Mw Documentation

Release 0.4.0

RJ Garcia

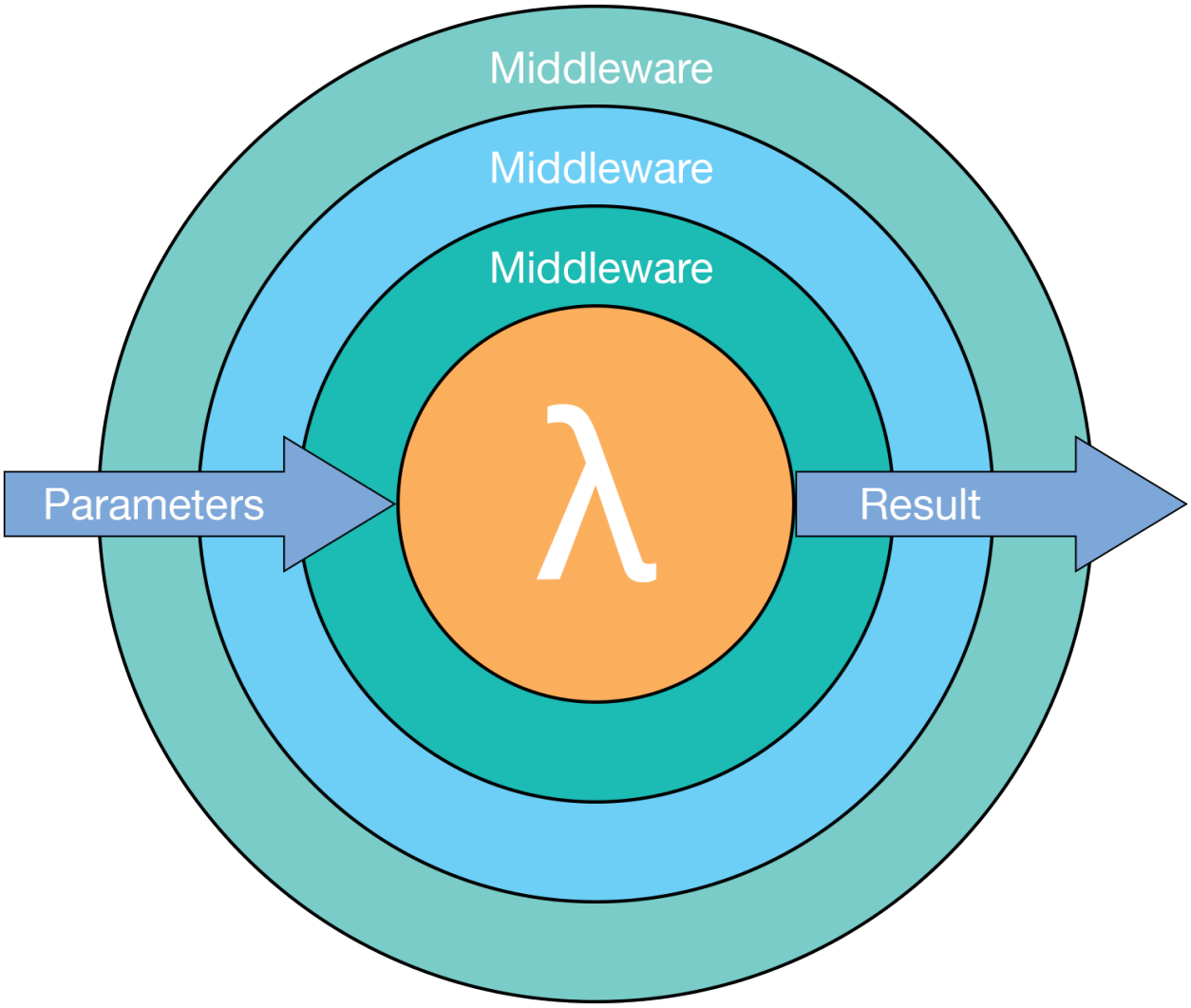
Apr 30, 2017

Contents

The Mw library is a very flexible framework for converting middleware into handlers. Middleware offer a clean syntax for implementing the 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

