NAME

Log::Any − Bringing loggers and listeners together

VERSION

version 1.708

SYNOPSIS

In a CPAN or other module:

```
package Foo;
use Log::Any qw($log);

# log a string
$log->error("an error occurred");

# log a string and some data
$log->info("program started",
    {progname => $0, pid => $$, perl_version => $]});

# log a string and data using a format string
$log->debugf("arguments are: %s", \@_);

# log an error and throw an exception
die $log->fatal("a fatal error occurred");
```

In a Moo/Moose−based module:

```
package Foo;
use Log::Any ();
use Moo;

has log => (
    is => 'ro',
    default => sub { Log::Any->get_logger },
);
```

In your application:

```
use Foo;
use Log::Any::Adapter;

# Send all logs to Log::Log4perl
Log::Any::Adapter->set('Log4perl');

# Send all logs to Log::Dispatch
my $log = Log::Dispatch->new(outputs => [[ ... ]]);
Log::Any::Adapter->set( 'Dispatch', dispatcher => $log );

# See Log::Any::Adapter documentation for more options
```

DESCRIPTION

`Log::Any` provides a standard log production API for modules. Log::Any::Adapter allows applications to choose the mechanism for log consumption, whether screen, file or another logging mechanism like Log::Dispatch or Log::Log4perl.

Many modules have something interesting to say. Unfortunately there is no standard way for them to say it − some output to STDERR, others to `warn`, others to custom file logs. And there is no standard way to get a module to start talking − sometimes you must call a uniquely named method, other times set a package variable.

This being Perl, there are many logging mechanisms available on CPAN. Each has their pros and cons. Unfortunately, the existence of so many mechanisms makes it difficult for a CPAN author to commit his/her users to one of them. This may be why many CPAN modules invent their own logging or choose not to log at all.

To untangle this situation, we must separate the two parts of a logging API. The first, *log production*, includes methods to output logs (like `$log->debug`) and methods to inspect whether a log level is activated (like `$log->is_debug`). This is generally all that CPAN modules care about. The second, *log consumption*, includes a way to configure where logging goes (a file, the screen, etc.) and the code to send it there. This choice generally belongs to the application.

A CPAN module uses `Log::Any` to get a log producer object. An application, in turn, may choose one or more logging mechanisms via Log::Any::Adapter, or none at all.

`Log::Any` has a very tiny footprint and no dependencies beyond Perl 5.8.1, which makes it appropriate for even small CPAN modules to use. It defaults to 'null' logging activity, so a module can safely log without worrying about whether the application has chosen (or will ever choose) a logging mechanism.

See <http://www.openswartz.com/2007/09/06/standard-logging-api/> for the original post proposing this module.

LOG LEVELS

`Log::Any` supports the following log levels and aliases, which is meant to be inclusive of the major logging packages:

```
trace
debug
info (inform)
notice
warning (warn)
error (err)
critical (crit, fatal)
alert
emergency
```

Levels are translated as appropriate to the underlying logging mechanism. For example, log4perl only has six levels, so we translate 'notice' to 'info' and the top three levels to 'fatal'. See the documentation of an adapter class for specifics.

CATEGORIES

Every logger has a category, generally the name of the class that asked for the logger. Some logging mechanisms, like log4perl, can direct logs to different places depending on category.

PRODUCING LOGS (FOR MODULES)

Getting a logger

The most convenient way to get a logger in your module is:

```
use Log::Any qw($log);
```

This creates a package variable *$log* and assigns it to the logger for the current package. It is equivalent to

```
our $log = Log::Any->get_logger;
```

In general, to get a logger for a specified category:

```
my $log = Log::Any->get_logger(category => $category)
```

If no category is specified, the calling package is used.

A logger object is an instance of Log::Any::Proxy, which passes on messages to the Log::Any::Adapter handling its category.

If the `proxy_class` argument is passed, an alternative to Log::Any::Proxy (such as a subclass) will be instantiated and returned instead. The argument is automatically prepended with "Log::Any::Proxy::". If instead you want to pass the full name of a proxy class, prefix it with a "+". E.g.

```
# Log::Any::Proxy::Foo
my $log = Log::Any->get_logger(proxy_class => 'Foo');

# MyLog::Proxy
my $log = Log::Any->get_logger(proxy_class => '+MyLog::Proxy');
```

Logging

To log a message, pass a single string to any of the log levels or aliases. e.g.

```
$log->error("this is an error");
$log->warn("this is a warning");
$log->warning("this is also a warning");
```

The log string will be returned so that it can be used further (e.g. for a `die` or `warn` call).

You should **not** include a newline in your message; that is the responsibility of the logging mechanism, which may or may not want the newline.

If you want to log additional structured data alongside with your string, you can add a single hashref after your log string. e.g.

```
$log->info("program started",
    {progname => $0, pid => $$, perl_version => $]});
```

If the configured Log::Any::Adapter does not support logging structured data, the hash will be converted to a string using Data::Dumper.

There are also versions of each of the logging methods with an additional "f" suffix (`infof`, `errorf`, `debugf`, etc.) that format a list of arguments. The specific formatting mechanism and meaning of the arguments is controlled by the Log::Any::Proxy object.

```
$log->errorf("an error occurred: %s", $@);
$log->debugf("called with %d params: %s", $param_count, \@params);
```

By default it renders like `sprintf`, with the following additional features:

- Any complex references (like `\@params` above) are automatically converted to single-line strings with Data::Dumper.
- Any undefined values are automatically converted to the string "<undef>".

