Enhancing Monolog

Configuring loggers effortlessly with Monolog Cascade

Raphaël Antonmattei Follow
Jul 14, 2015 · 6 min read

Monolog is awesome! We've been using it for quite some time and it clearly made our life easier at The Orchard given our hybrid infrastructure (cloud/on premise).

We use many available handlers: syslog, streams, Loggly, etc. One thing we noticed though is that configuring all those *loggers* and *handlers* was tedious and accessing those *logger* objects from different locations of our code base was also a bit cumbersome.

Introducing **Monolog Cascade**, a Monolog extension to configure and access your loggers from anywhere.

Excessive Boilerplate code

There were 2 main issues with our use of bare-bones Monolog:

- accessing the *logger* objects was difficult as you often had to pass those objects around in order to use them at the desired locations unless duplicating a lot of code to set them up.
- configuring all the *formatters*, *handlers* and *loggers* was a lot of code to include into some bootstrap script wherever you needed to log.

Here is what your life was like before Cascade:

```
1  <?php
```

```php
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Handler\LogglyHandler;

// Create a formatter
$formatter = new LineFormatter(
    '%datetime% > %level_name% > %message% %context% %extra%'."\n",
    'Y n j, g:i a'
);

// file handler
$fileHandler = new StreamHandler('path/to/logs/my_app.log', Logger::INFO);
$fileHandler->setFormatter($formatter);

// stdout handler
$stdOutHandler = new StreamHandler('php://stdout', Logger::DEBUG);
$stdOutHandler->setFormatter($formatter);

// Add Loggly handler
$logglyHandler = new LogglyHandler('xxxx-xxxx-xxxxxxxx', Logger::INFO);
$logglyHandler->addTag('cascade');
$logglyHandler->addTag('waterfall');

// Create the logger
$logger = new Logger('my_logger');

// Add handlers
$logger->pushHandler($fileHandler);
$logger->pushHandler($stdOutHandler);
$logger->pushHandler($logglyHandler);

// You can now use your logger
$logger->addInfo('My logger is now ready');

function doStuff($logger)
{
    try {
        doOtherThings();
        $logger->info('Finally logging something');
    } catch (\Exception $e) {
        $logger->error($e->getMessage());
    }
}

// or
```

```php
$app = new MyApp($logger);
$app->doSomethingThatLogs();

// or
SomeOtherComponent::somethingElseThatLogs($logger);

// etc.
```

Before.php hosted with ❤️ by GitHub view raw

Configuring Formatters, Handlers and Loggers was a long process and you had to pass logger objects around

In the example above we set up a single *logger* with 3 *handlers* and one *formatter*. Imagine if you had 3 times what's above to set up…

Ease the pain

The genesis of that project came during one of our monthly *fixathon/hackathon*. The first iteration of the project was a small wrapper around Monolog's *Registry* class so we could instantiate a *Logger* object on the fly and access it from anywhere. That would take care of the first pain point.

The Cascade class wraps Monolog's Registry and adds lazy loading.

Here, the first call to getLogger actually instantiates a *Logger* object and pushes it to Monolog's *Registry*.

This was nice but did only alleviate a tiny bit of the hassle of configuring your handlers, formatters, etc. You still needed to so "manually".

Stronger painkiller

As our Engineering team was diving deeper into writing Python code, a lot of ideas were borrowed from by the logging.config Python module.

Having a single config file where you can define options for your *loggers* was the goal. Not only you can define your components but with a hierarchical structure offered by *Yaml* or *JSON* formats, you can define dependencies between those and reuse them — i.e. *handlers* can share the same *formatter* — multiple times.

Here is a sample *Yaml* config file that is parseable by Monolog Cascade:

Formatters and Handlers are defined once and shared by Loggers

The immediate benefit is that the config file (whether it is *Yaml* or *JSON*) is much easier to read and to understand than plain Php code. Adding a few *loggers* or *handlers* takes no time and the file is still clean.

Then setting up your *loggers* becomes much shorter.

Cascade will read the file, parse it and work its magic to load all required objects in sequence and eventually set up you *loggers* and stick them into Monolog's *Registry*.

Now let's see what's under the hood…

Loading, Parsing, Resolving

Let's dive into some implementation details. The main idea behind this project was to provide a library that is simple to use and could support most of Monolog's features. The simplicity factor resides in the fact that by only checking out the constructor declaration in a *formatter* or *handler* class, you were able to write a valid config file and get the ball rolling within minutes.

Like mentioned above this project was a side project and although the need for that package was real from an engineering perspective, it was hard to make a business case and allocate lots of resources on such project. So, with that in mind we wanted to implement something simple but yet quite robust. Thus we leveraged a few proven and well tested Php libraries.