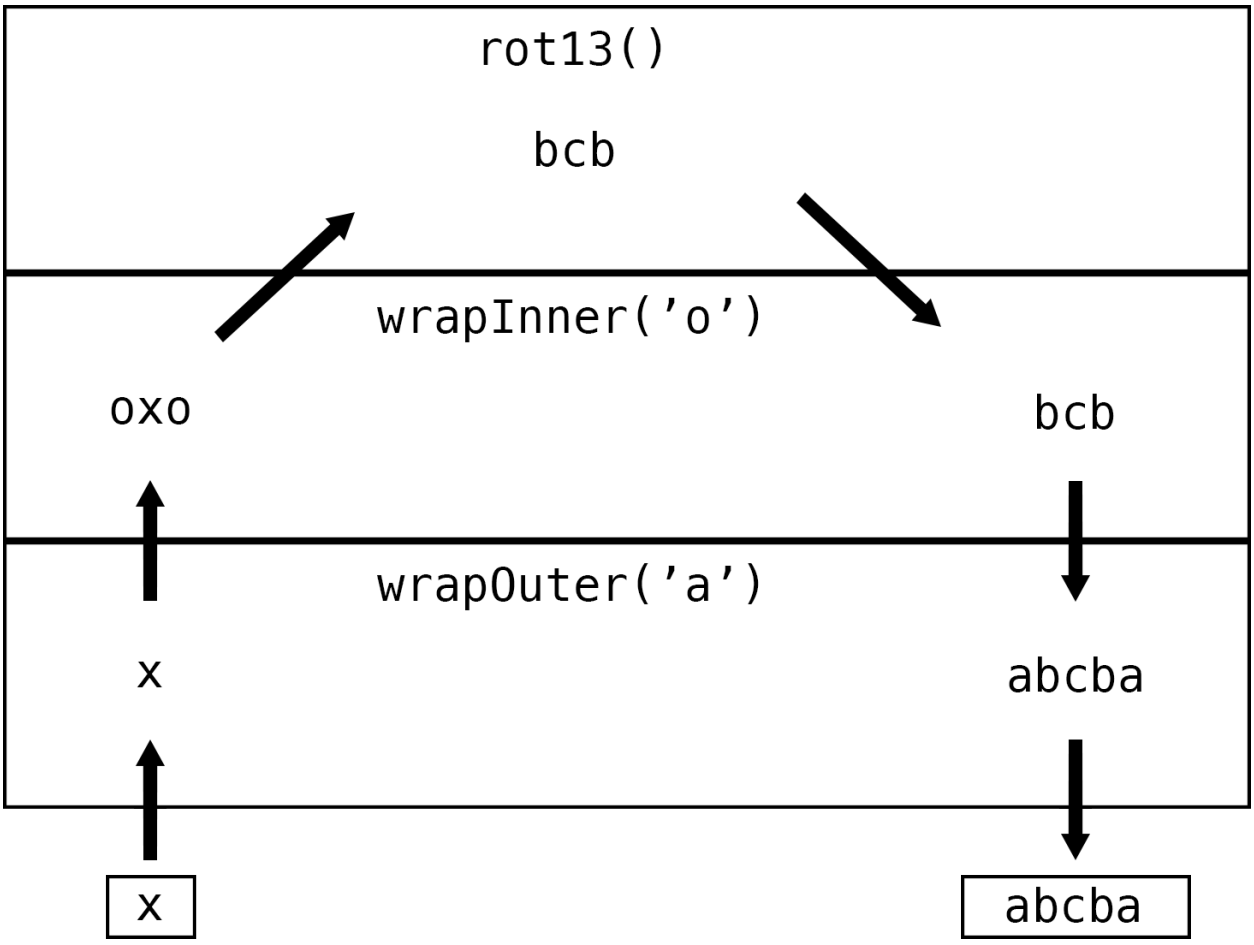
Usage

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.

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;
}
```

Stack

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

```
<?php

use Krak\Mw;

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

$handler = mw\compose([$stack]);
$res = $handler('a');
assert($res == 'abxcy');
```

Priority Stacks

You can also manage priority by determining the stack index when you push an entry. the default stack index is 0.