Log level detection

To detect whether a log level is on, use "is_" followed by any of the log levels or aliases. e.g.

```
if ($log->is_info()) { ... }
$log->debug("arguments are: " . Dumper(\@_))
    if $log->is_debug();
```

This is important for efficiency, as you can avoid the work of putting together the logging message (in the above case, stringifying `@_`) if the log level is not active.

The formatting methods (`infof`, `errorf`, etc.) check the log level for you.

Some logging mechanisms don't support detection of log levels. In these cases the detection methods will always return 1.

In contrast, the default logging mechanism – Null – will return 0 for all detection methods.

Log context data

`Log::Any` supports logging context data by exposing the `context` hashref. All the key/value pairs added to this hash will be printed with every log message. You can localize the data so that it will be removed again automatically at the end of the block:

```
$log->context->{directory} = $dir;
for my $file (glob "$dir/*") {
    local $log->context->{file} = basename($file);
    $log->warn("Can't read file!") unless -r $file;
}
```

This will produce the following line:

```
Can't read file! {directory => '/foo',file => 'bar'}
```

If the configured Log::Any::Adapter does not support structured data, the context hash will be converted to a string using Data::Dumper, and will be appended to the log message.

Setting an alternate default logger

When no other adapters are configured for your logger, `Log::Any` uses the `default_adapter`. To choose something other than Null as the default, either set the `LOG_ANY_DEFAULT_ADAPTER` environment variable, or pass it as a parameter when loading `Log::Any`

```
use Log::Any '$log', default_adapter => 'Stderr';
```

The name of the default class follows the same rules as used by Log::Any::Adapter.

To pass arguments to the default adapter's constructor, use an arrayref:

```
use Log::Any '$log', default_adapter => [ 'File' => '/var/log/mylog.log' ];
```

When a consumer configures their own adapter, the default adapter will be overridden. If they later remove their adapter, the default adapter will be used again.

Configuring the proxy

Any parameters passed on the import line or via the `get_logger` method are passed on to the Log::Any::Proxy constructor.

```
use Log::Any '$log', filter => \&myfilter;
```

Testing

Log::Any::Test provides a mechanism to test code that uses `Log::Any`.

CONSUMING LOGS (FOR APPLICATIONS)

Log::Any provides modules with a Log::Any::Proxy object, which is the log producer. To consume its output and direct it where you want (a file, the screen, syslog, etc.), you use Log::Any::Adapter along with a destination-specific subclass.

For example, to send output to a file via Log::Any::Adapter::File, your application could do this:

```
use Log::Any::Adapter ('File', '/path/to/file.log');
```

See the Log::Any::Adapter documentation for more details.

Q & A

Isn't Log::Any just yet another logging mechanism?

No. `Log::Any` does not include code that knows how to log to a particular place (file, screen, etc.) It can only forward logging requests to another logging mechanism.

Why don't you just pick the best logging mechanism, and use and promote it?

Each of the logging mechanisms have their pros and cons, particularly in terms of how they are configured. For example, log4perl offers a great deal of power and flexibility but uses a global and potentially heavy configuration, whereas Log::Dispatch is extremely configuration-light but doesn't handle categories. There is also the unnamed future logger that may have advantages over either of these two, and all the custom in-house loggers people have created and cannot (for whatever reason) stop using.

Is it safe for my critical module to depend on Log::Any?

Our intent is to keep `Log::Any` minimal, and change it only when absolutely necessary. Most of the ''innovation'', if any, is expected to occur in `Log::Any::Adapter`, which your module should not have to depend on (unless it wants to direct logs somewhere specific). `Log::Any` has no non-core

dependencies.

Why doesn't Log::Any use *insert modern Perl technique*?

To encourage CPAN module authors to adopt and use `Log::Any`, we aim to have as few dependencies and chances of breakage as possible. Thus, no `Moose` or other niceties.

AUTHORS

- Jonathan Swartz <swartz@pobox.com>
- David Golden <dagolden@cpan.org>
- Doug Bell <preaction@cpan.org>
- Daniel Pittman <daniel@rimspace.net>
- Stephen Thirlwall <sdt@cpan.org>

CONTRIBUTORS

- bj5004 <bartosz.jakubski@hurra.com>
- cm-perl <cm−perl@users.noreply.github.com>
- Jonathan <jjrs.pam+github@gmail.com>
- Karen Etheridge <ether@cpan.org>
- Konstantin S. Uvarin <khedin@gmail.com>
- Lucas Kanashiro <kanashiro.duarte@gmail.com>
- Maros Kollar <maros.kollar@geizhals.at>
- Maxim Vuets <maxim.vuets@booking.com>
- mephinet <mephinet@gmx.net>
- Michael Conrad <mconrad@intellitree.com>
- Nick Tonkin <1nickt@users.noreply.github.com>
- Paul Durden <alabamapaul@gmail.com>
- Philipp Gortan <philipp.gortan@apa.at>
- Phill Legault <saladdayllc@gmail.com>
- Shlomi Fish <shlomif@shlomifish.org>

COPYRIGHT AND LICENSE