Logging levels - Logback - rule-of-thumb to assign log levels

Asked 9 years, 8 months ago Active 1 year, 10 months ago Viewed 150k times

264

182

I'm using logback in my current project.

It offers six levels of logging: TRACE DEBUG INFO WARN ERROR OFF

I'm looking for a rule of thumb to determine the log level for common activities. For instance, if a thread is locked, should the log message be set to the debug level or the info level. Or if a socket is being used, should its specific id be logged at the debug level or the trace level.

I will appreciate answers with more examples for each logging level.

logging logback

Share Edit Follow

edited Feb 17 '14 at 1:14

n00begon
3,477 3 26 41

asked Oct 20 '11 at 17:08

crimsonsky2005
2,833 4 14 12

3 Actually those levels are defined by Simple Logging Facade for Java (SLF4J), the set of interfaces intended to be a façade in front of a logging implementation. Logback is such an implementation. – Basil Bourque Jun 10 '15 at 20:28

Possible duplicate of When to use the different log levels – Eyal Roth Jun 18 '18 at 11:57

5 Answers

Active | Oldest | Votes

482

I mostly build large scale, high availability type systems, so my answer is biased towards looking at it from a production support standpoint; that said, we assign roughly as follows:

- **error**: the system is in distress, customers are probably being affected (or will soon be) and the fix probably requires human intervention. The "2AM rule" applies here- if you're on call, do you want to be woken up at 2AM if this condition happens? If yes, then log it as "error".
- **warn**: an unexpected technical or business event happened, customers may be affected, but probably no immediate human intervention is required. On call people won't be called immediately, but support personnel will want to review these issues asap to understand what the impact is. Basically any issue that needs to be tracked but may not require immediate intervention.

- **info**: things we want to see at high volume in case we need to forensically analyze an issue. System lifecycle events (system start, stop) go here. "Session" lifecycle events (login, logout, etc.) go here. Significant boundary events should be considered as well (e.g. database calls, remote API calls). Typical business exceptions can go here (e.g. login failed due to bad credentials). Any other event you think you'll need to see in production at high volume goes here.
- **debug**: just about everything that doesn't make the "info" cut... any message that is helpful in tracking the flow through the system and isolating issues, especially during the development and QA phases. We use "debug" level logs for entry/exit of most non-trivial methods and marking interesting events and decision points inside methods.
- **trace**: we don't use this often, but this would be for extremely detailed and potentially high volume logs that you don't typically want enabled even during normal development. Examples include dumping a full object hierarchy, logging some state during every iteration of a large loop, etc.

As or more important than choosing the right log levels is ensuring that the logs are meaningful and have the needed context. For example, you'll almost always want to include the thread ID in the logs so you can follow a single thread if needed. You may also want to employ a mechanism to associate business info (e.g. user ID) to the thread so it gets logged as well. In your log message, you'll want to include enough info to ensure the message can be actionable. A log like " FileNotFound exception caught" is not very helpful. A better message is "FileNotFound exception caught while attempting to open config file: /usr/local/app/somefile.txt. userId=12344."

There are also a number of good logging guides out there... for example, here's an edited snippet from JCL (Jakarta Commons Logging):

> - error - Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.
> - warn - Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.
> - info - Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.
> - debug - detailed information on the flow through the system. Expect these to be written to logs only.
> - trace - more detailed information. Expect these to be written to logs only.

Share Edit Follow

edited Aug 9 '19 at 5:20

Chege

333 4 8

answered Nov 5 '11 at 16:40

ecodan

5,429 1 14 12

1 Interesting, so I presume if you are logging API requests and a user makes a mistake with a parameter format (IllegalArgumentException), this is an INFO level, right? – Emilio Nov 11 '13 at 8:38

53

My approach, i think coming more from an development than an operations point of view, is:

- **Error** means that the execution of some task could not be completed; an email couldn't be sent, a page couldn't be rendered, some data couldn't be stored to a database, something like that. Something has definitively gone wrong.
- **Warning** means that something unexpected happened, but that execution can continue, perhaps in a degraded mode; a configuration file was missing but defaults were used, a price was calculated as negative, so it was clamped to zero, etc. Something is not right, but it hasn't gone properly wrong yet - warnings are often a sign that there will be an error very soon.
- **Info** means that something normal but significant happened; the system started, the system stopped, the daily inventory update job ran, etc. There shouldn't be a continual torrent of these, otherwise there's just too much to read.
- **Debug** means that something normal and insignificant happened; a new user came to the site, a page was rendered, an order was taken, a price was updated. This is the stuff excluded from info because there would be too much of it.
- **Trace** is something i have never actually used.

