the Java virtual machine or the .NET ecosystem, a programmer could drop into a functional language like Scala or F# to implement the parts of their code influenced by functional programming thinking. However, we need a different strategy in PHP since we have features that support functional programming but not a specific tool to do so.

I find inspiration in a concept called "imperative shell, functional core," which I first learned of Gary Bernhardt of "Destroy All Software." There, he describes an approach where the core business logic is written in a functional style. This plan means objects are immutable and behavior is written as pure functions, which take some input and return a new object. These objects are then glued together by a thin shell of imperative code which would interact with the outside world, including user input, APIs, or whatever other concerns you have.

The project he uses to illustrate this is a small Ruby-based Twitter client. What would this look like in a current PHP framework like Laravel? Laravel controller and request classes could hold the imperative shell, while the business logic resides in plain PHP objects.

One side effect of writing objects in a functional style is the positive effect on testing. Since your core business logic is now defined in pure functions, they are testable with unit tests, based only on the inputs and outputs, with little setup required. This could even open up your codebase to using mathematical testing tools such as quickcheck[4], though I have not verified this.

User Interface Programming

User interface programming in the functional programming paradigm takes the form of functional reactive programming (FRP). The idea behind FRP is to move toward a completely event-based model and allowing a user interface to use immutable objects and pure functions to change the UI. We should contrast this to the imperative model of UI development, which mainly takes the form of maintaining state and creating behavior based on that state. FRP requires an event-driven runtime, such as that provided by the ReactPHP library. You can't react to events without events, after all!

If your UI is written in Javascript as a single-page app, more options are open to you, such as RxJS. There are even ways to include FRP methods in React and Vue applications.

Concluding Remarks

Let's review some of the myths that surround functional programming and their responses. Functional programming

4 *quickcheck: https://en.wikipedia.org/wiki/QuickCheck*

Sample

is not the same as imperative programming. Functional programming is structured, just like object-oriented programming, but aims for different philosophical underpinnings. The foundations of functional programming include immutability, pure functions, and the composability of those pure functions.

You do not need to be working in a functional programming language to benefit from a functional programming paradigm. While a language like OCaml or Haskell may force you into functional programming, PHP has the language features to support you. First-class functions, that is, functions that can be assigned to variables and passed around, are available in PHP. Lambda functions are anonymous, nameless functions and are also available in PHP. You may already be using them as callables in array functions or `call_user_func()`.

The best practices of object-oriented programming are not incompatible with functional programming. The SOLID design principles, which form the base of so much object-oriented thought, can be augmented and refined by a functional programming mindset. As the SOLID design principles force you to refine your class definitions to smaller chunks of behavior, you may even find yourself with a class that is a thin wrapper around a single piece of behavior in a method. This isn't so different from a pure function!

As a method of attack for including functional programming in your codebase, I've offered the idea of an imperative shell around a function core of business logic. Not only does this take some of the SOLID design principles much further than you might usually, but it also has positive knock-on effects for unit testing. Developers can even use functional programming in the UI through functional reactive programming.

Hopefully, this walkthrough will help dispel not only some preconceived notions about functional programming but also show how to use functional programming in your existing codebase. As developers, we can often recognize that multiple perspectives sharpen our analysis of a problem and help us create work that solves the problem while being more understandable and maintainable.

Camilo rose from the swamps of South Florida, where he taught the alligators PHP and Vim. He traveled the Sahara with the Tuaregs of Morocco, and found the SOLID design principles etched into glass beneath the sands. He then moved from Miami to Chattanooga, to teach what he could, and receive the wisdom of the ascetic programmers of Appalachia. Whether or not Camilo received a Computer Science degree from Florida International University remains a hotly contested debate among his biographers, critics, and acolytes. @camdotbio

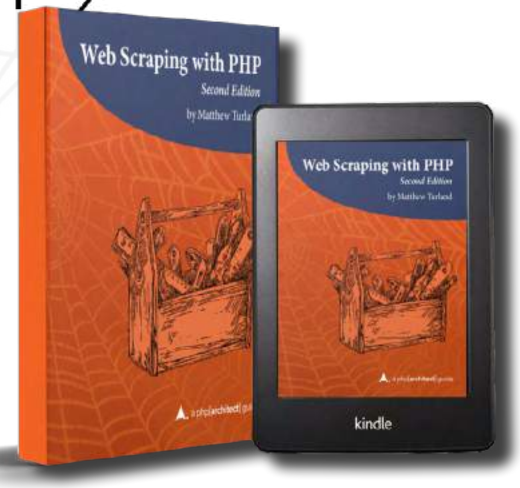
Related Reading

- *A Case for Functional Programming in PHP* by Lochemem Bruno Michael, February 2021. http://phpa.me/3qkebjs
- *Education Station: Calling all Callables* by Chris Tankersley, June 2020. https://phpa.me/education-june-20
- *Removing the Magic with Functional PHP* by David Corona, July 2016. https://www.phparch.com/magazine/2016-2/july/

Sample

What if there's no API?

