Banishing Loops with Functional PHP

@davidbhayes

Goals

To understand...

1. what functional programming is
2. practical substitutions we can make in our code using it
3. why it can be beneficial to make these changes

In the beginning, there was code. And that was all...

As time goes on, we learn that maintaining code is hard

To ponder...

Why is maintaining code hard?

Today, there are basically

Three Paradigms

Procedural Code

- You tell the computer exactly what to do
- You might have "functions" which contain your common procedures

OO Code

- You use the inherent human understanding of objects to make code more comprehensible
- In PHP, you use classes to define blueprint for instances of objects

Functional Code

- Mathematical-functions have magic powers, pure functions are more unitary than procedures
- All pure functions are stateless and create only their own output

Functional Programming

in a bit more depth

Pure functions...

Read only their inputs

Pure functions...

Affect only their outputs

Only touching inputs and outputs means

- No DB changes
- No files created
- No reading of global state (like time)

It also means...

- Functions are completely *idempotent*
- Functions can be *composed*
- Rerunning a function for debugging requires *only* knowing its name and parameters

Functional programming languages

- Haskell
- Elm
- Clojure
- Scala (sort of)
- JavaScript (sort of)

Now, Lets Talk About

PHP

Most "classic" or "legacy" PHP is

procedural

Most modern PHP code is

object-oriented

with procedural processes inside

I love `foreach`

`foreach` is fundamentally procedural

(yes, even inside an object method)

A classic PHP pattern:

```
$saved = [];
foreach($items as $item) {
    if ($item->size > 5) {
        $saved[] = $item;
    }
}
$display_titles = [];
foreach($saved as $item) {
    $display_titles[] = ucfirst($item->title);
}
foreach($display_titles as $title) {
    echo '<h1>'.$title.'</h1>';
}
```

How Does Functional Help?

What's `map`?

Transform each entity in a list using a given function

What's `filter`?

Keep items in a list if a function return true when run on an item

Getting Practical

FP in PHP

@davidbhayes from ThoughtfulCode.com

array_map

```
// array_map ( callable $callback , array $array1 [, array $... ] )

$start = [1, 2, 3, 4, 5];
$end = array_map(function ($i) {
    return $i * 2;
}, $start);

// [2, 4, 6, 8, 10]
```

`array_map` makes this code...

```
$start = [1, 2, 3, 4, 5];
$end = [];
foreach($start as $i) {
    $end[] = $i * 2;
}
```

Into this

```
$start = [1, 2, 3, 4, 5];
$end = array_map(function ($i) {
    return $i * 2;
}, $start);
```

array_filter

```
// array_filter ( array $array [, callable $callback [, int $flag = 0 ]] )

$start = [1, 2, 3, 4, 5, 6];
$even = array_filter($start, function ($i) {
    return $i % 2 === 0;
});

// [2, 4, 6]
```

`array_filter` makes this code...

```
$start = [1, 2, 3, 4, 5, 6];
$even = [];
foreach($start as $i) {
    if ($i % 2 === 0) {
        $even[] = $i;
    }
}
```

Into this

```
$start = [1, 2, 3, 4, 5, 6];
$even = array_filter($start, function ($i) {
    return $i % 2 === 0;
});
```

Let's Talk about Callables

Simple function declarations are global

```
function hello($name) {
    return 'Hello, '.$name;
}
```

Called like...

```
echo hello('David');
```

Anonymous functions can be saved to variables

```
$hello = function($name) {
    return 'Hello, '.$name;
}
```

Called like...

```
echo $hello('David');
```

Classes can have static methods

```
class Namer {
    public static function hello($name) {
        return 'Hello, '.$name;
    }
}
```

Called like...

```
echo Namer::hello('David');
```

Objects can have methods

```
class Namer {
    public function hello($name) {
        return 'Hello, '.$name;
    }
}
```

