

We just launched a new product: Collectives on Stack Overflow. How do they work? [Listen to learn more.](#)

## When to use the different log levels

Asked 11 years, 5 months ago   Active 3 months ago   Viewed 387k times

There are different ways to log messages, in order of fatality:

626

1. FATAL

2. ERROR

3. WARN



356

4. INFO



5. DEBUG

6. TRACE

How do I decide when to use which?

What's a good heuristic to use?

[logging](#) [conventions](#)

Share Edit Follow

edited Mar 8 at 23:39



[codeforester](#)

29.3k 11 81 106

asked Jan 8 '10 at 22:19



[raoulsson](#)

12.2k 11 39 61

13 Quite broad question. So more than one answer is possible, depending on the actual circumstances of logging. Someone will miss notice in this collection someone will not ... – [Wolf](#) Oct 4 '17 at 13:06

2 @Wolf where would 'notice' fall under this hierarchy? Just for the record... – [pgblu](#) Jul 20 '18 at 14:17

2 notice might well be missing because some popular logging services like log4j do not use it. – [pgblu](#) Jul 20 '18 at 14:41

notice falls between warning and info. [datatracker.ietf.org/doc/html/rfc5424#page-11](http://datatracker.ietf.org/doc/html/rfc5424#page-11) – [datashaman](#) Jun 17 at 7:48

20 Answers

Active

Oldest

Votes

I generally subscribe to the following convention:

892

- **Trace** - Only when I would be "tracing" the code and trying to find one part of a function specifically.



- **Debug** - Information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.).
- **Info** - Generally useful information to log (service start/stop, configuration assumptions, etc). Info I want to always have available but usually don't care about under normal circumstances. This is my out-of-the-box config level.
- **Warn** - Anything that can potentially cause application oddities, but for which I am automatically recovering. (Such as switching from a primary to backup server, retrying an operation, missing secondary data, etc.)
- **Error** - Any error which is fatal to the operation, but not the service or application (can't open a required file, missing data, etc.). These errors will force user (administrator, or direct user) intervention. These are usually reserved (in my apps) for incorrect connection strings, missing services, etc.
- **Fatal** - Any error that is forcing a shutdown of the service or application to prevent data loss (or further data loss). I reserve these only for the most heinous errors and situations where there is guaranteed to have been data corruption or loss.

Share Edit Follow

edited Jul 25 '17 at 1:55



Hansaka perera

3 3

answered Jan 8 '10 at 22:26



GrayWizardx

16.7k 2 28 43

Is this answer outdated? Yes | No

- 2 Why cant you merge info and warn!?! Isnt a warning about something actually "info"... – mP. Feb 2 '11 at 12:33
- 44 @mP You could merge info and warn, I guess generally they are separate because of the "panic" principle. If I have a bunch of info thats routine and just listing off state its not really worth looking at "first", but if there are tons of "warnings" I want to see those prioritized (after Errors and Fatales) so I can look into them. I would be more "panicked" at a lot of warnings than a lot of info messages. – GrayWizardx Feb 3 '11 at 1:39
- 3 @dzieciou it depends on your particular needs. Sometimes it might be fatal, sometimes nearly a warning. If I got a 4xx from a critical service I depend on and cant continue it would be an Error/Fatal for my designs. If I was trying to cache some data for later use, but could live without it it would be a WARN. The only time I see it being info would be for something like a monitoring app that is reporting the status of its URL checks. So I would INFO log that I got a 4xx from URL and move on. – GrayWizardx Jul 9 '15 at 17:24
- 2 @GrayWizardx, I think the other factor is whether this is client that received 4xx or the server that sent it. In the first case, I would be more willing to use ERROR (OMG, it's my fault I cannot prepare right request), while in the latter case I would log WARN (It's clients fault they cannot formulate requests correctly) – dzieciou Jul 10 '15 at 17:11
- 4 I suspect this is true - Debug - Information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.). . Logger.Debug is only for developers to track down very nasty issues in production e.g. If you want to print the value of a variable at any given point inside a for loop against a condition – RBT Feb 9 '17 at 8:08



324



Would you want the message to get a system administrator out of bed in the middle of the night?

