www.phparch.com

June 2020
Volume 19 - Issue 6

php[architect]

Advanced Design & Development

Asynchronous Programming in PHP

Building a REST API from Scratch

Browser APIs: The Unknown Super Heroes

ALSO INSIDE

a php[architect] anthology

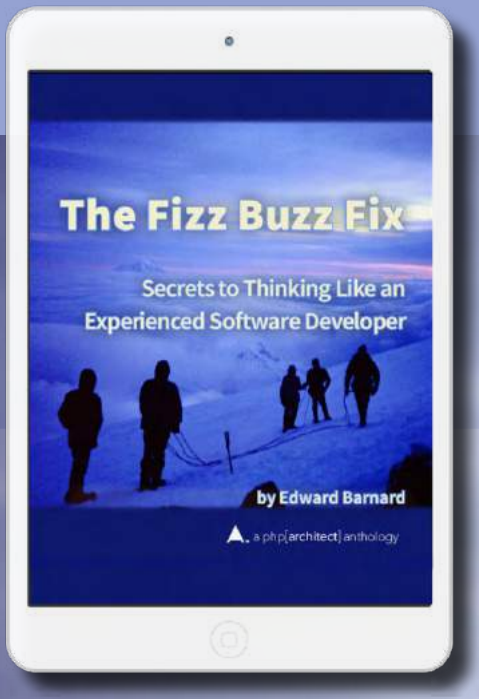
Learn how a Grumpy Programmer approaches testing PHP applications, covering both the technical and core skills you need to learn in order to make testing just a thing you do instead of a thing you struggle with.

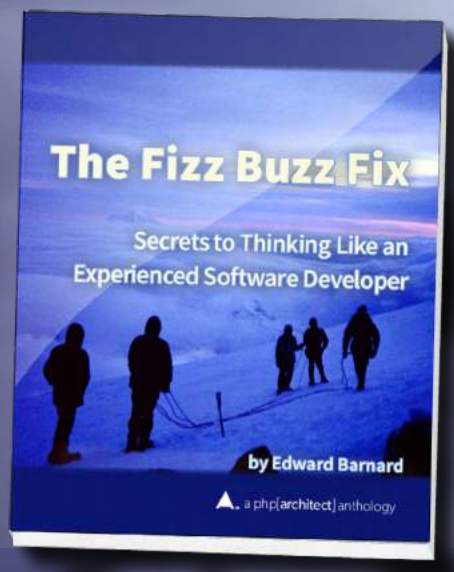
The Grumpy Programmer's Guide To Testing PHP Applications by Chris Hartjes (@ grmpyprogrammer) provides help for developers who are looking to become more test-centric and reap the benefits of automated testing and related tooling like static analysis and automation.

Available in Print+Digital and Digital Editions.

Order Your Copy

phpa.me/grumpy-testing-book





Tackle Any Coding Challenge With Confidence

Companies routinely incorporate coding challenges when screening and hiring new developers. This book teaches the skills and mental processes these challenges target. You won't just learn "how to learn," you'll learn how to think like a computer. These principles are the bedrock of computing and have withstood the test of time.

Coding challenges are problematic but routinely used to screen candidates for software development jobs. This book discusses the historical roots of why they select for a specific kind of programmer. If your next interview includes a coding exercise, this book can help you prepare.

Available in Print, Kindle Unlimited, and Kindle Lending Library

Order Your Copy

http://phpa.me/fizzbuzz-book

Calling all Callables

Chris Tankersley

When facing a challenging problem, you want a flexible codebase that adapts quickly. Object-oriented programming facilitates it by giving you the power through inheritance, encapsulating code in reusable objects, and generally making them work for your application as you see fit. However, we can find flexibility in other programming approaches.

Languages such as JavaScript, which until very recently had a vastly different concept of an object, relied heavily on the idea of Callables and Callbacks. JavaScript uses objects and structures that can be called like functions and encourages a programming paradigm of passing these "Callable" objects around.

As it turns out, PHP has had a similar way of functioning for a very long time. The use of Callables and Callbacks in PHP has grown as the language has taken inspiration from other languages like JavaScript (though I am still on the fence about arrow functions).

Let's take a look at how we can use these ideas in PHP and provide even greater flexibility in our code.

