SCALYR®

WHITE PAPER

Logging Best Practices

August 2019

A good logging strategy and fast, flexible tools are crucial to effectively troubleshoot and monitor today's applications. While logging used to be an afterthought, increasing complexity within software operations is making logs the first stop in troubleshooting, rather than the last. DevOps teams need logs that are lightning fast, easy to use and quickly turn data into information. The question is no longer whether to log - it's what to log and how to do so for the best results.

This white paper examines key elements to consider when as you create your logging strategy. Inevitably, issues will arise, and they'll require time and resources to trace and resolve. Follow these best practices to make that process significantly easier. Here are the steps to build your logging strategy:

1. Know your audience
2. Set a consistent logging structure
3. Use a logging framework
4. Provide meaningful information
5. Include context
6. Use log levels
7. Know how much to log
8. Use logs for more than just troubleshooting

1. KNOW YOUR AUDIENCE

One of the key drivers of your logging strategy is your audience. There are two types of consumers of log data: humans and machines. Ideally, log entries should be written in such a way that they are consumable by both.

HUMAN CONSUMERS

Various people will review log entries for different purposes. Each type of user will need specific details.

The following users may need to review log entries:

- End-users attempting to troubleshoot a problem
- Nontechnical team members assisting with troubleshooting
- System administrators or operation engineers troubleshooting production issues
- Developers debugging during development or solving bugs in production

Take the time to understand the intended use of the logged data. This will help determine what language and information you should be using. By considering your end users at the start, your logs will provide the maximum benefit to those who need them.

MACHINE PARSEABLE

In most cases, it's also important for log entries to be machine parseable. This allows for quick and efficient data processing. You can then apply this data to a range of uses that would be impossible with human review alone. An example of this is automated alerting to help you catch errors early—or even prevent them entirely. Another benefit is being able to perform search requests and analysis on collections of logged data.

For logs to be machine parseable, they must be structured in a consistent format. An efficient way to do this is by using key-value pairs. Alternatively, log entries can be structured as JSON. This allows consumers to use several ready-made tools for parsing and storage. Both formats provide structured data that is standard and flexible.

2. SET A CONSISTENT LOGGING STRUCTURE

Structure log entries consistently throughout your application. You already know you need coding standards to make code easier to understand across an application. Likewise, it is a best practice to set standards for writing log messages.

Consider each log entry as an event. You can then use the logging standards as a guideline for the following:

- Formatting for each entry
- Deciding what information to include
- Selecting the frequency for event capture

Some types of information, such as timestamps, can be standardized and should be given a standard format. It may also be helpful to include a unique ID in each entry, which you can use to correlate requests.

When you consistently format log files to a given standard, you'll find it much easier to track and correlate issues.

3. USE A LOGGING FRAMEWORK

Avoid manually writing logs to files or handling log rotation yourself. Attempts to create your own logging framework are not only error prone, they're also usually a waste of time. Instead of designing and testing your own, choose from several excellent logging frameworks that already exist. You can use them to handle the standard parts of application logging for you. Select one that serves your purposes and implement that instead.

Another benefit of using a logging framework is standardization. This makes it much easier to keep your logging structure consistent. There's no need to worry about different log formats, logging to the wrong place, or other similar concerns. The logging framework takes care of it for you.

Lastly, most logging frameworks have been optimized to the point where they have almost no impact on performance. This also means you don't need to worry about any interference with critical application functionality. This means that the rest of your application can continue running as intended without any undesirable side effects.

4. PROVIDE MEANINGFUL INFORMATION

For developers regularly working directly with the code, many things may seem obvious. Often, developers feel that context and relevant background information can reasonably be inferred. As a result, they don't include this information in logged messages.

As a logging strategy, however, this often leads to cryptic entries in the log file. Without some preexisting knowledge about what's going on in the back-end, these messages are uninterpretable. Frequently, even the developer who authored the entries will end up frustrated by the lack of meaningful information available to help troubleshoot.

The better strategy is to assume that situations will arise where no additional information beyond the logs will be available. If the only way to troubleshoot an issue is by using the log file, what information would you need to have? This becomes increasingly important as the issue becomes more severe. Error messages should include details about how to recover from the error, if possible. If not, they should at least describe what the operation was trying to do and the actual result.

Also note that log messages should never rely on previous messages to provide context. The previous entries may be separated by other lines of log data—and in some situations, they may not appear at all.

5. INCLUDE CONTEXT

For logs to be useful, they must contain information about the context surrounding the event. Generic entries such as "error occurred" are useless if they don't include any relevant details. Such messages provide no assistance when troubleshooting and serve only to create noise within a log file. This is especially true where humans are reviewing the log files.

In case of an error, log entries should help answer such questions as:

- When did the error occur?