- **yes -> error**
- **no -> warn**

Share Edit Follow

edited Oct 17 '17 at 8:15



Peter Mortensen

28.5k 21 95 123

answered Jan 8 '10 at 22:23



pm100

32.5k 19 69 124

Is this answer outdated? [Yes](#) | [No](#)

14 Except most people don't care if they get people out of bed at night. We've had customers raise severity-1 dockets (meant for 100% outage, i.e., national) because one site couldn't do their work (their reasoning was that it's 100% of that site). We've since "educated" them on that score. – [paxdiablo](#) Jan 8 '10 at 22:29

83 **FATAL** is when the sysadmin wakes up, decides he's not paid enough for this, and goes back to sleep. – [Mateen Ulhaq](#) May 5 '17 at 18:53



151



I find it more helpful to think about severities from the perspective of viewing the log file.

**Fatal/Critical:** Overall application or system failure that should be investigated immediately. Yes, wake up the SysAdmin. Since we prefer our SysAdmins alert and well-rested, this severity should be used very infrequently. If it's happening daily and that's not a BFD, it's lost its meaning. Typically, a Fatal error only occurs once in the process lifetime, so if the log file is tied to the process, this is typically the last message in the log.

**Error:** Definitely a problem that should be investigated. SysAdmin should be notified automatically, but doesn't need to be dragged out of bed. By filtering a log to look at errors and above you get an overview of error frequency and can quickly identify the initiating failure that might have resulted in a cascade of additional errors. Tracking error rates as versus application usage can yield useful quality metrics such as MTBF which can be used to assess overall quality. For example, this metric might help inform decisions about whether or not another beta testing cycle is needed before a release.

**Warning:** This MIGHT be problem, or might not. For example, expected transient environmental conditions such as short loss of network or database connectivity should be logged as Warnings, not Errors. Viewing a log filtered to show only warnings and errors may give quick insight into early hints at the root cause of a subsequent error. Warnings should be used sparingly so that they don't become meaningless. For example, loss of network access should be a warning or even an error in a server application, but might be just an Info in a desktop app designed for occasionally disconnected laptop users.

**Info:** This is important information that should be logged under normal conditions such as successful initialization, services starting and stopping or successful completion of significant transactions. Viewing a log showing Info and above should give a quick overview of major state changes in the process providing top-level context for understanding any warnings or errors that also occur. Don't have too many Info messages. We typically have < 5% Info messages relative to Trace.

**Trace:** Trace is by far the most commonly used severity and should provide context to understand the steps leading up to errors and warnings. Having the right density of Trace messages makes software much more maintainable but requires some diligence because the value of individual Trace statements may change over time as programs evolve. The best way to achieve this is by getting the dev team in the habit of regularly reviewing logs as a standard part of troubleshooting customer reported issues. Encourage the team to prune out Trace messages that no longer provide useful context and to add messages where needed to understand the context of subsequent messages. For example, it is often helpful to log user input such as changing displays or tabs.

**Debug:** We consider Debug < Trace. The distinction being that Debug messages are compiled out of Release builds. That said, we discourage use of Debug messages. Allowing Debug messages tends to lead to more and more Debug messages being added and none ever removed. In time, this makes log files almost useless because it's too hard to filter signal from noise. That causes devs to not use the logs which continues the death spiral. In contrast, constantly pruning Trace messages encourages devs to use them which results in a virtuous spiral. Also, this eliminates the possibility of bugs introduced because of needed side-effects in debug code that isn't included in the release build. Yeah, I know that shouldn't happen in good code, but better safe than sorry.

Share Edit Follow

answered Mar 11 '11 at 20:33



Jay Cincotta

3,932 3 19 16

👍 Is this answer outdated? [Yes](#) | [No](#)

- 
- 4 I like that it stresses to think about the audience. The key in any communication (and log messages are a form of communication) is to think about your audience and what it needs. – [sleske](#) Feb 23 '15 at 17:59
- 
- 24 About Debug <-> Trace: Note that at least in Java-land, the order of priority is "debug > trace". That's the convention all logging frameworks I know use (SLF4J, Logback, log4j, Apache Commons Logging, Log4Net, NLog). So Debug < Trace seems unusual to me. – [sleske](#) Feb 23 '15 at 18:03
- 
- 2 @Jay Cincotta Great answer. I think Debug/Trace is a matter of preference but certainly these kind of details tend to be app/company specific so its good to see differing opinions. – [GrayWizardx](#) Jul 9 '15 at 17:21
- 
- 7 I just did a survey of 7 logging frameworks across several languages. Of the three that include a "trace" severity level, all of them have it as being less severe than debug. i.e., trace < debug; I have no real-world cases where the opposite is true. @RBT It's not always possible to break into a debugger. E.g., web servers must serve requests in a finite amount of time, or exist in multithreaded

and/or server environments that might be difficult to instrument, or the bug might be rare enough that a debugger isn't an option. Or you don't know what you're looking for. – [Thanatos](#) Feb 18 '17 at 8:33

