Mw Documentation

Release 0.3.1

RJ Garcia

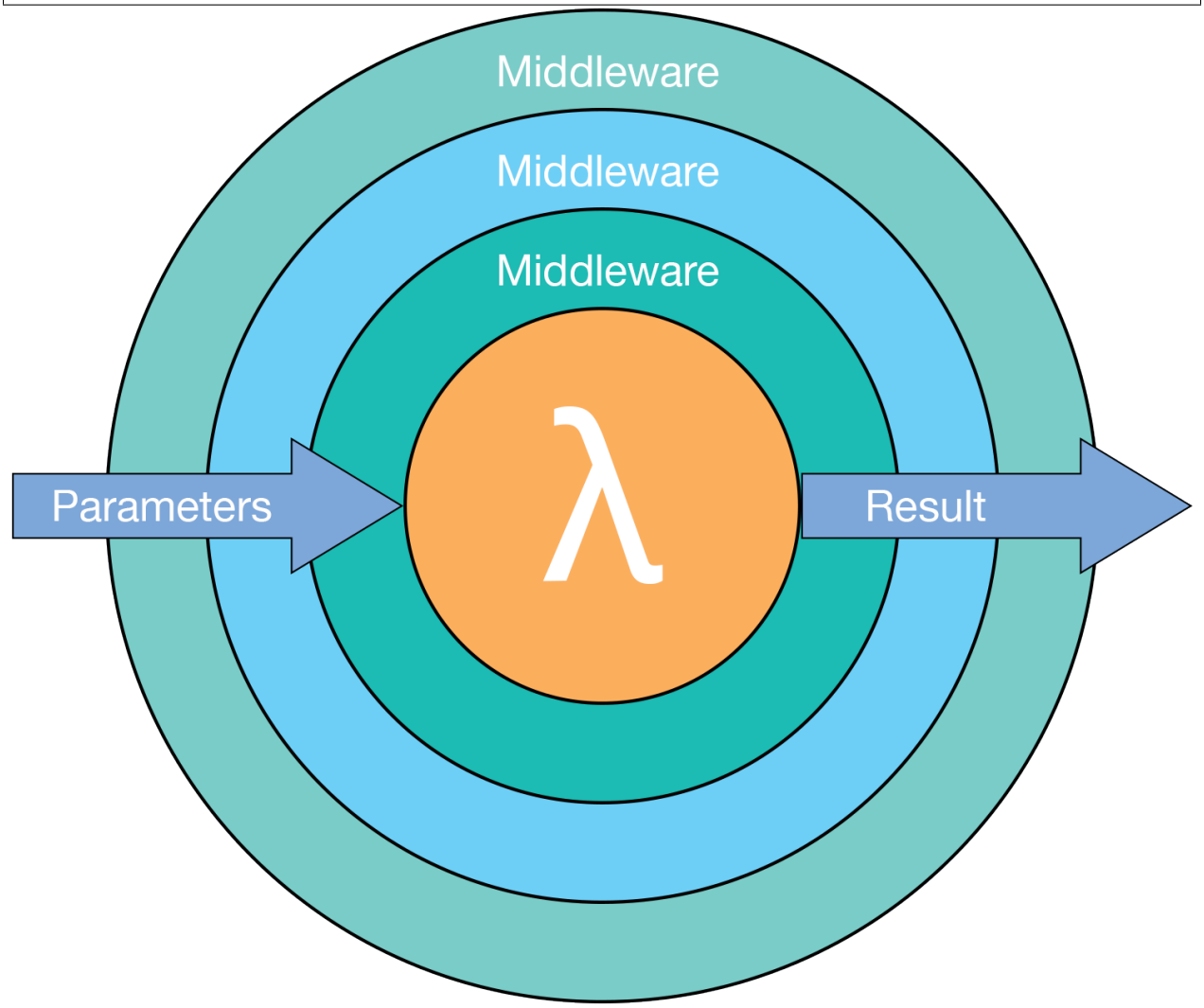
December 20, 2016

Contents

The Mw library is a very flexible framework for converting middleware into handlers. Middleware offer a clean syntax for implementing the [Decorator Pattern](https://en.wikipedia.org/wiki/Decorator_pattern)