- What type of error occurred?
- In what section of the codebase is the error?
- How serious is the error?
- What are the possible implications of the error?

How much contextual information should you include? Consider how much relevant information would be required so that anyone reviewing the entry could answer two questions:

- What is going on?
- What is the state of the application?

This will vary depending on the application and the intended audience, but consider the following as a helpful baseline:

- The date and time, generally as a timestamp
- The use case in which the event occurred
- The name of the relevant field, function, class, or file where the event occurred
- The reason for the event
- Relevant values such as identifiers, invalid data that results in an error, or other pertinent data

Contextualizing log data in this way makes troubleshooting much faster and easier and can also make information events more meaningful.

6. USE LOG LEVELS

Not all logs are equal. Some merely contain information about regular events. Others bear critical warnings regarding the application's state. The ability to differentiate between events is vital to processing logs quickly and efficiently. Otherwise, it becomes nearly impossible to pick out the really important entries.

The following are some commonly used log levels:

- **INFO**: Generally expected user-driven or system-specific actions.
- **DEBUG**: Informational events used mostly during debugging. Remove the majority before moving into the production stage. If desired, you can keep the most useful ones and activate them for troubleshooting purposes.
- **WARN**: Events that could potentially become an error, but from which the system can automatically recover
- **ERROR**: Events that require user intervention. They can relate to anything from internal error conditions to API calls that return errors. Log all error conditions at this level.
- **FATAL**: Rarely used events in which overall application or system failure is the most logical course of action. Investigate these events immediately.

- **TRACE**: Considered a “code smell” if used in production, meaning they may indicate a deeper problem in the code. Traces are generally used during development to track bugs.
- **NOTICE**: Notable events that aren’t considered to be errors. These are less frequently used; they’re missing from some popular logging services, such as log4j.

In addition to making log entries easier to review and prioritize, using log levels also makes it easier for algorithms to sort and search through logged data.

7. KNOW HOW MUCH TO LOG

There are no definitive rules for how much to log. Each application requires the right balance between too much and too little. The right balance will depend on the application’s stability and complexity. That balance will most likely evolve over time. Ideally, you should review and evaluate logging statements whenever you refactor the code. This will keep it up to date, informative, and helpful.

When you log every detail, you are logging too much. Manual review will require endless scrolling to find useful information. As a general rule, if you find specific types of entries repeated frequently and they provide little to no additional value, you may want to remove them. If you still want to include the information, it might be possible to log it more succinctly another way. You can also use log levels to sort through such data more easily without including it in production.

By contrast, logging too little makes troubleshooting much more difficult. At a minimum, all exceptions should be logged, but you’ll also want to include information about events before and after the exception. This detail will help provide a clearer picture of what happened and what the implications are.

When in doubt, err on the side of logging too much. If you’re unsure whether a piece of information will prove useful at some point, include it. Having to sift through extra data is frustrating and time consuming. But finding yourself unable to solve a problem because you’re missing some crucial information is a much bigger issue that is potentially customer and / or revenue impacting. You can always remove logging statements later if you determine that they’re unnecessary.

More importantly, with properly structured and machine-parseable logs, you can use modern log aggregation tools. These tools let you efficiently filter and search logged data. Software will continue to become more complex. As a result, the amount of data required to understand and troubleshoot your applications will continue to increase. Eventually, these tools will prove to be indispensable.

8. USE LOGS FOR MORE THAN JUST TROUBLESHOOTING

The primary motivation for logging is generally troubleshooting. However, there are other use cases worth your consideration—namely, alerts, auditing, profiling, and statistics. Consider implementing these strategies in addition to logging for troubleshooting. This is an excellent way to maximize the value of the data that your application is already creating.

ALERTS

As mentioned briefly above, a good logging strategy can provide a system for triggering automated alerts. This goes beyond efficient troubleshooting and can help avoid some issues altogether. You can pick out particular sequences or combinations of events and have them monitored in real time to trigger an alert.

AUDITING

Auditing involves capturing events that matter from a business perspective. These are often related to either management or legal compliance. Generally, these entries provide insight into what the system's users are doing. Some examples are tracking who is using the system and what changes they are making.

PROFILING

Profiling tracks the performance of different parts of the application by using timestamps in the logged entries. Information such as how long a particular function or operation takes to run can be very insightful when reviewing overall performance.

STATISTICS

Statistical insights can be gathered by looking at how often a particular event is logged within a given time frame. It's also possible to determine if an event is happening multiple times in a row. Based on this information, you may decide that this behavior is expected, based on its historical frequency. Or, by contrast, it may raise cause for alarm.

CONCLUSION

A good logging strategy is critical to quickly track down and understand issues that arise within an application. Understanding what and how to log will make this process significantly easier. You may also want to use some of the tools that are available to organize and automate the processing of this data.