How to Log and Monitor Errors in Laravel with Monolog

Error logging is one of the most important parts of maintaining a reliable Laravel application. While simple `dd()` statements can help in development, production apps need structured, configurable, and persistent logs. Laravel uses **Monolog** under the hood, which is a powerful logging library that supports multiple channels, handlers, and integrations. In this article, we'll learn how to configure Monolog, log errors to different destinations, monitor logs in production, and apply best practices for scaling logging in real-world projects.

Laravel Logging Basics

Laravel provides a unified `Log` facade for writing logs. By default, logs are written to `storage/logs/laravel.log`. The logging configuration is located in `config/logging.php` and defines channels like `stack`, `single`, `daily`, `slack`, and more.

```
use Illuminate\Support\Facades\Log;

Log::info('User visited dashboard.');
Log::warning('Payment processing is slow.');
Log::error('Unable to connect to database.');Code language: PHP (php)
```

Each log entry has a level (`info`, `warning`, `error`, `critical`, etc.) defined by the RFC 5424 standard. This helps filter logs when monitoring large systems.

Configuring Log Channels

Channels define where and how logs are stored. Laravel supports multiple channels at once using the `stack` driver. Common drivers include `single` (one file), `daily` (rotating logs), `slack`, `syslog`, and custom Monolog handlers.

```
// config/logging.php
return [
    'default' => env('LOG_CHANNEL', 'stack'),

    'channels' => [
        'stack' => [
            'driver' => 'stack',
            'channels' => ['daily', 'slack'],
        ],

        'daily' => [
            'driver' => 'daily',
            'path' => storage_path('logs/laravel.log'),
            'level' => 'debug',
            'days' => 14,
        ],

        'slack' => [
            'driver' => 'slack',
            'url' => env('LOG_SLACK_WEBHOOK_URL'),
            'level' => 'critical',
        ],
    ],
];Code language: PHP (php)
```

This configuration sends all logs to daily files and sends `critical` level errors to Slack. This way, developers are alerted instantly for severe issues, while other logs are stored for

analysis.

Custom Monolog Handlers

Sometimes you want to send logs to third-party services (like Logstash, Graylog, or CloudWatch). Laravel lets you define custom Monolog handlers by extending `tap` in `logging.php`.

```
// app/Logging/CustomizeFormatter.php
namespace App\Logging;

use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    public function __invoke($logger)
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(new LineFormatter(
                "[%datetime%] %channel%.%level_name%: %message%
%context% %extra%\n"
            ));
        }
    }
}Code language: PHP (php)
```

Then register it in `config/logging.php`:

```
'daily' => [
    'driver' => 'daily',
    'path' => storage_path('logs/laravel.log'),
    'level' => 'debug',
    'days' => 14,
```

```
    'tap' => [App\Logging\CustomizeFormatter::class],
],Code language: PHP (php)
```

This allows you to format log messages consistently across all environments, making parsing easier for log aggregators.

Context and Structured Logging

Logs become far more useful when you include structured context data, like user IDs, request IDs, or payloads.

```
Log::error('Order failed to process.', [
    'order_id' => $order->id,
    'user_id' => $order->user_id,
]);Code language: PHP (php)
```

This context can be filtered in monitoring tools, making it easier to trace specific errors and build dashboards.

Monitoring Logs in Production

For production, logs should be centralized and monitored. Common approaches include:

- **Slack / Teams Alerts** for critical errors.
- **Log Aggregators** (ELK stack, Graylog, Datadog, Papertrail, Sentry).
- **Cloud Logging** (AWS CloudWatch, GCP Logging).
- **Custom Dashboards** with daily log summaries.

By centralizing logs, you can set up alerting, search logs by metadata, and ensure compliance for audits.

Best Practices for Logging

- Use **appropriate log levels** (`info` for events, `warning` for recoverable issues, `error` for failures).
- **Rotate logs** with the `daily` driver to avoid large files.
- **Don’t log sensitive data** (passwords, credit card numbers).
- Include **context** like user ID, request ID, or environment.
- Integrate with **alerting tools** for critical errors.

Monolog vs Telescope vs Ray

Tool	Best For	Environment	Output
Monolog	Persistent error logging	Local + Production	Files, Slack, Syslog, Cloud
Telescope	Monitoring requests, jobs, queries	Local + Staging	Web dashboard
Ray	Quick, interactive debugging	Local Development	Desktop app

Monolog is your backbone for logging across environments. Ray and Telescope are great companions for debugging and monitoring, but Monolog handles the long-term storage and integration with external systems.

Wrapping Up

Laravel's Monolog integration makes logging flexible and production-ready. We covered basic logging, channel configuration, custom handlers, structured context, monitoring, and best practices. Combined with Ray and Telescope, Monolog gives you full visibility into your app across local development and production.

What's Next

Continue learning about monitoring and optimization with these articles:

- Using Laravel Telescope to Debug Performance Issues
- Debugging Laravel Applications with Ray and Telescope
- Query Performance Tuning in Laravel + MySQL