Middleware provide a great system for extendable features and the Krak\Mw library offers a simple yet powerful implementation to create middleware at ease.

```php
<?php

use Krak\Mw;

$handler = mw\compose([
    function($s, $next) {
        return strtoupper($s);
    },
    function($s, $next) {
        return 'x' . $next($s . 'x');
    }
]);

$res = $handler('abc');
assert($res == 'xABCX');
```



CHAPTER 1

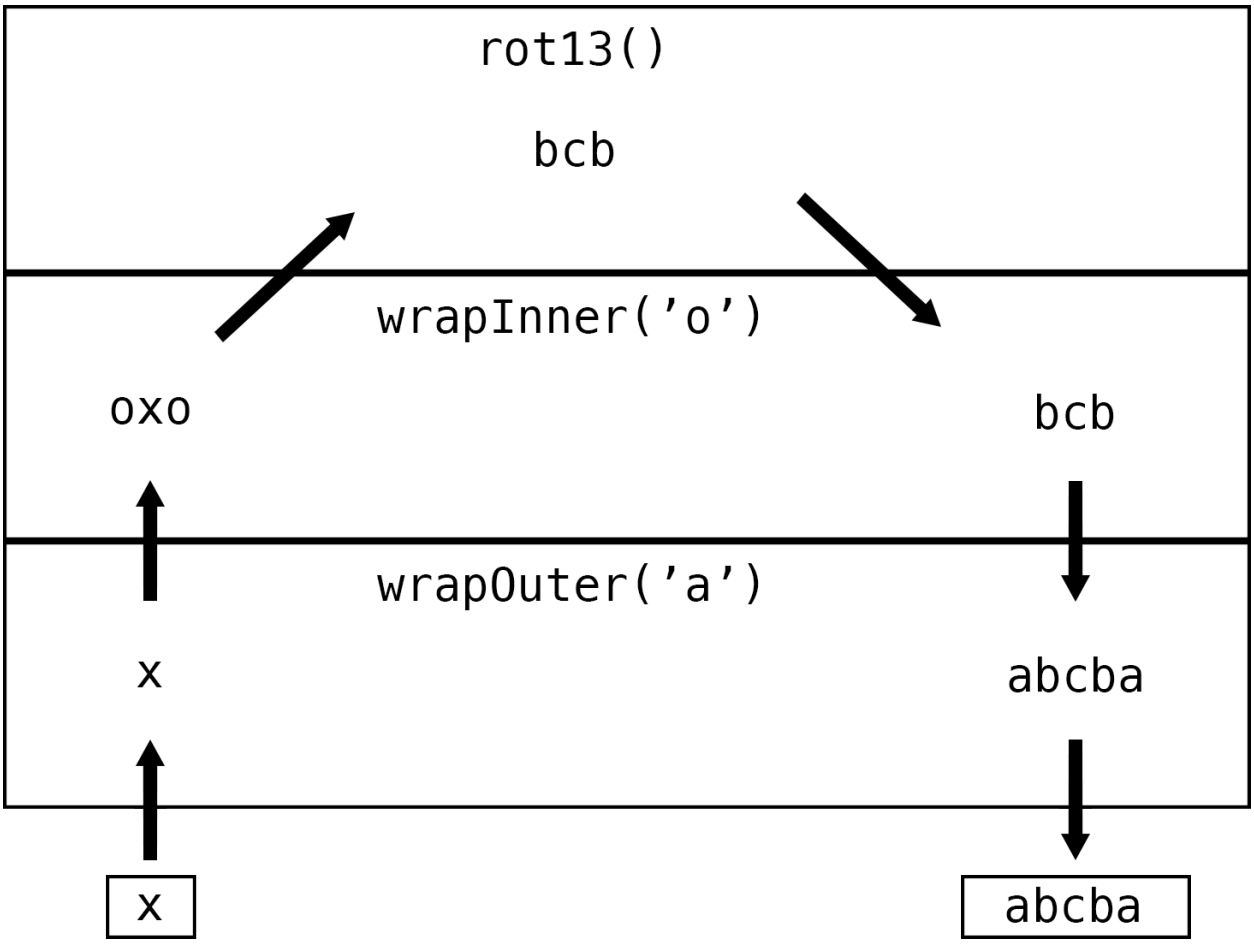
Usage

Note: each of these code samples can be seen in the `example` directory of the repo.