- 6 @RBT I have been working with Java systems for over 4 years. I can tell you that what you are asking is completely impractical. IDE debugging can only take you so far. **At a certain point, you simply need debug logs from another system (often a production server) in order to understand what's going on and fix the bug.** You may think that it should be **reproducible in your local IDE**, but if you work with real systems, you will find that often many bugs are unique to the production server. – [ADTC](#) Oct 22 '17 at 23:34

Here's a list of what "the loggers" have.

36

Apache log4j: [§1](#), [§2](#)

1. FATAL :

[v1.2: ..] very severe error events that will presumably lead the application to abort.

[v2.0: ..] severe error that will prevent the application from continuing.

2. ERROR :

[v1.2: ..] error events that might still allow the application to continue running.

[v2.0: ..] error in the application, possibly recoverable.

3. WARN :

[v1.2: ..] potentially harmful situations.

[v2.0: ..] event that might possible [sic] lead to an error.

4. INFO :

[v1.2: ..] informational messages that highlight the progress of the application at coarse-grained level.

[v2.0: ..] event for informational purposes.

5. DEBUG :

[v1.2: ..] fine-grained informational events that are most useful to debug an application.

[v2.0: ..] general debugging event.

6. TRACE :

[v1.2: ..] finer-grained informational events than the `DEBUG` .

[v2.0: ..] fine-grained debug message, typically capturing the flow through the application.

Apache Httpd (as usual) likes to go for the overkill: [§](#)

#### 1. **emerg:**

Emergencies – system is unusable.

#### 2. **alert:**

Action must be taken immediately [but system is still usable].

#### 3. **crit:**

Critical Conditions [but action need not be taken immediately].

- *"socket: Failed to get a socket, exiting child"*

#### 4. **error:**

Error conditions [but not critical].

- *"Premature end of script headers"*

#### 5. **warn:**

Warning conditions. [close to error, but not error]

#### 6. **notice:**

Normal but significant [[notable](#)] condition.

- *"httpd: caught SIGBUS , attempting to dump core in ..."*

#### 7. **info:**

Informational [and unnotable].

- *["Server has been running for x hours."]*

#### 8. **debug:**

Debug-level messages [, i.e. messages logged for the sake of *de-bugging*)].

- *"Opening config file ..."*

- *Opening coming me ...*

## 9. **trace1** → **trace6**:

Trace messages [, i.e. messages logged for the sake of *tracing*].

- *"proxy: FTP: control connection complete"*
- *"proxy: CONNECT: sending the CONNECT request to the remote proxy"*
- *"openssl: Handshake: start"*
- *"read from buffered SSL brigade, mode 0, 17 bytes"*
- *"map lookup FAILED: map=rewritemap key=keyname "*
- *"cache lookup FAILED, forcing new map lookup"*

## 10. **trace7** → **trace8**:

Trace messages, dumping large amounts of data

- *" | 0000: 02 23 44 30 13 40 ac 34 df 3d bf 9a 19 49 39 15 | "*
- *" | 0000: 02 23 44 30 13 40 ac 34 df 3d bf 9a 19 49 39 15 | "*

Apache commons-logging: [§](#)

### 1. **fatal**:

Severe errors that cause premature termination. Expect these to be immediately visible on a status console.

### 2. **error**:

Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.

### 3. **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.

### 4. **info**:

Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.

### 5. **debug**:



detailed information on the flow through the system. Expect these to be written to logs only.

## 6. trace:

more detailed information. Expect these to be written to logs only.

Apache commons-logging "best practices" for enterprise usage makes a distinction between **debug** and **info** based on what kind of boundaries they cross.

### Boundaries include:

- External Boundaries - Expected Exceptions.
- External Boundaries - Unexpected Exceptions.
- Internal Boundaries.
- Significant Internal Boundaries.


(See [commons-logging guide](#) for more info on this.)