Why?

Every language is different, and those differences help frame the decisions that we make when it comes to writing our code. I love PHP, but there are some things that I miss from other languages as well. Python's decorators, or annotations you can attach to functions that modify their output and invocation, are powerful tools for making code do what I want. Using callables can make your application more adaptable without requiring a full-blown object, or having to anticipate the methods you need to define. You can short-circuit this by setting expectations for what the callable should or should not return.

Listing 1

```
1. <?php
2.
3. function square(int $num) {
4.     return $num * $num;
5. }
6.
7. function cube(int $num) {
8.     return $num * $num * $num;
9. }
10.
11. function addAndModify(int $num1, int $num2, string $modifier) {
12.     $total = $num1 + $num2;
13.     return call_user_func($modifier, $total);
14. }
15.
16. echo addAndModify(1, 2, 'square'); // 9
17. echo addAndModify(1, 2, 'cube'); // 27
```

The Old Way

PHP has supported the general idea of a callback since the PHP 4 days. A callback is another piece of code, typically a function, passed as a parameter to a method or function and is executed after the original code runs. In most circumstances, the original function will pass data into that second function.

The earliest instance of this is the `call_user_func()`[1] function and it's companion, `call_user_func_array()`[2], which are still handy today. `call_user_func()` relies on being passed a function to call as the first argument. Any additional arguments passed to create the function are called as arguments to the new function.

Let's say we need to add two numbers together, and then modify them in some way. We don't know how the numbers will be changed, that is up to the developer. We can write a function that takes the two numbers, and the name of a function to modify the output as in Listing 1.

Here we define two modifier functions—`square()` and `cube()`. Since these are named functions, we can pass their names into our `addAndModify()` function, and use `call_user_func()` to invoke the appropriate method for us. We've made `addAndModify()` more flexible by not hard coding what modification is applied to the numbers we add together. We're not limited to passing in the names of custom functions, either. We can pass in any PHP native function too.

What if you don't have a function, but you have a method on an object? Well, `call_user_func()` can take an array containing a class name and method name as the first parameter. How would that look? See Listing 2.

Instead of passing a function name, in this case, we have passed an array with our class name, `Modifiers`, and the method name we want to call (either `square()` or `cube()`). Functionally this is the same as having two functions, but it does allow us to use encapsulation to structure our code more cleanly.

The keen-eyed among you may ask, "What about classes that have constructor dependencies?" Good catch! There is a third way that you can invoke a method call, and that's

1 `call_user_func()`: https://php.net/function.call-user-func

2 `call_user_func_array()`: https://php.net/function.call-user-func-array

passing an instantiated object. It looks the same, and we pass the object instead of the class name as shown in Listing 3.

The difference is that we instantiate the object we want to call, instead of just passing the name.

When it comes to `call_user_func()`, none of these invocations are better than the other, so use whatever way works with your code.

Listing 2

```
1. <?php
2.
3. class Modifiers
4. {
5.    public function square(int $num) {
6.       return $num * $num;
7.    }
8.
9.    public function cube(int $num) {
10.      return $num * $num * $num;
11.   }
12. }
13.
14. function addAndModify(int $num1, int $num2,
15.                       array $modifier) {
16.   $total = $num1 + $num2;
17.   return call_user_func($modifier, $total);
18. }
19.
20. echo addAndModify(1, 2, [Modifiers::class, 'square']); // 9
21. echo addAndModify(1, 2, [Modifiers::class, 'cube']); // 27
```

Listing 3

```
1. <?php
2.
3. class Modifiers
4. {
5.    protected $baz;
6.
7.    public function __construct(Foo $bar) {
8.       $this->baz = $bar;
9.    }
10.
11.   public function square(int $num) {
12.      return $num * $num;
13.   }
14.
15.   public function cube(int $num) {
16.      return $num * $num * $num;
17.   }
18. }
19.
20. function addAndModify(int $num1, int $num2,
21.                       array $modifier) {
22.   $total = $num1 + $num2;
23.   return call_user_func($modifier, $total);
24. }
25.
26. $modifier = new Modifiers(new Foo());
27. echo addAndModify(1, 2, [$modifier, 'square']); // 9
28. echo addAndModify(1, 2, [$modifier, 'cube']); // 27
```

Anonymous Functions

`call_user_func()` and its associated syntax is nice, but PHP 5.3 took things a step further. It introduced the concept of "Anonymous Functions," which allowed binding a function invocation to a variable or using them for one-time encapsulation. Using this feature, we could now pass around functions like any other variable.

