Advanced Logging and Monitoring in Laravel with Monolog

In the previous article, we covered the basics of logging with Monolog in Laravel. Now, let’s go deeper into advanced logging and monitoring strategies. In large-scale projects, simply writing logs to files is not enough. You need structured logs, log rotation, external storage, and integration with observability platforms. In this article, we’ll configure Monolog for production-ready logging, integrate with services like AWS CloudWatch, explore asynchronous logging, and apply best practices for scaling.

Advanced Log Channels

Laravel supports multiple channels combined in a stack. You can mix `daily` logs with cloud-based services and instant alerts. Let’s look at a custom `stack` channel that combines multiple destinations:

```
// config/logging.php
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['daily', 'slack', 'cloudwatch'],
        'ignore_exceptions' => false,
    ],

    'cloudwatch' => [
        'driver' => 'monolog',
        'handler' => Monolog\Handler\StreamHandler::class,
        'with' => [
            'stream' => 'php://stdout',
        ],
        'formatter' => Monolog\Formatter\JsonFormatter::class,
    ],
],Code language: PHP (php)
```

This setup logs daily to disk, sends critical alerts to Slack, and streams JSON logs to AWS CloudWatch. The JSON format is preferred for production because it integrates better with log aggregators.

Asynchronous Logging with Queues

Heavy logging can slow down requests. To reduce impact, send logs asynchronously using Laravel’s queue system. Here’s how you can wrap logging inside a job:

```
// app/Jobs/AsyncLog.php
namespace App\Jobs;

use Illuminate\Support\Facades\Log;

class AsyncLog extends Job
{
    public $message;
    public $level;
    public $context;

    public function __construct($message, $level = 'info', $context =
[])
    {
        $this->message = $message;
        $this->level = $level;
        $this->context = $context;
    }

    public function handle(): void
    {
        Log::{$this->level}($this->message, $this->context);
    }
}
```
Code language: PHP (php)

Dispatch logs asynchronously:

```
dispatch(new \App\Jobs\AsyncLog(
    'Payment gateway timeout',
    'error',
    ['order_id' => 1234]
));
```
Code language: PHP (php)

This allows logs to be processed in the background, keeping API responses fast while still persisting errors for monitoring.

Integrating with AWS CloudWatch

For apps hosted on AWS, it's best to send logs directly to **CloudWatch**. This provides centralized storage, metrics, and alerts. Install the CloudWatch Monolog driver:

```
composer require maxbanton/cwh
```
Code language: Bash (bash)

Then configure the channel:

```
// config/logging.php
'cloudwatch' => [
    'driver' => 'monolog',
    'handler' => Maxbanton\Cwh\Handler\CloudWatch::class,
    'with' => [
        'group' => env('CLOUDWATCH_GROUP', 'laravel-app'),
        'stream' => env('CLOUDWATCH_STREAM', 'production'),
        'retention' => 14,
    ],
],Code language: PHP (php)
```

Now all logs will appear in the CloudWatch console where you can create alarms for error spikes or unusual activity.

Centralized Log Aggregation

In microservices or multi-server environments, centralizing logs is essential. Common solutions include:

- **ELK Stack (Elasticsearch, Logstash, Kibana)**
- **Graylog**
- **Datadog**
- **Sentry** (error tracking with stack traces)

These tools provide real-time dashboards, error grouping, and search functionality. By shipping Monolog logs to them, you get observability across your entire infrastructure.

Structured JSON Logging

Plain text logs are difficult to parse at scale. JSON logging makes logs machine-readable and easy to filter in aggregators.

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

Log::error('Order processing failed', [
    'order_id' => 456,
```

```php
    'user_id' => 789,
    'gateway' => 'stripe',
]);
```
Code language: PHP (php)

With JSON format, these fields become structured attributes in Elasticsearch or CloudWatch, making it possible to query by `order_id` or `user_id`.

Best Practices for Scalable Logging

- Use **JSON logging** for production.
- **Send logs asynchronously** for performance.
- Centralize logs in a **log aggregator** for search and alerting.
- Rotate logs to avoid disk overflows (`daily` driver).
- Mask sensitive data like passwords and tokens.
- Tag logs with environment (`staging`, `production`) for filtering.

Wrapping Up

Advanced logging with Monolog ensures that your Laravel applications are production-ready. By combining multiple channels, sending logs asynchronously, integrating with cloud services, and centralizing logs in aggregators, you gain full visibility into your system. Following these practices makes debugging faster, improves observability, and supports compliance requirements.

What’s Next

Continue exploring Laravel error handling and performance monitoring:

- How to Log and Monitor Errors in Laravel with Monolog
- Using Laravel Telescope to Debug Performance Issues
- Optimizing Laravel for High Concurrency with Octane