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



Order Your Copy
http://phpa.me/web-scraping-2ed

a php[architect] print edition

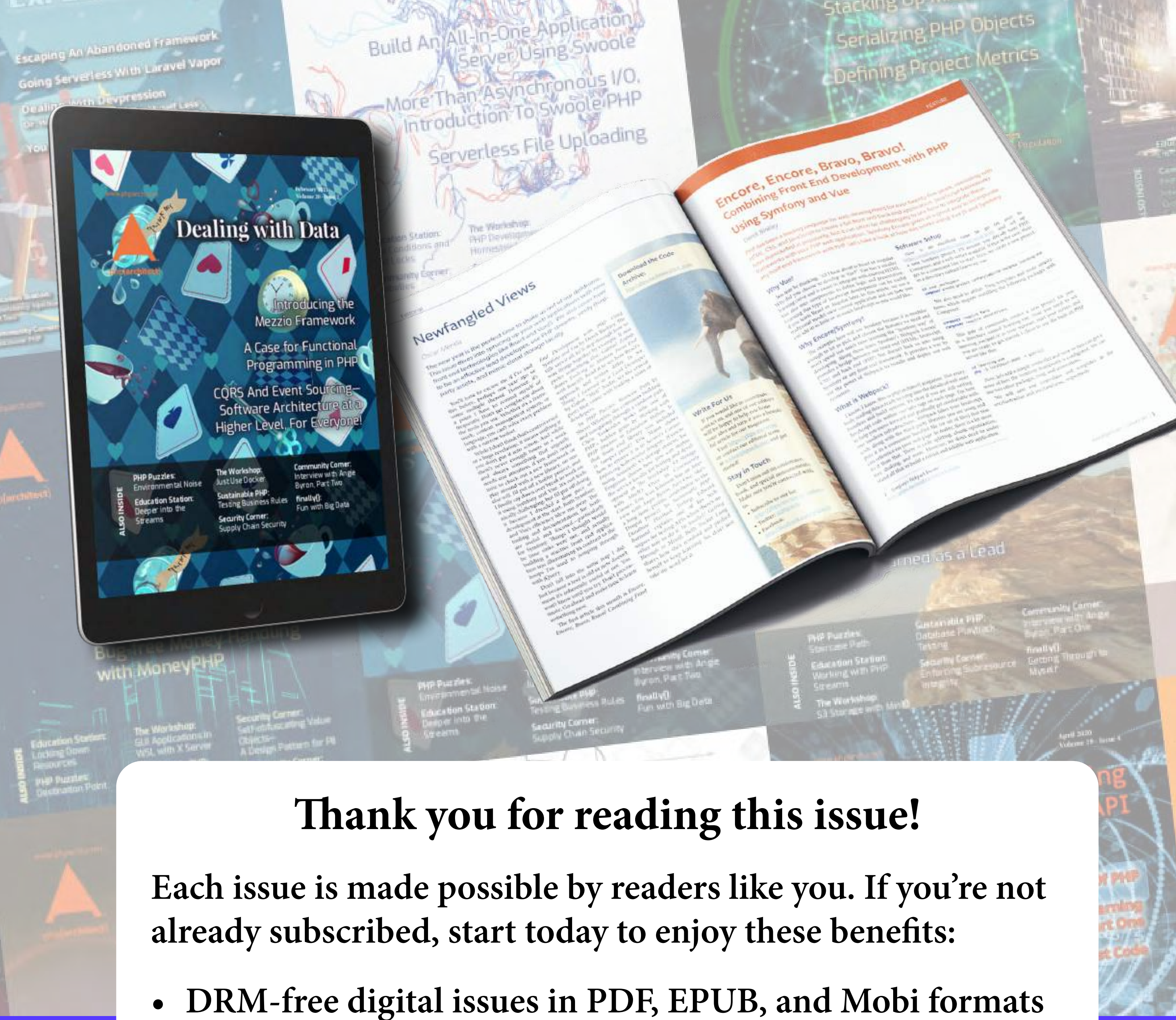
Learn how a Grumpy Programmer approaches testing PHP applications, covering both the technical and core skills you need to learn in order to make testing just a thing you do instead of a thing you struggle with.

The Grumpy Programmer's Guide To Testing PHP Applications by Chris Hartjes (@grmpyprogrammer) provides help for developers who are looking to become more test-centric and reap the benefits of automated testing and related tooling like static analysis and automation.

Available in Print+Digital and Digital Editions.

Order Your Copy

phpa.me/grumpy-testing-book



Thank you for reading this issue!

Each issue is made possible by readers like you. If you're not already subscribed, start today to enjoy these benefits:

- DRM-free digital issues in PDF, EPUB, and Mobi formats
- Discord channel for subscribers.
- Access to all back issues starting in 2002.
- We also have print subscriptions available.

Subscribe or Renew Today
https://phparch.com/magazine