Here's an example of basic usage of the mw library

```php
<?php

use Krak\Mw;

function rot13() {
    return function($s) {
        return str_rot13($s);
    };
}

function wrapInner($v) {
    return function($s, $next) use ($v) {
        return $next($v . $s . $v);
    };
}
function wrapOuter($v) {
    return function($s, $next) use ($v) {
        return $v . $next($s) . $v;
    };
}

$handler = mw\compose([
    rot13(),
    wrapInner('o'),
    wrapOuter('a'),
]);

echo $handler('p') . PHP_EOL;
// abcba
```

The first value in the array is executed last; the last value is executed first.



Each middleware shares the same format:

```
function($arg1, $arg2, ..., $next);
```

A list of arguments, with a final argument $next which is the next middleware function to execute in the stack of middleware.

You can have 0 to n number of arguments. Every middleware needs to share the same signature. Composing a stack of middleware will return a handler which has the same signature as the middleware, but without the *$next* function.

IMPORTANT: At least one middleware MUST resolve a response else the handler will throw an error. So make sure that the last middleware executed (the first in the set) will return a response.

1.1 Before/After Middleware

Middleware can either be a *before* or *after* or both middleware. A before middleware runs before delegating to the `$next` middleware. An after middleware will runs *after* delegating to the `$next` middleware.

Before Style

```
<?php

function($param, $next) {
    // code goes here
    // you can also modify the $param and pass the modified version to the next middleware
```

```
    return $next($param);
}
```

After Style

```
<?php

function($param, $next) {
    $result = $next($param);

    // code goes here
    // you can also modify the $result and return the modified version to the previous handler

    return $result;
}
```

1.2 Stack

The library also comes with a MwStack that allows you to easily build a set of middleware.

```
<?php

use Krak\Mw;

$stack = mw\stack('Stack Name');
$stack->push(function($a, $next) {
    return $next($a . 'b');
})
->push(function($a, $next) {
    return $next($a) . 'c';
}, 0, 'c')
// this goes on first
->unshift(function($a, $next) {
    return $a;
}))
->before('c', function($a, $next) {
    return $next($a) . 'x';
})
->after('c', function($a, $next) {
    return $next($a) . 'y';
});

$handler = $stack->compose();
$res = $handler('a');
// $res = abxcy
```

CHAPTER 2

API

The api documentation is broken up into 2 parts: Middleware documentation and Middleware Stack documentation.

2.1 Middleware Functions

Closure compose(array $mws, callable $last = null, callable $invoke = null) Composes a set of middleware into a handler.

```php
<?php

$handler = mw\compose([
    function($a, $b, $next) {
        return $a . $b;
    },
    function($a, $b, $next) {
        return $next($a . 'b', $b . 'd');
    }
]);

$res = $handler('a', 'c');
assert($res === 'abcd');
```

The middleware stack passed in is executed in LIFO order. So the last middleware will be executed first, and the first middleware will be executed last.

After composing the stack of middleware, the resulting handler will share the same signature as the middleware except that it **won't** have the `$next`.

The final parameter is a callable with the same interface as `call_user_func` (`function($function_name, ...$params)`) and that is the defaulted value if no `$invoke` is set. The `$invoke` function is used to actually invoke or call the middleware. This allows for neat things like container aware middleware where a middleware is just a reference to an entry in a service container. See *Advanced Usage* for more detail.

Formally, the middleware signature is exactly the same as the resulting composed handler except that they have 2 additional parameters: `$next` and `$invoke`.