```php
<?php

use Krak\Mw;

$stack = mw\stack()
    ->push(mw2(), 1)
    ->push(mw1(), 1)
    ->push(mw3())
    ->push(mw4, -1);
```

In the given stack, the flow of execution is `mw1 -> mw2 -> mw3 -> mw4` because `mw1` and `mw2` were pushed at a higher stack index than the other entries.

Moving Entries

You can change the position of an entry by calling the `toTop` or `toBottom` methods of the stack. These will move the named entry to either the top or bottom of their stacks respectively.

```php
<?php

use Krak\Mw;

$stack = mw\stack()->push(function($s, $next) {
    return $next($s . 'a');
}, 0, 'append-a')
->push(function($s, $next) {
    return $next($s . 'b');
});

// the append-a entry is now at the top of the stack
$stack->toTop('append-a');

$handler = mw\compose([$stack]);
assert($handler('') == 'ab');
```

CHAPTER 2

Avanced Usage

Context

Each middleware is invoked with a `Mw\Context` instance. This is responsible for holding additional data to be used internally within the mw system *and* to provide additional features/usage for users. The context is available via the `Mw\Link` object of the middleware.

```
<?php

use Krak\Mw;

$handle = Mw\compose([
    function($v, Mw\Link $next) {
        $ctx = $next->getContext();
        return 1;
    }
], new Mw\Context\StdContext());
```

You can configure or pass in any context as long as it implements the `Mw\Context` interface. Currently, the context provides an invoker via the `getInvoke` method. This allows custom invocation of the middleware as shown in the *Cookbook*.

Custom Invocation

You can provide custom invocation of the middleware via the context. An invoker is any function that shares the signature of `call_user_func`. Its sole purpose is to invoke functions with their parameters. With custom invocation, you can do cool things like have middleware as pimple identifiers.

Link

The final argument to each middleware is an instance of `Mw\Link`. The link is represents the link/chain between middlewares. Technically speaking, it's a singly-linked list of middleware that once executed will invoke the entire chain of middleware.

The link is responsible for building a set of middleware via the `chain`.

```
<?php

use Krak\Mw;

$link = new Mw\Link(function($i) {
    return $i * 2;
}, new Mw\Context\StdContext());
$link = $link->chain(function($i, $next) {
    return $next($i) + 1;
});
assert($link(2) == 5);
```

`chain` takes a middleware and produces a new link that is appened to the head of the linked list of mw links. As you can see, the middleware on the second link is executed first.

Meta Middleware

Custom Context and invocation 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 inject middleware into the chain of middleware.

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

These are all meta middleware. To allow *all* middleware to be properly linked and have access to the context, these meta middleware need to learn how to properly inject middleware into the mw link.

Here's an example:

```
<?php

use Krak\Mw;

// maybe middleware will only invoke the middleware if the parameter is < 10
function maybe($mw) {
    return function($i, $next) use ($mw) {
        if ($i < 10) {
            /** NOTE - this is the crucial part where we prepend the `$mw` onto the
↪link. Now, when we execute `$next`,
                the `$mw` func will be first to be executed */
            $next = $next->chain($mw);
        }

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

```php
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;
    })
], new Mw\Context\StdContext(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 3

Cookbook

The cookbook provides documentation on how to extend or utilize the mw system in advanced ways.

- cookbook/custom-link-class
- cookbook/custom-method-middleware
- cookbook/container-middleware

CHAPTER 4

API

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

Middleware Functions

Closure compose(array $mws, Context $ctx = null, $link_class = Link::class) 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`.

`$ctx` will default to `Context\StdContext` if none is supplied, and it will be the context that is passed to the start link (see: *Avanced Usage* for more details).

`$link_class` is the class that will be constructed for the middleware link. It must be or extend `Krak\Mw\Link` (see: *Avanced Usage* for more details).

Closure composer(Context $ctx, $link_class = Link::class) Creates a composer function that accepts a set of middleware and composes a handler.