> *OK, I'm lying just a little bit. Technically, PHP 4 had* `create_function()` *which created a randomly named function. You could assign that name to a variable and then invoke that variable. It was inefficient and a security vulnerability due to its internal use of* `eval()`*. Don't use it. Thankfully it's deprecated. On that note, don't use* `eval()` *directly to create a new function either.*

An anonymous function is a function without a concrete name. They can be used any place that accepts a callable construct, so let's go back to our original example and look at how we can make each function a variable (Listing 4).

Instead of having named functions called `increment()` and `decrement()`, we stuffed the functions into appropriately named variables. Unlike a standard function declaration, we have the = assignment operator, and we don't give the function a name. We go straight from the `function` keyword to defining the parameters. We also have a semicolon at the end of the closing curly brace, since this is all technically one operation.

We can then pass these functions around in place of the array or string invocations we were using. Doing so allows us to take advantage of scoping, as these functions will exist only the appropriate scope, but at the same time can be passed and returned like any other value. Since `call_user_func()` accepts any callable, these anonymous functions slot right in and work as we expect! We'll see when this is useful shortly.

Listing 4

```
1. <?php
2.
3. $square = function (int $num) {
4.    return $num * $num;
5. };
6.
7. $cube = function (int $num) {
8.    return $num * $num * $num;
9. };
10.
11. function addAndModify(int $num1, int $num2,
12.                       callable $modifier) {
13.   $total = $num1 + $num2;
14.   return call_user_func($modifier, $total);
15. }
16.
17. echo addAndModify(1, 2, $square); // 9
18. echo addAndModify(1, 2, $cube); // 27
```

I did make one other small change, and that was the type hint I used for the `$modifier` parameter. PHP 5.4 introduced the `callable` type hint, which encompasses both the array syntax that we used previously with `call_user_func()` but also a class that implements `__invoke()`. We get the safety of a type hint but the ability to use the old syntax still. I will be using this type hint going forward.

Assigning a function to a variable can be useful when we need to re-use the function in multiple places. What happens when we need to help deal with scope issues as opposed to portability? We can define an anonymous function directly where a callable is expected as in Listing 5.

This time, instead of directly assigning the function to a variable, we pass it to `addAndModify()`. PHP stuffs it into `$modifier`, and it works fine with `call_user_func()` because this is a callable. This syntax is excellent when you need to help scope a portion of code, but do not need to re-use it afterward. Many times this is used with the various array functions, such as passing a function to `usort()`[3] for custom sorting.

I also made another syntax change. See if you can figure out what it was… go ahead, I have time. I mean, I already wrote all this, so I can wait as long as needed.

I changed the syntax for how we call our callable. I removed `call_user_func()` and just invoked the function like any other function, except it has a dollar sign at the beginning to refer to the variable holding the anonymous function. As it turns out, we can invoke any callable this way.

PHP tries to see if it is invokable if you put parentheses at the end of a variable.

```
$result = $canDoSomething();
```

The invokable array syntax is called correctly, the same as a function assigned to a variable. This syntax makes our code even more readable. I am using that syntax going forward.

Lambdas, Closures, And Anonymous Functions

There are a few definitions and constructs that exist around the idea of "anonymous functions." It is close to the argument that all squares are rectangles, but not all rectangles are squares.

From a computer science perspective, a lambda is just an "anonymous function." Some languages like Python make a slight distinction, but these terms are effectively interchangeable under PHP. They are both functions that have no name.

In PHP, an anonymous function has the same scoping mechanism as any other function. It knows only of the parameters it is designed to accept. Any variables it creates—and any global variables you pull in via `global`, but we're not monsters. If you want access to some bit of data, you must pass it in.

Closures are anonymous functions that have some limited knowledge of the world around them. The `use` keyword allows a closure to "use" external things without needing to pass them in directly by the calling code—which may not know about that variable at all.