Handler Signature:

```
function(...$params): mixed
```

Middleware Signature:

```
function(...$params, $next, $invoke): mixed
```

Closure group(array $mws) Creates a new *middleware* composed as one from a middleware stack.

Internally, this calls the `compose` function, so the same behaviors will apply to this function.

```php
<?php

/** some middleware that will append values to the parameter */
function appendMw($c) {
    return function($s, $next) use ($c) {
        return $next($s . $c);
    };
}

$handler = mw\compose([
    function($s) { return $s; },
    append('d'),
    mw\group([
        append('c'),
        append('b'),
    ]),
    append('a'),
]);

$res = $handler('');
assert($res === 'abcd');
```

On the surface, this doesn't seemv very useful, but the ability group middleware into one allows you to then apply other middleware onto a group.

For example, you can do something like:

```
$grouped = mw\group([
    // ...
]);
mw\filter($grouped, $predicate);
```

In this example, we just filted an entire group of middleware

Closure lazy(callable $mw_gen) Lazily creates and executes middleware when it's executed. Useful if the middleware needs to be generated from a container or if it has expensive dependencies that you only want initialized if the middleware is going to be executed.

```php
<?php

$mw = lazy(function() {
    return expensiveMw($expensive_service_that_was_just_created);
});
```

The expensive service won't be created until the *$mw* is actually executed

Closure filter(callable $mw, callable $predicate) Either applies the middleware or skips it depending on the result of the predicate. This if very useful for building conditional middleware.

```php
<?php

$mw = function() { return 2; };
$handler = mw\compose([
    function() { return 1; },
```

```
        mw\filter($mw, function($v) {
            return $v == 4;
        })
    ]);
    assert($handler(5) == 1 && $handler(4) == 2);
```

In this example, the stack of middleware always returns 1, however, the filtered middleware gets executed if the value is 4, and in that case, it returns 2 instead.

2.2 Invoke Functions

Closure pimpleAwareInvoke(Pimple\Container $c, $invoke = 'call_user_func') invokes middleware while checking if the mw is a service defined in the pimple container

2.3 Stack Functions

MwStack stack($name, array $entries = [], $invoke = null) Creates a MwStack instance. Every stack must have a name which is just a personal identifier for the stack. It's primary use is for errors/exceptions that help the user track down which stack has an issue.

```
    <?php

    $stack = mw\stack('demo stack');
    $stack->push($mw)
        ->unshift($mw1);

    // compose into handler
    $handler = $stack->compose();
    // or, use as a grouped middleware
    $handler = mw\compose([
        $mw2,
        $stack
    ]);
```

array stackEntry(callable $mw, $sort = 0, $name = null) Creates an entry for the MwStack. This is only used if you want to initialize a stack with entries, else, you'll just be using the stack methods to create stack entries.

```
    <?php

    $stack = mw\stack('demo stack', [
        stackEntry($mw1, 0, 'mw1'),
        stackEntry($mw2),
        stackEntry($mw3, 5, 'mw3'),
    ]);
    // equivalent to
    $stack = mw\stack('demo stack')
        ->push($mw1, 0, 'mw1')
        ->push($mw2)
        ->push($mw3, 5, 'mw3');
```

MwStack stackMerge(...$stacks) Merges stacks into one another. The resulting stack has the same name as the first stack in the set. The values from the later stacks will override the values from the earlier stacks.

```php
<?php

$a = mw\stack('stack', [
    mw\stackEntry($mw1),
    mw\stackEntry($mw2),
    mw\stackEntry($mw3, 0, 'mw')
]);
$b = mw\stack('stack', [
    mw\stackEntry($mw4, 0, 'mw'),
]);
$c = mw\stackMerge($a, $b);
// stack $c is equivalent to
$c = mw\stack('stack')
    ->push($mw1)
    ->push($mw2)
    ->push($mw4, 0, 'mw')
```

2.4 class MwStack implements Countable

The stack presents a mutable interface into a stack of middleware. Middleware can be added with a name and priority. Only one middleware with a given name may exist. Middleware that are last in the stack will be executed first once the stack is composed.

