NAME

Log::Any::Adapter::Development – Manual for developing new Log::Any adapters

VERSION

version 1.708

SYNOPSIS

The adapter module:

```
package Log::Any::Adapter::YAL;
use strict;
use warnings;
use Log::Any::Adapter::Util ();
use base qw(Log::Any::Adapter::Base);

# Optionally initialize object, e.g. for delegation
#
sub init {
    my ($self) = @_;

    $self->{attr} = ...;
}

# Create logging methods: debug, info, etc.
#
foreach my $method ( Log::Any::Adapter::Util::logging_methods() ) {
    no strict 'refs';
    *$method = sub { ... };
}

# or, support structured logging instead
sub structured {
    my ($self, $level, $category, @args) = @_;
    # ... process and log all @args
}


# Create detection methods: is_debug, is_info, etc.
#
foreach my $method ( Log::Any::Adapter::Util::detection_methods() ) {
    no strict 'refs';
    *$method = sub { ... };
}
```

and the application:

```
Log::Any->set_adapter('YAL');
```

DESCRIPTION

This document describes how to implement a new Log::Any adapter.

The easiest way to start is to look at the source of existing adapters, such as Log::Any::Adapter::Log4perl and Log::Any::Adapter::Dispatch.

NAMING

If you are going to publicly release your adapter, call it 'Log::Any::Adapter::*something*' so that users can use it with

```
Log::Any->set_adapter(I<something>);
```

If it's an internal driver, you can call it whatever you like and use it like

```
Log::Any->set_adapter('+My::Log::Adapter');
```

BASE CLASS

All adapters must directly or indirectly inherit from Log::Any::Adapter::Base.

LOG LEVELS

Log::Any supports the following log levels:

If the logging mechanism used by your adapter supports different levels, it's your responsibility to map them appropriately when you implement the logging and detection methods described below. For example, if your mechanism only supports "debug", "normal" and "fatal" levels, you might map the levels like this:

METHODS

Constructor

The constructor (`new`) is provided by Log::Any::Adapter::Base. It will:

At this point, overriding the default constructor is not supported. Hopefully it will not be needed.

The constructor is called whenever a log object is requested. e.g. If the application initializes Log::Any like so:

```
Log::Any->set_adapter('Log::YAL', yal_object => $yal, depth => 3);
```

and then a class requests a logger like so:

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

Then `$log` will be populated with the return value of:

```
Log::Any::Adapter::Yal->new(yal_object => $yal, depth => 3, category => 'Foo');
```

This is memoized, so if the same category should be requested again (e.g. through a separate `get_logger` call, the same object will be returned. Therefore, you should try to avoid anything non-deterministic in your "init" function.

Logging methods

The following methods have no default implementation, and MUST be defined by your subclass, unless your adapter supports "Structured logging":

These methods must log a message at the specified level.

To help generate these methods programmatically, you can get a list of the sub names with the Log::Any::Adapter::Util::logging_methods function.

Log-level detection methods (required)

The following methods have no default implementation, and MUST be defined by your subclass:

These methods must return a boolean indicating whether the specified level is active, i.e. whether the adapter is listening for messages of that level.

To help generate these methods programmatically, you can get a list of the sub names with the Log::Any::Adapter::Util::detection_methods function.

Structured logging

Your adapter can choose to receive structured data instead of a string. In this case, instead of implementing all the "Logging methods", you define a single method called `structured`. The method receives the log level, the category, and all arguments that were passed to the logging function, so be prepared to not only handle strings, but also hashrefs, arrayrefs, coderefs, etc.

Aliases

Aliases (e.g. "err" for "error") are handled by Log::Any::Proxy and will call the corresponding real name in your adapter class. You do not need to implement them in your adapter.

Optional methods

The following methods have no default implementation but MAY be provided by your subclass:

init This is called after the adapter object is created and blessed into your class. Perform any necessary validation or initialization here. For example, you would use `init` to create a logging object for delegation, or open a file or socket, etc.

Support methods

The following Log::Any::Adapter::Base method may be useful for defining adapters via delegation:

delegate_method_to_slot ($slot, `$method`, `$adapter_method`)

Handle the specified `$method` by calling `$adapter_method` on the object contained in `$self->{$slot}`.

See Log::Any::Adapter::Dispatch and Log::Any::Adapter::Log4perl for examples of usage.

The following Log::Any::Adapter::Util functions give you a list of methods that you need to implement. You can get logging methods, detection methods or both:

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>

COPYRIGHT AND LICENSE