In Listing 6, we allow the `$increment` anonymous function to access a new variable named `$incrementBy`. It turns it semantically into a closure. `$incrementBy` is defined outside of the function itself, but we make its value available inside of the closure. Another small gotcha is that the value of the `use` parameters is that their values are set at definition time, not invocation time.

```
$name = 'Bob';
$welcomer = function($message) use ($name) {
    echo $message . " " . $name;
};
$name = "Alice";

$welcomer("Hello");
// Hello Bob
```

This code binds `$name` to the closure, but all the standard parameter passing rules apply. By default, PHP will pass scalar objects by value and objects by reference. You can force passing by reference using the `&` modifier, so the syntax becomes `use (&$incrementBy)`.

Listing 5

```
<?php

function addAndModify(int $num1, int $num2,
                      callable $modifier) {
    $total = $num1 + $num2;
    return $modifier($total);
}

echo addAndModify(1, 2, function (int $num) {
    return $num * $num;
}); // 9
echo addAndModify(1, 2, function (int $num) {
    return $num * $num * $num;
}); // 27
```

Listing 6

```
<?php

$incrementBy = 2;
$increment = function ($num) use ($incrementBy) {
    return $num + $incrementBy;
};

function addAndModify(int $num1, int $num2,
                      callable $modifier) {
    $total = $num1 + $num2;
    return $modifier($total);
}

echo addAndModify(1, 2, $increment); // 5
```

3 `usort()`: https://php.net/function.usort

Listing 7

```php
<?php

class Increment
{
    public $incrementBy = 1;

    public function getClosure() {
        return function ($num) {
            return $this->incrementBy + $num;
        };
    }
}

function addAndModify(int $num1, int $num2,
                      callable $modifier) {
    $total = $num1 + $num2;
    return $modifier($total);
}

$one = new Increment();
$two = new Increment();
$two->incrementBy = 2;

$oneClosure = $one->getClosure();
$oneClosure = $oneClosure->bindTo($two);
echo addAndModify(1, 2, $oneClosure); // 5
```

`$this` works differently when creating closures. If we create a closure inside of an object, it gets scoped to that object. We can shift this around if we need to, in any case (Listing 7).

In this case, we create a closure that would normally increment by one, but by re-binding it to the second instance with the increment of 2, we changed from where it can pull data.

Invokable Objects

Between `call_user_func()` and anonymous functions, we have a few ways of handling functions and passing them around. PHP 5.3 went even further and introduced the `__invoke()`[4] magic method. It makes classes invokable all by themselves, which gives us the power of object-oriented programming alongside callables.

If a class implements the `__invoke()` magic method, then an instantiated object can be called and used as a function. You can pass arguments into the invocation, and it can return data. For all intents and purposes, it works like a function. `__invoke()` can be defined with as many parameters as you need.

An everyday use case for this type of functionality is in the concept of "Action Domain Responder," or ADR. It's a web application structure where a single class represents each Action (or URL). This class handles all the work for that particular URL. This approach is different from "Model View Controller", or MVC, where your controller classes generally encapsulate various actions as methods.

If we have a website with a homepage ("/") and an admin page ("/admin"), it equates to two Action classes. Let's represent them as `HomepageAction` and `AdminAction`. We can store those class names in an array that maps them to a requested route. When a route is requested, we can look up the class name, instantiate an object with its dependencies, and call the object as a function as in Listing 8.

Listing 8

```php
<?php

class HomepageAction
{
    public function __invoke() {
        echo "Hello World";
    }
}

class AdminAction
{
    public function __invoke() {
        echo "Secret Admin Area";
    }
}

$routes = [
    '/' => HomepageAction::class,
    '/admin' => AdminAction::class,
];

If (!empty($routes[$_SERVER['REQUEST_URI']])) {
    $className = $routes[$_SERVER['REQUEST_URI']];
    $action = new $className();
    $action();
}
```

When we call `$action()`, the PHP engine checks to see if `$action` has implemented `__invoke()`. If the class has, it calls that method. If we request `/admin`, effectively we end up calling the `__invoke()` method on the `AdminAction` class.

By implementing `__invoke()`, in Listing 9 we now get something more manageable to pass around in our code. We can even take this invokable class and pass it into `call_user_func()`!