Symfony to the rescue

Symfony is a very mature and well known *MVC* framework in the Php community. Since version 2.0, they have broken down the library into standalone usable components. Better yet, they are all independently *Composer-installable*. It turned out *Symfony*

already had a lot of what was needed for Monolog Cascade to take form: *loaders*, *parsers* and *option resolvers*.

Having those components right off the bat helped a lot not only getting the first version emerged in a timely manner but also keeping each layer of the package nicely *separated* and therefore easier to reuse and test.

Most of the challenges of the implementation were around extracting constructor parameters of *handler* and *formatter* classes to then feed option resolvers in order to validate those parameters coming from the config file.

Mirror, Mirror on the Wall

Figuring out whether or not parameter name and value were valid was essential to the robustness and the maintainability of the package. This is the part where I had wished Php had something similar to Python's *kwargs*. It would have been so much easier… Php arrays kind of emulates that behavior, but Monolog's *formatters* and *handlers* have real arguments (value only), not just a single associative array with a bunch of key/value options.

The simplest way to address that issue was using Php's *Reflection API*. *Reflection* is a strategy for metaprogramming. It allows type introspections of *interfaces*, *classes*, and *methods* so you can *reverse-engineer* those components and instantiate objects with proper arguments at runtime dynamically for instance.

Reflection allows us to find out what argument is required or optional and pull the default value from the constructor declaration (if any)

Reflection is used extensively in Cascade to inspect constructor declarations and build an array of valid parameters out of the parsed config file before instantiating the object. That works pretty well. This approach was intended to minimize updating the package every time a new *Formatter* or *Handler* is added to Monolog. Cascade will just pick it up as long as the class exists in *Monolog*. Your config file just has to follow basic rules.

From constructor declaration to config file

The *Formatter* section must have a *class* parameter. Cascade will then try to match all the other parameters against the constructor args it has found. Parameters will be placed into 2 buckets: constructor parameters and extra parameters (i.e. everything else that is not constructor's).

Let's go over Monolog's *LineFormatter* class to see how it would be configured using Cascade. Here is the constructor declaration along with the corresponding Yaml section to configure that *formatter*:

You would know how to build your configuration file just by looking up the constructor declaration of you handler/formatter class.

As you can see in the code samples above the *formatters* section of the Yaml file matches the constructor declaration. Note that Cascade will convert underscore separated param names (if this is your preferred syntax) to camel case internally and then figure out param positions to call the constructor method appropriately.

Configuring Handlers works the same way. There is only an additional "*formatter*" reserved keyword if you would like your handler to use a specific *formatter*.

Processors will work pretty much the same way but are not yet implemented.
Update (09/02/2015): implemented as of 0.1.0

Would You Like to Know More?

There is one thing we left off in the previous section. This is those mysterious extra parameters. While solely using constructor args will take care of 90% of your use cases, there are times when you want to do additional things with your loggers prior to using them. Those things include calling a method to set some values (that are not "settable" from the constructor) or you have a more complex case where you would like to use custom *handlers* and *formatters* and call a bunch of other things before using you logger objects.

Cascade adds the ability to implement those custom behaviors at runtime if needed. Here is an example of configuring a custom *Formatter* using an extra parameter:

By adding an extra parameter, Cascade will try to resolve it in that order:

- it will look for a public method in the formatter class with the name of that parameter and call that method with the parameter value
- it will look for a public member and set it with the parameter value
- it will look for a custom handler (like the closure in the code sample above) and execute it passing the instance (Formatter or Handler) along with the parameter value.

Although this implementation is not perfect, it is a cheap way of introducing more flexible behavior should you need them.

Note to Symfony users:
You might want to use MonologBundle as it integrates better with your favorite framework.

Limitations

Some *Handlers* use *Dependency Injection* upon instantiation (most of the *Database* and *Key/Value* store do). This scenario is a bit hard to support with a simple config file and would require some more head scratching to figure out.
Update (08/24/2015): this has been fixed in 0.2.0

What's Next?

This first release is still green and early stage. As we get more usage, we'll continue to add support for more *Handlers*, *Formatters* and *Processors* and other features that *Monolog* introduces.

Coming Up

- support for more config file formats: *.ini*, etc.
- support for namespaced *Loggers* with message propagation (through handler inheritance) so children *Loggers* can log messages using parent's *Handlers*

Feedback / Contributions

If you have feedback, questions, or simply want to contribute, we'd love to hear from you. You can send pull request directly through Github.

Raphaël
Currently listening *Roses, dEUS*

Thanks to Michael Ortali.

Monolog Tech PHP