Called like...

```
$namer = new Namer;
echo $namer->hello('David');
```

All forms can be used with `array_map`, etc

```
$names = ['David', 'Megan', 'Sierra'];

array_map('hello', $names);

array_map($hello, $names);

array_map('Namer::hello', $names);
array_map(['Namer', 'hello'], $names);

$namer = new Namer;
array_map([$namer, 'hello'], $names);
```

Let's Talk about use

Variable Scoping

This does not work!

```
$nonlocal = 7;
$greaterThan = function($number) {
    return $number > $nonlocal;
}
$greaterThan(1);
```

This also does not work!

```
global $nonlocal = 7;
$greaterThan = function($number) {
    return $number > $nonlocal;
}
$greaterThan(1);
```

This does work!

```
global $nonlocal = 7;
$greaterThan = function($number) {
    global $nonlocal;
    return $number > $nonlocal;
}
$greaterThan(1);
```

But this is better!

```
$nonlocal = 7;
$greaterThan = function($number) use ($nonlocal) {
    return $number > $nonlocal;
}
$greaterThan(1);
```

Why this matters

```
$nonlocal = 7;
$greaterThan = function($number) use ($nonlocal) {
    return $number > $nonlocal;
}
array_filter($array, $greaterThan);
```

Or else

```
$nonlocal = 7;
array_filter($array, function($number) use ($nonlocal) {
    return $number > $nonlocal;
});
```

Remember this?

```
$saved = [];
foreach($items as $item) {
    if ($item->size > 5) {
        $saved[] = $item;
    }
}
$display_titles = [];
foreach($saved as $item) {
    $display_titles[] = ucfirst($item->title);
}
foreach($display_titles as $title) {
    echo '<h1>'.$title.'</h1>';
}
```

Let's transform

filter in for foreach

```
$saved = array_filter($items, function($i) {
    return $item->size > 5;
});
$display_titles = [];
foreach($saved as $item) {
    $display_titles[] = ucfirst($item->title);
}
foreach($display_titles as $title) {
    echo '<h1>'.$title.'</h1>';
}
```

Let's transform

map for capitalizing

```
$saved = array_filter($items, function($i) {
    return $i->size > 5;
});
$display_titles = array_map(function($item) {
    return ucfirst($item->title);
}, $saved);
// somewhere else
foreach($display_titles as $title) {
    echo '<h1>'.$title.'</h1>';
}
```

Hassles with that

- Temporary variables at every step
- Argument order on `array_filter` and `array_map` is inconsistent

But maybe we can solve that...

Collection Libraries

Collections allow us to streamline filter, map, etc

Why collection pipelines?

Easier to read -- we read left to right in English

`array_map(array_filter())` executes inside out

Skip argument order issues

Which one's the array?

No need for temps with fluent interface

```
->chaining()->methods()->is()->cool();
```

Your ORM may already have one...

Using a Laravel Collection

We have a Reddit-like site:

- Posts have scores
- Comments have scores
- We have a 'fluency scorer'
- We have a hypothesis that good posts get higher scoring and more fluent comments

Without FP

```
$posts = Posts::all();
$goodPosts = [];
foreach($posts as $post) {
    if ($post->score > 500) {
        $goodPosts[] = $post;
    }
}
$goodComments = [];
foreach($goodPosts as $post) {
    $comments = $post->comments;
    foreach($comments as $c) {
        if ($c->score > 100) {
            $goodComments[] = $c;
        }
    }
}
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Refactor 1 - filter out posts

```
$goodPosts = Posts::all()
                ->filter(function($post) {
                    return $post->score > 500
                });
$goodComments = [];
foreach($goodPosts as $post) {
    $comments = $post->comments;
    foreach($comments as $c) {
        if ($c->score > 100) {
            $goodComments[] = $c;
        }
    }
}
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Quick Explanation

- `flatten` will take nested arrays and de-nest them
- PHP doesn't have an native `array_flatten`, but you can make one