Share Edit Follow

answered Apr 15 '16 at 20:14

[Pacerier](#)

76.8k 86 327 602

 Is this answer outdated? [Yes](#) | [No](#)



27



I'd recommend adopting Syslog severity levels: DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY .

See [http://en.wikipedia.org/wiki/Syslog#Severity\\_levels](http://en.wikipedia.org/wiki/Syslog#Severity_levels)

They should provide enough fine-grained severity levels for most use-cases and are recognized by existing log-parsers. While you have of course the freedom to only implement a subset, e.g. DEBUG, ERROR, EMERGENCY depending on your app's requirements.

Let's standardize on something that's been around for ages instead of coming up with our own standard for every different app we make. Once you start aggregating logs and are trying to detect patterns across different ones it really helps.

Share Edit Follow


edited Sep 28 '18 at 9:51

answered Jan 8 '13 at 7:56



[kvz](#)

4,437 1 34 30

 Is this answer outdated? [Yes](#) | [No](#)

1 I need a trace log as I want to see how things are executing in my code. What does syslog do to fix this? – [basickarl](#) Aug 24 '17 at 6:43



Traces are typically not something you'd want to transmit over syslog and I think you're free to add this level for your own interactive debugging sessions? – [kvz](#) Aug 24 '17 at 12:07

- 3 **All these expanded levels increase the complexity of logging IMO. It's best to stick to the simplest set serving the specific app's needs. For me, you should start with `DEBUG` , `INFO` , `WARNING` and `ERROR` . Developers should see all levels. SysAdmins up to `INFO` , and End Users can see warnings and errors *but only if there is a framework to alert them about it*.** – [ADTC](#) Oct 22 '17 at 23:51
- 1 *(cont'd)* As the app matures, you can expand to more levels if needed. Like both `DEBUG` and `TRACE` for developers to control the granularity. And `ERROR` expanded to other levels like `CRITICAL` , `ALERT` , `EMERGENCY` to distinguish the severity of errors and determine the action based on severity. – [ADTC](#) Oct 22 '17 at 23:51

If you can recover from the problem then it's a warning. If it prevents continuing execution then it's an error.

26

[Share](#) [Edit](#) [Follow](#)

answered Jan 8 '10 at 22:22

[Ignacio Vazquez-Abrams](#)**703k** 133 1242  
1288 Is this answer outdated? [Yes](#) | [No](#)

6 But then, what is the difference between error and fatal error ? – [user192472](#) Jan 8 '10 at 22:31

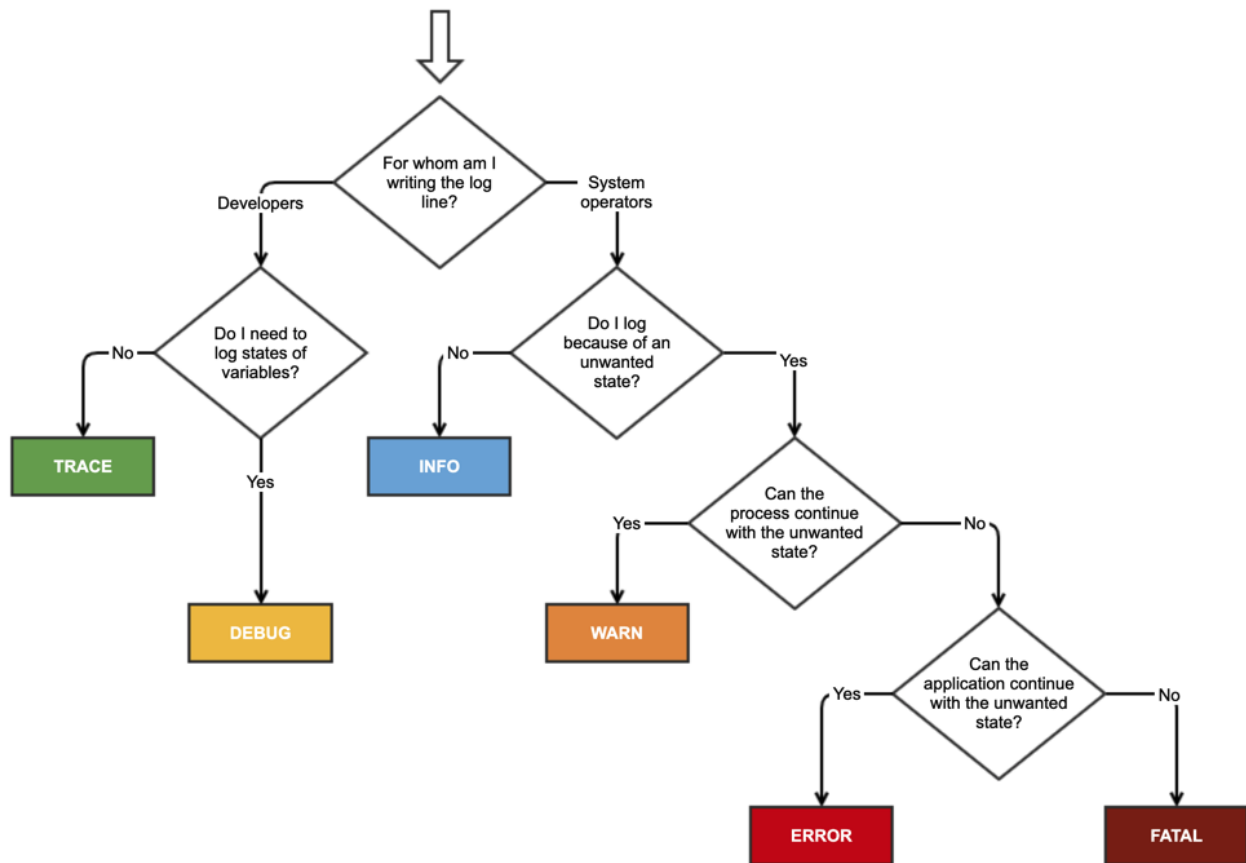
40 **An error is something that you do (e.g. read a non-existent file), a fatal error is something that is done to you** (e.g. run out of memory). – [Ignacio Vazquez-Abrams](#) Jan 8 '10 at 22:35

