How To Make Your PHP Code Beautiful With Chainable Methods

Functional programming, meet PHP

Chris Geelhoed Follow
Nov 11 · 5 min read ★

Photo by William Zhang on Unsplash

PHP has a bad reputation for being ugly. The language evolved piecemeal over time, and it shows.

But it has a lot of redeeming qualities as well. The PHP development team has consistently improved the language with every release. PHP 7, in particular, made great strides in both features and performance.

Other benefits of the language include friendly documentation, ubiquitous hosting, and a considerable amount of online resources to draw from (on Stack Overflow and otherwise).

And yes, all the biggest CMS platforms (Wordpress, Drupal, Joomla) are PHP-based and there is a lot of work to be done in that arena.

So, for those of us that use PHP and want to write beautiful code, how can this be achieved?

Naysayers might claim that this isn't possible, but I disagree. Frameworks like Laravel ("the PHP framework for web artisans") have already proven them wrong.

One pattern that Laravel and other respected PHP frameworks often implement is *method chaining*.

Here's an example of what I'm talking about, Laravel's query builder API:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

Reference: https://laravel.com/docs/5.6/queries.

This "chaining" syntax is nice because it allows developers to write code that does one thing at a time in an expressive and readable way. In the example above, it is easy to see what is happening:

1. Look at rows in the `users` table.
2. Where votes are greater than 100.
3. Or where the user's name is John.

4. Get the results.

Pretty slick, right? Let's dive into how this pattern can be brought to life.

• • •

A Concrete Example

To illustrate how method-chaining works, let's look at array operations as an example. PHP has a lot of helpful array functions like `array_map`, `array_filter`, `usort`, etc. but performing multiple operations on one array can be an eyesore.

Let's say that we have an array of Simpson character names, and we want to apply the following transformations:

1. Filter out any names that don't end in "Simpson".
2. Map each full name to just a first name. For example, we'd replace "Lisa Simpson" with simply "Lisa".
3. Sort our results alphabetically.

Here is one way our problem could be handled:

```
$characters = [
    'Maggie Simpson',
    'Edna Krabappel',
    'Marge Simpson',
    'Lisa Simpson',
    'Moe Szyslak',
    'Waylon Smithers',
    'Homer Simpson',
    'Bart Simpson'
];

// Look for full names that end in Simpson
// Filter out items that don't match
$simpsons = array_filter($characters, function ($character) {
    return preg_match('/^.+\sSimpson$/', $character);
});
```

```
// Replace " Simpson" with an empty string for each item
$simpsons = array_map(function ($character) {
    return str_replace(' Simpson', '', $character);
}, $simpsons);

// Sort the items using PHP's "strcasecmp"
usort($simpsons, function ($a, $b) {
    return strcasecmp($a, $b);
});

var_dump($simpsons); // ['Bart', 'Homer', 'Lisa', 'Maggie', 'Marge']
```

This gets us the correct result but those with a keen eye might spot a few details:

1. `array_filter` accepts an array and a callback but `array_map` accepts a callback and array. Why would these functions accept arguments in a different order?
2. The filter and map operations are being saved into the `$simpsons` variable but `usort` is accessing our array by reference. Why the inconsistency?
3. This just looks kind of ugly.

As a comparison, let's see how the problem could be done in JavaScript:

```
const simpsons = characters
  .filter(character => character.match(/^.+\sSimpson$/))
  .map(character => character.replace(' Simpson', ''))
  .sort((a, b) => b < a)

console.log(simpsons)
```

The JavaScript solution is far more elegant. Arrays are a type of object in JavaScript, and this allows for array operations to be chained.

Aside: The JavaScript above uses arrow functions for brevity. PHP doesn't have arrow functions yet, but we might have them soon!

Method-chaining on arrays comes for free in JavaScript, but we can emulate this in PHP by writing our own custom class.

. . .

Creating a Class With Chainable Methods

Let's call it a `Collection`. An instance of this class can be created by passing an array into the constructor.

```
class Collection
{
    private $array;

    public function __construct($array)
    {
        $this->array = $array;
    }
}

$characters = new Collection([
    'Maggie Simpson',
    'Edna Krabappel',
    'Marge Simpson',
    'Lisa Simpson',
    'Moe Szyslak',
    'Waylon Smithers',
    'Homer Simpson',
    'Bart Simpson'
]);
```

So far, the class doesn't do a lot — it just saves the array passed as a private property. Let's set our sights on adding a public `filter` method.

```
public function filter($callback)
{
    $this->array = array_filter($this->array, $callback);
    return $this;
}
```

Instead of having to pass both an array and callback function as arguments, like before, only the callback function is required now. Once the instance's array property has been transformed, the instance itself is returned. This makes method chaining possible.

Next, let's add methods for `map` and `sort`:

```
public function map($callback)
{
    $this->array = array_map($callback, $this->array);
    return $this;
}

public function sort($callback)
{
    usort($this->array, $callback);
    return $this;
}
```

Now our methods are ready to be chained, but we need a way to get back the final result as an array. This is similar to how the Laravel query builder uses `get()` to execute a query after it has been built up with conditionals.

```
public function execute()
{
    return $this->array;
}
```

Putting it all together, our class looks like this:

```
class Collection
{
    private $array;

    public function __construct($array)
    {
        $this->array = $array;
    }

    public function filter($callback)
    {
        $this->array = array_filter($this->array, $callback);
        return $this;
    }

    public function map($callback)
    {
```

```
        $this->array = array_map($callback, $this->array);
        return $this;
    }

    public function sort($callback)
    {
        usort($this->array, $callback);
        return $this;
    }

    public function execute()
    {
        return $this->array;
    }
}
```

And we can finally chain array methods as follows!

```
$characters = new Collection([
    'Maggie Simpson',
    'Edna Krabappel',
    'Marge Simpson',
    'Lisa Simpson',
    'Moe Szyslak',
    'Waylon Smithers',
    'Homer Simpson',
    'Bart Simpson'
]);

$simpsons = $characters
    ->filter(function ($character) {
        return preg_match('/^.+\sSimpson$/', $character);
    })
    ->map(function ($character) {
        return str_replace(' Simpson', '', $character);
    })
    ->sort(function ($a, $b) {
        return strcasecmp($a, $b);
    })
    ->execute();

var_dump($simpsons); // ['Bart', 'Homer', 'Lisa', 'Maggie', 'Marge']
```

Very JavaScript-esque! The issues with argument order and reference access still exist under the hood, but our `Collection` class handles this behind the scenes. The result is code that is much cleaner and more readable!

• • •

Summary

PHP is infamous for being messy, but Laravel and other frameworks are using chainable methods (among other things) to encourage beautiful code and challenge this notion.

Array operations in PHP are not chainable like they are in JavaScript, but a basic utility class (like our `Collection` class!) can be used to emulate this and keep some of PHP's little quirks behind the curtain.

PHP Software Development Functional Programming Coding Programming