Logging

# Introduction

To help you learn more about what's happening within your application, Laravel provides robust logging services that allow you to log messages to files, the system error log, and even to Slack to notify your entire team.

Under the hood, Laravel utilizes the Monolog library, which provides support for a variety of powerful log handlers. Laravel makes it a cinch to configure these handlers, allowing you to mix and match them to customize your application's log handling.

# Configuration

All of the configuration for your application's logging system is housed in the `config/logging.php` configuration file. This file allows you to configure your application's log channels, so be sure to review each of the available channels and their options. We'll review a few common options below.

By default, Laravel will use the `stack` channel when logging messages. The `stack` channel is used to aggregate multiple log channels into a single channel. For more information on building stacks, check out the documentation below.

Configuring The Channel Name

By default, Monolog is instantiated with a "channel name" that matches the current environment, such as `production` or `local`. To change this value, add a `name` option to your channel's configuration:

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

Available Channel Drivers

Name	Description
`stack`	A wrapper to facilitate creating "multi-channel" channels
`single`	A single file or path based logger channel (`StreamHandler`)
`daily`	A `RotatingFileHandler` based Monolog driver which rotates daily
`slack`	A `SlackWebhookHandler` based Monolog driver
`papertrail`	A `SyslogUdpHandler` based Monolog driver
`syslog`	A `SyslogHandler` based Monolog driver
`errorlog`	A `ErrorLogHandler` based Monolog driver
`monolog`	A Monolog factory driver that may use any supported Monolog handler
`custom`	A driver that calls a specified factory to create a channel

Check out the documentation on advanced channel customization to learn more about the `monolog` and `custom` drivers.

Configuring The Single and Daily Channels

The `single` and `daily` channels have three optional configuration options: `bubble`, `permission`, and `locking`.

Name	Description	Default
`bubble`	Indicates if messages should bubble up to other channels after being handled	`true`
`permission`	The log file's permissions	`0644`
`locking`	Attempt to lock the log file before writing to it	`false`

Configuring The Papertrail Channel

The `papertrail` channel requires the `url` and `port` configuration options. You can obtain these values from Papertrail.

Configuring The Slack Channel

The `slack` channel requires a `url` configuration option. This URL should match a URL for an incoming webhook that you have configured for your Slack team.

Building Log Stacks

As previously mentioned, the `stack` driver allows you to combine multiple channels into a single log channel. To illustrate how to use log stacks, let's take a look at an example configuration that you might see in a production application:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],

    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],
```

Let's dissect this configuration. First, notice our `stack` channel aggregates two other channels via its `channels` option: `syslog` and `slack`. So, when logging messages, both of these channels will have the opportunity to log the message.

Log Levels

Take note of the `level` configuration option present on the `syslog` and `slack` channel configurations in the example above. This option determines the minimum "level" a message must be in order to be logged by the channel. Monolog, which powers Laravel's logging services, offers all of the log levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info**, and **debug**.

So, imagine we log a message using the `debug` method:

```
Log::debug('An informational message.');
```

Given our configuration, the `syslog` channel will write the message to the system log; however, since the error message is not `critical` or above, it will not be sent to Slack. However, if we log an `emergency` message, it will be sent to both the system log and Slack since the `emergency` level is above our minimum level threshold for both channels:

```
Log::emergency('The system is down!');
```

Writing Log Messages

You may write information to the logs using the `Log` facade. As previously mentioned, the logger provides the eight logging levels defined in the RFC 5424 specification: **emergency**, **alert**, **critical**, **error**, **warning**, **notice**, **info** and **debug**:

```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
```

```
Log::info($message);
Log::debug($message);
```

So, you may call any of these methods to log a message for the corresponding level. By default, the message will be written to the default log channel as configured by your `config/logging.php` configuration file:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\User;
use Illuminate\Support\Facades\Log;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        Log::info('Showing user profile for user: '.$id);

        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

Contextual Information

An array of contextual data may also be passed to the log methods. This contextual data will be formatted and displayed with the log message:

```
Log::info('User failed to login.', ['id' => $user->id]);
```

Writing To Specific Channels

Sometimes you may wish to log a message to a channel other than your application's default channel. You may use the `channel` method on the `Log` facade to retrieve and log to any channel defined in your configuration file:

```
Log::channel('slack')->info('Something happened!');
```

If you would like to create an on-demand logging stack consisting of multiple channels, you may use the `stack` method:

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

Advanced Monolog Channel Customization

Customizing Monolog For Channels

Sometimes you may need complete control over how Monolog is configured for an existing channel. For example, you may want to configure a custom Monolog `FormatterInterface` implementation for a given channel's handlers.

To get started, define a `tap` array on the channel's configuration. The `tap` array should contain a list of classes that should have an opportunity to customize (or "tap" into) the Monolog instance after it is created:

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
    'path' => storage_path('logs/laravel.log'),
    'level' => 'debug',
],
```

Once you have configured the `tap` option on your channel, you're ready to define the class that will customize your Monolog instance. This class only needs a single method: `__invoke`, which receives an `Illuminate\Log\Logger` instance. The `Illuminate\Log\Logger` instance proxies all method calls to the underlying Monolog instance:

```
<?php
```

```
namespace App\Logging;

class CustomizeFormatter
{
    /**
     * Customize the given logger instance.
     *
     * @param  \Illuminate\Log\Logger  $logger
     * @return void
     */
    public function __invoke($logger)
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(...);
        }
    }
}
```

All of your "tap" classes are resolved by the service container, so any constructor dependencies they require will automatically be injected.

Creating Monolog Handler Channels

Monolog has a variety of available handlers. In some cases, the type of logger you wish to create is merely a Monolog driver with an instance of a specific handler. These channels can be created using the `monolog` driver.

When using the `monolog` driver, the `handler` configuration option is used to specify which handler will be instantiated. Optionally, any constructor parameters the handler needs may be specified using the `with` configuration option:

```
'logentries' => [
    'driver'  => 'monolog',
    'handler' => Monolog\Handler\SyslogUdpHandler::class,
    'with' => [
        'host' => 'my.logentries.internal.datahubhost.company.com',
        'port' => '10000',
    ],
],
```

Monolog Formatters

When using the `monolog` driver, the Monolog `LineFormatter` will be used as the default formatter. However, you may customize the type of formatter passed to the handler using the `formatter` and `formatter_with` configuration options:

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

If you are using a Monolog handler that is capable of providing its own formatter, you may set the value of the `formatter` configuration option to `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

Creating Channels Via Factories

If you would like to define an entirely custom channel in which you have full control over Monolog's instantiation and configuration, you may specify a `custom` driver type in your `config/logging.php` configuration file. Your configuration should include a `via` option to point to the factory class which will be invoked to create the Monolog instance:

```
'channels' => [
    'custom' => [
        'driver' => 'custom',
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

Once you have configured the `custom` channel, you're ready to define the class that will create your Monolog instance. This class only needs a single method: `__invoke`, which should return the Monolog instance:

```php
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Create a custom Monolog instance.
     *
     * @param  array  $config
     * @return \Monolog\Logger
     */
    public function __invoke(array $config)
    {
        return new Logger(...);
    }
}
```

and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.