```php
<?php

$compose = mw\composer();
$handler = $compose([
    mw1(),
    mw2()
]);
```

Closure guardedComposer($composer, $msg)

Creates a composer that will automatically append a guard middleware with the given message when composing.

```
$compose = mw\composer();
$compose = mw\guardedComposer($compose, 'No result was returned.');
$handler = $compose([]);
$handler();
// A NoResultException will be thrown with the `No result was returned.`␣
↪message
```

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.

Invoke Functions

Closure containerAwareInvoke(Psr\Container\ContainerInterface $c, $invoke = 'call_user_func') invokes middleware while checking if the mw is a service defined in the psr container.

Closure methodInvoke(string $method_name, $allow_callable = true, $invoke = 'call_user_func') This will convert the middleware into a callable array like `[$obj, $method_name]` and invoke it. The `$allow_callable` parameter will allow the stack to either invoke objects with the given method or invoke callables. If you want to only allow objects with that method to be invokable, then set `$allow_callable` to `false`.

Stack Functions

Stack stack(array $entries = []) Creates a Stack instance. This is an alias of the `Stack::__construct`

```php
<?php

$stack = mw\stack([
    $mw1,
    $mw2
])->unshift($mw0);
```

Utility Functions

array splitArgs(array $args) Splits arguments between the parameters and middleware.

```
<?php

use Krak\Mw

function middleware() {
    return function(...$args) {
        list($args, $next) = Mw\splitArgs($args);
        return $next(...$args);
    };
}
```

class Stack

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(array $entries = []) Creates the stack and will `fill` it with the given entries.

Stack fill($entries) Pushes each entry onto the stack in the order defined.

Stack 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.

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

Stack on($name, $mw, $sort = 0) Simply an alias of `push`; however, the argument order lends it nicer for adding/replacing named middleware.

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

Stack 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.

bool has($name) Returns true if a named middleware exists in the stack

array get($name) Returns a 3-tuple of an entry like so: `[$entry, $sort, $name]`. This will throw an exception if no entry is found.

Stack toTop($name) Moves the given entry to the top of its stack to be executed first.

Stack toBottom($name) Moves the given entry to the bottom of its stack to be executed last.

array toArray() 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.

class Link

Represents a link in the middleware chain. A link instance is passed to every middleware as the last parameter which allows the next middleware to be called. See *Avanced Usage* for more details.

__construct($mw, Context $ctx, Link $next = null) Creates a link. If `$next` is provided, then the created link will be the new head of that linked list.

__invoke(...$params) Invokes the middleware. It forwards the params to the middleware and additionaly adds the next link to the end of argument list for the middleware.

chain($mw) Creates a new link to be the head of the current list of links. The context is copied from the current link.

getContext() returns the context instance apart of the link.

class Link\ContainerLink

Extends the Link class and implements the Psr\Container\ContainerInterface and ArrayAccess. Keep in mind that it offers read-only access, so setting and deleting offsets will cause an exception to be thrown.

interface Context

Represents the middleware context utilized by the internal system.

getInvoke() Returns the invoker configured for this context.

class Context\StdContext implements Context

The default context for the mw system. It simply holds the a value to the invoker for custom invocation.

__construct($invoke = 'call_user_func')

class Context\ContainerContext implements Context

Provides psr container integeration by allowing the context to act like a psr container and it provides container invocation by default.

View the cookbook/container-middleware for example on this.

__construct(ContainerInterface $container, $invoke = null) The psr container and an optional invoker if you don't want to use the `containerAwareInvoke`

CHAPTER 5

Troubleshooting

Here are few common errors and how to resolve them

"Cannot invoke last middleware in chain. No middleware returned a result." NoResultException

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 add a *guard* middleware when you compose your middleware set to provide custom error messages.

"Middleware cannot be invoked because it does not contain the '' method"

This exception is thrown when using the `methodInvoke` for composing your middleware. This means that one of the middleware on your stack doesn't have the proper method to be called.

To fix this, you should check your middleware stack and verify that every middleware has the proper method. The stack trace should also show you which class instance caused the problem to help you track down the problem.