@IgnacioVazquez-Abrams I like your way of distinguishing. But what is your comment is based on what? AFAIK among iOS developers it's convention to write an assert which relates to `fatalError` when a file doesn't exist. Basically it's the opposite of what you said. – [Honey](#) Jul 17 '17 at 16:15

@Honey: In a mobile situation it is reasonable to consider a missing file to be a fatal error. – [Ignacio Vazquez-Abrams](#) Jul 17 '17 at 16:20

It's an old topic, but still relevant. This week, I wrote a small article about it, for my colleagues. For that purpose, I also created this cheat sheet, because I couldn't find any online.

25



Share Edit Follow

answered Nov 12 '20 at 15:24



Taco Jan Osinga

482 5 9

👍 Is this answer outdated? [Yes](#) | [No](#)

About similar with mine, except that for me, "WARN" doesn't always mean unwanted state, but can also mean "you might in some circumstances end up where you don't want to be". For example, on a mail server, if you enable authentication *but* does not require TLS, the server should log a warning. So, there's an additional diamond there before INFO – [pepoluan](#) Dec 4 '20 at 3:19

- 1 That's a great example of a warning, which I also intended with 'unwanted state'. The 'unwanted state' should be read in a broad sense. – [Taco Jan Osinga](#) Dec 4 '20 at 15:28

Warnings you can recover from. Errors you can't. That's my heuristic, others may have other ideas.

18


For example, let's say you enter/import the name "Angela Müller" into your application (note the umlaut over the u). Your code/database may be English only (though it probably *shouldn't* be in this day and age) and could therefore warn that all "unusual" characters had been converted to regular English characters.

Contrast that with trying to write that information to the database and getting back a network down message for 60 seconds straight. That's more of an error than a warning.

Share Edit Follow

edited Jan 29 '18 at 12:19

answered Jan 8 '10 at 22:22



[paxdiablo](#)  
776k 212 1484 1847

👍 Is this answer outdated? [Yes](#) | [No](#)

If the database is in a certain character set that does not include the umlaut, this input must be rejected. – [Cochise Ruhulessin](#) Dec 13 '18 at 11:35

1 Cochise, the world is rarely that black and white :- ) – [paxdiablo](#) Dec 13 '18 at 23:06

- ▲
- 6
- ▼
- 🕒
- As others have said, errors are problems; warnings are potential problems.
- In development, I frequently use warnings where I might put the equivalent of an assertion failure but the application can continue working; this enables me to find out if that case ever actually happens, or if it's my imagination.
- But yes, it gets down to the recoverability and actuality aspects. If you can recover, it's probably a warning; if it causes something to actually fail, it's an error.