__construct($name, $invoke = null) Creates the mw stack with a name and an optional invoker. The invoker will be passed along to `mw\compose`.

string getName() returns the name of the middleware

callable getInvoke() returns the current value set for invoke which could be null or a callable.

MwStack push($mw, $sort = 0, $name = null) Pushes a new middleware on the stack. The sort determines the priority of the middleware. Middleware pushed at the same priority will be pushed on like a stack.

MwStack unshift($mw, $sort = 0, $name = null) Similar to push except it prepends the stack at the beginning.

MwStack before($name, $mw, $mw_name = null) Inserts a middleware right before the given middleware.

MwStack after($name, $mw, $mw_name = null) Inserts a middleware right after the given middleware.

array shift($sort = 0) Shifts the stack at the priority given by taking an element from the front/bottom of the stack. The shifted stack entry is returned as a tuple.

array pop($sort = 0) Pops the stack at the priority given be taking an element from the back/top of the stack. The popped stack entry is returned as a tuple.

array remove($name) Removes a named middleware. The removed middleware is returned as a tuple.

array normalize() Normalizes the stack into an array of middleware that can be used with `mw\compose`

mixed __invoke(...$params) Allows the middleware stack to be used as middleware itself.

Closure compose(callable $last = null) Composes the stack into a handler.

Generator getEntries() Yields the raw stack entries in the order they were added.

MwStack static createFromEntries($name, $entries) Creates a stack with a set of entries. `mw\stack` internally calls this.

CHAPTER 3

Avanced Usage

3.1 Custom Invocation

The final argument to `mw\compose` is a callable `$invoke`. It defaults to `call_user_func` but can be any callable with the same function signature. This allows you to customize how the middleware is invoked. One specific example of this would be container aware invocation. A middleware instead of being a callable can be reference to a service in the container.

Here's an example of how to use the `pimpleAwareInvoke`

```
<?php

$c = new Pimple\Container();
$c['service'] = function() {
    return function() {
        return 1;
    };
};

$handler = mw\compose([
    'service',
    function($next) {
        return $next() + 1;
    }
], null, mw\pimpleAwareInvoke($c));

assert(2 == $handler());
```

3.1.1 Meta Middleware

Custom invocation is creation is a very useful feature; however, it requires special consideration if you are creating your own Meta Middleware. Meta middleware are middleware that accept other middleware and perform some action with the middleware.

```
mw\group
mw\lazy
mw\filter
```

These are all meta middleware. To allow custom invocation work work for *all* middleware whever they are added, these meta middleware need to make use of the `$invoke` parameter that is passed to all middleware (see *mw\compose*).

Here's an example:

```php
<?php

// maybe middleware will only invoke the middleware if the parameter is < 10
function maybe($mw) {
    return function($i, $next, $invoke) use ($mw) {
        if ($i >= 10) {
            return $next($i); // forward to next middleware
        }

        return $invoke($mw, $i, $next, $invoke);
    };
}

function loggingInvoke() {
    return function($func, ...$params) {
        echo "Invoking Middleware with Param: $params[0]\n";
        return call_user_func($func, ...$params);
    };
}

$handler = mw\compose([
    function() { return 1; },
    maybe(function($i, $next) {
        return $next($i) + 100;
    })
], null, loggingInvoke());

echo $handler(1) . PHP_EOL;
echo $handler(10) . PHP_EOL;

/*
Outputs:

Invoking Middleware with Param: 1
Invoking Middleware with Param: 1
Invoking Middleware with Param: 1
101
Invoking Middleware with Param: 10
Invoking Middleware with Param: 10
1
*/
```

CHAPTER 4

Troubleshooting

Here are few common errors and how to resolve them

4.1 “No middleware returned a response” Error

When you get this error or something similar, this means that no middleware in the set of middleware returned a response.

You can get this error if you:

- Forget to put a return statement in your middleware so the chain breaks and no response is returned.
- Have a logic error where no middleware actually accepts the response

If you are having trouble finding which handler is causing the issue, you can use a MwStack instead. These provide better error messages because each middleware stack has a name which can help track down which middlware stack is causing the problem.