Share Edit Follow

edited Feb 22 '15 at 18:38
K-Gun
10.2k 2 51 56

answered Nov 5 '11 at 17:01
Tom Anderson
43.2k 15 82 123

20

This may also tangentially help, to understand if a logging *request* (from the code) at a certain level will result in it actually being logged given the *effective* logging level that a deployment is configured with. Decide what *effective* level you want to configure you deployment with from the other Answers here, and then refer to this to see if a particular logging *request* from your code will actually be logged then...

For examples:

- "Will a logging code line that logs at WARN actually get logged on my deployment configured with ERROR?" The table says, NO.
- "Will a logging code line that logs at WARN actually get logged on my deployment configured with DEBUG?" The table says, YES.

from [logback documentation](#):

> In a more graphic way, here is how the selection rule works. In the following table, the

vertical header shows the level of the logging request, designated by p, while the horizontal header shows effective level of the logger, designated by q. The

intersection of the rows (level request) and columns (effective level) is the boolean resulting from the basic selection rule.

level of request *p*	effective level *q*					
	TRACE	**DEBUG**	**INFO**	**WARN**	**ERROR**	**OFF**
TRACE	YES	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO
ERROR	YES	YES	YES	YES	YES	NO

So a code line that requests logging will only actually get logged if the *effective* logging level of its deployment is less than or equal to that code line's *requested* level of severity.

Share Edit Follow

edited Sep 12 '17 at 22:28

answered Sep 12 '17 at 21:57

cellepo

2,877 2 31 49

8

I answer this coming from a component-based architecture, where an organisation may be running many components that may rely on each other. During a propagating failure, logging levels should help to identify both which components are affected and which are a root cause.

- **ERROR** - This component has had a failure and the cause is believed to be internal (any internal, unhandled exception, failure of encapsulated dependency... e.g. database, REST example would be it has received a 4xx error from a dependency). Get me (maintainer of this component) out of bed.
- **WARN** - This component has had a failure believed to be caused by a dependent component (REST example would be a 5xx status from a dependency). Get the maintainers of THAT component out of bed.
- **INFO** - Anything else that we want to get to an operator. If you decide to log happy paths then I recommend limiting to 1 log message per significant operation (e.g. per incoming http request).

For all log messages be sure to log useful context (and prioritise on making messages human readable/useful rather than having reams of "error codes")

- **DEBUG** (and below) - Shouldn't be used at all (and certainly not in production). In development I would advise using a combination of TDD and Debugging (where necessary) as opposed to polluting code with log statements. In production, the above INFO logging, combined with other metrics should be sufficient.

A nice way to visualise the above logging levels is to imagine a set of monitoring screens for each component. When all running well they are green, if a component logs a WARNING then it

will go orange (amber) if anything logs an ERROR then it will go red.

In the event of an incident you should have one (root cause) component go red and all the affected components should go orange/amber.

Share Edit Follow

answered Sep 1 '14 at 9:13

Phil Parker
1,219 11 10

2 +1 for the monitor analogy -- really helps visualize why you have the levels set up that way – emragins Sep 8 '14 at 19:55

3

Not different for other answers, my framework have almost the same levels:

1. Error: critical logical errors on application, like a database connection timeout. Things that call for a bug-fix in near future
2. Warn: not-breaking issues, but stuff to pay attention for. Like a requested page not found
3. Info: used in functions/methods first line, to show a procedure that has been called or a step gone ok, like a insert query done
4. log: logic information, like a result of an if statement
5. debug: variable contents relevant to be watched permanently

Share Edit Follow

answered Aug 7 '13 at 18:03

blagus
1,456 3 15 20