Share Edit Follow

answered Jan 8 '10 at 22:32



[Michael Ekstrand](#)  
26.2k 8 54 88

👍 Is this answer outdated? [Yes](#) | [No](#)

- ▲
- 6
- ▼
- 🕒
- From RFC 5424, the [Syslog Protocol](#) (IETF) - Page 10:
- Each message Priority also has a decimal Severity level indicator. These are described in the following table along with their numerical values. Severity values MUST be in the range of 0 to 7 inclusive.

Numerical Code	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

Table 2. Syslog Message Severities

Share Edit Follow

edited Mar 7 '20 at 19:31



ADJenks

1,866 17 26

answered Apr 11 '17 at 16:06



ThangTD

1,336 14 15

 Is this answer outdated? [Yes](#) | [No](#)

5



I think that SYSLOG levels NOTICE and ALERT/EMERGENCY are largely superfluous for application-level logging - while CRITICAL/ALERT/EMERGENCY may be useful alert levels for an operator that may trigger different actions and notifications, to an application admin it's all the same as FATAL. And I just cannot sufficiently distinguish between being given a notice or some information. If the information is not noteworthy, it's not really information :)

I like Jay Cincotta's interpretation best - tracing your code's execution is something very useful in tech support, and putting trace statements into the code liberally should be encouraged - especially in combination with a dynamic filtering mechanism for logging the trace messages from specific application components. However DEBUG level to me indicates that we're still in the process of figuring out what's going on - I see DEBUG level output as a development-only option, not as something that should ever show up in a production log.

There is however a logging level that I like to see in my error logs when wearing the hat of a sysadmin as much as that of tech support, or even developer: OPER, for OPERATIONAL messages. This I use for logging a timestamp, the type of operation invoked, the arguments supplied, possibly a (unique) task identifier, and task completion. It's used when e.g. a standalone task is fired off, something that is a true invocation from within the larger long-running app. It's the sort of thing I want always logged, no matter whether anything goes wrong or not, so I consider the level of OPER to be higher than FATAL, so you can only turn it off by going to totally silent mode. And it's much more than mere INFO log data - a log level often abused for spamming logs with minor operational messages of no historical value whatsoever.

As the case dictates this information may be directed to a separate invocation log, or may be obtained by filtering it out of a large log recording more information. But it's always needed, as historical info, to know what was being done - without descending to the level of AUDIT, another totally separate log level that has nothing to do with malfunctions or system operation, doesn't really fit within the above levels (as it needs its own control switch, not a severity classification) and which definitely needs its own separate log file.

Share Edit Follow

edited Jun 18 '14 at 17:30

answered Jun 18 '14 at 17:24



volkerk

124 1 7

 Is this answer outdated? [Yes](#) | [No](#)



I totally agree with the others, and think that GrayWizardx said it best.

5



All that I can add is that these levels generally correspond to their dictionary definitions, so it can't be that hard. If in doubt, treat it like a puzzle. For your particular project, think of everything that you might want to log.



Now, can you figure out what might be fatal? You know what fatal means, don't you? So, which items on your list are fatal.

Ok, that's fatal dealt with, now let's look at errors ... rinse and repeat.

Below Fatal, or maybe Error, I would suggest that more information is always better than less, so err "upwards". Not sure if it's Info or Warning? Then make it a warning.

I do think that Fatal and error ought to be clear to all of us. The others might be fuzzier, but it is arguably less vital to get them right.

Here are some examples:

**Fatal** - can't allocate memory, database, etc - can't continue.

**Error** - no reply to message, transaction aborted, can't save file, etc.

**Warning** - resource allocation reaches X% (say 80%) - that is a sign that you might want to re-dimension your.

**Info** - user logged in/out, new transaction, file created, new d/b field, or field deleted.

**Debug** - dump of internal data structure, Anything Trace level with file name & line number.  
Trace - action succeeded/failed, d/b updated.

Share Edit Follow

edited Jun 3 '20 at 8:10

answered Jan 10 '10 at 0:40



Mawg says reinstate  
Monica

35k 92 283 512

👍 Is this answer outdated? [Yes](#) | [No](#)



G'day,

4



As a corollary to this question, communicate your interpretations of the log levels and make sure that all people on a project are aligned in their interpretation of the levels.