Refactor 2 - collect good comments on good posts

```
$goodComments = Posts::all()
                ->filter(function($post) {
                    return $post->score > 500
                })
                ->map(function($p) {
                    return $post->comments;
                })
                ->flatten()
                ->filter(function($c) {
                    return $c->score > 100;
                });
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Quick Simplification

- `flatMap` is a shortcut for `map` then `flatten`

Refactor 2 - collect good comments on good posts (again)

```
$goodComments = Posts::all()
                ->filter(function($post) {
                    return $post->score > 500
                })
                ->flatMap(function($p) {
                    return $post->comments;
                })
                ->filter(function($c) {
                    return $c->score > 100;
                });
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Refactor 3 - create new set with a map

```
$scoredGoodCommments =
    Posts::all()
        ->filter(function($post) {
            return $post->score > 500
        })
        ->flatMap(function($p) {
            return $post->comments;
        })
        ->filter(function($c) {
            return $c->score > 100;
        })
        ->map(function($c) {
            return [
                'score' => $c->score,
                'fluency' => FluentScorer::score($c->content),
                'comment' => $c,
                'post' => $c->post,
            ];
        });
```

Did it get better? (original)

```
$posts = Posts::all();
$goodPosts = [];
foreach($posts as $post) {
    if ($post->score > 500) {
        $goodPosts[] = $post;
    }
}
$goodComments = [];
foreach($goodPosts as $post) {
    $comments = $post->comments;
    foreach($comments as $c) {
        if ($c->score > 100) {
            $goodComments[] = $c;
        }
    }
}
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Did it get better? (final)

```
$scoredGoodCommments =
    Posts::all()
        ->filter(function($post) {
            return $post->score > 500
        })
        ->flatMap(function($p) {
            return $post->comments;
        })
        ->filter(function($c) {
            return $c->score > 100;
        })
        ->map(function($c) {
            return [
                'score' => $c->score,
                'fluency' => FluentScorer::score($c->content),
                'comment' => $c,
                'post' => $c->post,
            ];
        });
```

What makes code hard to maintain?

- Short-lived variables are clutter
- Disguised intent via `foreach`
- Deeply nested conditionals

Does FP help?

Local and temporary `$variables` are reduced, especially with pipelines

Does FP help?

Replacing `foreaches` with `map` and `filter` makes it clearer what each does

Does FP help?

Nesting is minimized by small, single purpose functions

"A program cannot change until it is alive in a programmer's head."

—Jessica Kerr on Ruby Rogues

(via @johnkary)

Localizing control and complexity makes it easier for you to

jump in

between meetings, children, life, etc

Again

Did it get better? (original)

```
$posts = Posts::all();
$goodPosts = [];
foreach($posts as $post) {
    if ($post->score > 500) {
        $goodPosts[] = $post;
    }
}
$goodComments = [];
foreach($goodPosts as $post) {
    $comments = $post->comments;
    foreach($comments as $c) {
        if ($c->score > 100) {
            $goodComments[] = $c;
        }
    }
}
$scoredGoodComments = [];
foreach($goodComments as $c) {
    $local['score'] = $c->score;
    $local['fluency'] = FluentScorer::score($c->content);
    $local['comment'] = $c;
    $local['post'] = $c->post;
    $scoredGoodComments[] = $local;
}
```

Did it get better? (final)

```
$scoredGoodCommments =
    Posts::all()
        ->filter(function($post) {
            return $post->score > 500
        })
        ->flatMap(function($p) {
            return $post->comments;
        })
        ->filter(function($c) {
            return $c->score > 100;
        })
        ->map(function($c) {
            return [
                'score' => $c->score,
                'fluency' => FluentScorer::score($c->content),
                'comment' => $c,
                'post' => $c->post,
            ];
        });
```

Thank You!

I've been @davidbhayes

I run a site called WPShout

and another called Thoughtful Code