Doing so makes our code clearer. We can create classes that do one specific job, which perfectly fits the Single Responsibility idea (each class should affect one change in the system). While encapsulating both `increment()` and `decrement()` into

Listing 9

```php
<?php

class Square
{
    public function __invoke(int $num) {
        return $num * $num;
    }
}

function addAndModify(int $num1, int $num2,
                      callable $modifier) {
    $total = $num1 + $num2;
    return $modifier($total);
}

echo addAndModify(1, 2, new Square()); // 9
```

the old `Modifiers` class makes some sense as they both modify a number, having bespoke classes is even clearer.

Promises

If you have worked with older Node.js code, or any async PHP code, you may have come across this idea of "Callback Hell." This situation arises where you have callbacks that have callbacks that have callbacks. You end up with this nested mess of code that is very hard to move around and even interpret sometimes.

I love the array functions, but many of them use callbacks, and it is very easy to fall into callback hell with them, especially if you need to chain multiple operations together. Callbacks are perfectly fine until it impacts readability and maintainability.

Read the code in Listing 10. Try to figure out what I'm doing, and in what order everything happens.

This code finds all the active users with a balance of "10" or higher and totals it all up. This operation is essentially a very filter-and-reduce block of code, but this is very close to what can happen when it comes to overusing the idea of callbacks, especially anonymous ones.

In asynchronous code, this flow happens frequently. Async code works on generating new jobs based on existing workloads, and the callback paradigm fits well into this. Throw in PHP's flexible callable infrastructure and you have a mess waiting to happen.

Most of the time, this happens when developers from more function-first languages, like JavaScript, try and replicate their workflows in PHP. PHP has some additional language constructs, like our callables and object structure, that make this chaos more manageable.

Since PHP has a more defined object structure, I would recommend taking your callbacks and moving them into invokable classes. This change removes a lot of the boilerplate from the business logic and moves things into their code blocks. The improved readability would be more than welcome with this block of code.

In the JavaScript world, this problem bore out a solution: Promises/A+. These are an attempt to provide some structure to dealing with feeding data into one end of a pipeline and getting data out at the other, without knowing what those pipelines would look like or the order of execution.

Promises/A+, or more commonly just known as Promises, add an object-oriented wrapper for defining the order to call our callables and efficiently handling errors. A Promise is essentially two callables. One handles incoming data, and the other handles if there are any exceptions thrown. You can then chain these promises together into a workflow pipeline.

Guzzle, the HTTP client has a Promises implementation called Guzzle Promises[5]. If you are interested in seeing how Promises could work within the confines of PHP, I would recommend taking a look at that library.

Old Tricks In a Modern Era

PHP has been able to do most of this for a very long time, but because developers tend to focus on object-oriented programming, it is a part of PHP that gets overlooked. Callbacks and Callables are something you should understand and all to your toolbox.

The next time you write code, try and see how others may want to extend it. Instead of only focusing on how you can extend an individual class, look at how you can let other developers add additional steps to your workflows through callbacks. Are you writing something that a user may want to do further processing on, like data filtering or text manipulation?

Do not be afraid to sprinkle in some callbacks. Just watch out for Callback Hell.

Listing 10

```php
<?php

$a = [
    ['name' => 'Bob', 'active' => 1, 'balance' => 10],
    ['name' => 'Alice', 'active' => 1, 'balance' => 10],
    ['name' => 'Jane', 'active' => 0, 'balance' => 0],
    ['name' => 'John', 'active' => 1, 'balance' => 5],
    ['name' => 'Bill', 'active' => 0, 'balance' => 10],
    ['name' => 'Jan', 'active' => 1, 'balance' => 10],
];

$d = array_reduce(array_filter(array_filter($a, function ($b) {
    return (bool)$b['active'];
}), function ($b) {
    return $b['balance'] >= 10;
}), function ($c, $b) {
    return $c + $b['balance'];
});
```

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. He spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows.
@dragonmantank

Related Reading

- *Juggle Arrays Using Functional Callbacks*
  by Andrew Koebbe, October 2016.
  https://www.phparch.com/magazine/2016-2/october/
- *Removing the Magic with Functional PHP*
  by David Corona, July 2016.
  https://www.phparch.com/magazine/2016-2/july/

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]