It's painful to see a vast variety of log messages where the severities and the selected log levels are inconsistent.

Provide examples if possible of the different logging levels. And be consistent in the info to be logged in a message.

HTH

Share Edit Follow

answered Jan 8 '10 at 22:40



[Rob Wells](#)

**34.7k** 13 77 144

Is this answer outdated? [Yes](#) | [No](#)

[Taco Jan Osinga's answer](#) is very good, and very practical, to boot.

3 I am in partial agreement with him, though with some variations.

On **Python**, there are [only 5 "named" logging levels](#), so this is how I use them:



- **DEBUG** -- information important for troubleshooting, and usually suppressed in normal day-to-day operation
- **INFO** -- day-to-day operation as "proof" that program is performing its function as designed
- **WARN** -- out-of-nominal but recoverable situation, \*or\* coming upon something that *may* result in future problems
- **ERROR** -- something happened that necessitates the program to do recovery, but recovery *is* successful. Program is likely not in the originally expected state, though, so user of the program will need to adapt
- **CRITICAL** -- something happened that cannot be recovered from, and program likely need to terminate lest everyone will be living in a state of sin

Share Edit Follow

answered Dec 4 '20 at 3:30



[pepoluan](#)

**4,879** 3 34 60

Is this answer outdated? [Yes](#) | [No](#)

I've always considered warning the first log level that for sure means there is a problem (for example, perhaps a config file isn't where it should be and we're going to have to run with default settings). An error implies, to me, something that means the main goal of the software is now impossible and we're going to try to shut down cleanly.

3



Share Edit Follow

answered Jan 8 '10 at 22:28



[dicroce](#)

**41.2k** 27 94 137

👤 Is this answer outdated? [Yes](#) | [No](#)



An error is something that is wrong, plain wrong, no way around it, it needs to be fixed.

3

A warning is a sign of a pattern that *might* be wrong, but then also might not be.



Having said that, I cannot come up with a good example of a warning that isn't also an error. What I mean by that is that if you go to the trouble of logging a warning, you might as well fix the underlying issue.



However, things like "sql execution takes too long" might be a warning, while "sql execution deadlocks" is an error, so perhaps there's some cases after all.

Share Edit Follow

answered Jan 8 '10 at 22:24



[Lasse V. Karlsen](#)

352k 94 585 781

👤 Is this answer outdated? [Yes](#) | [No](#)

- 1 A good example of a warning is that in MySQL, by default, if you try to insert more characters in a `varchar` than it is defined for, it warns you that the value was truncated, but still inserts it. But one person's warning may be another's error: In my case, this is an error; it means I made an error in my validation code by defining a length incongruous with the database. And I wouldn't be terribly surprised if another DB engine considered this an error, and I'd have no real right to be indignant, after all, it is erroneous. – [Crastr](#) Jan 8 '10 at 22:37

I too would consider that an error. In some cases, the contents is "text" (not in the datatype meaning), which means that *perhaps* it is OK to truncate it. In another case it's a code, where chopping bits off it will corrupt it or change its meaning, which is not OK. In my opinion, it's not up to the software to try to guess what I meant. If I try to force a 200 character string into a column that only takes 150 characters, that's a problem I'd like to know about. I do, however, like the distinction made by others here, that if you can recover, it's a warning, but then... do you need to log? – [Lasse V. Karlsen](#) Jan 8 '10 at 22:55

One example I could think of is: Some message taking suprisingly longer to process than usual. It might be an indication that something is wrong (maybe some other system is overloaded or an external resource was temporarily down). – [Laradda](#) Jan 5 '17 at 19:38



My two cents about `FATAL` and `TRACE` error log levels.

3

`ERROR` is when some `FAULT` (exception) occur.



`FATAL` is actually `DOUBLE FAULT`: when exception occur while handling exception.



It's easy to understand for web service.

1. Request come. Event is logged as `INFO`

2. System detects low disk space. Event is logged as `WARN`



3. Some function is called to handle the request. While processing division by zero occur. Event is logged as `ERROR`
4. Web service's exception handler is called to handle division by zero. Web service/framework is going to send email, but it can not because mailing service is offline now. This second exception can not be handled normally, because Web service's exception handler can not process exception.
5. Different exception handler called. Event is logged as `FATAL`

`TRACE` is when we can trace function entry/exit. This is not about logging, because this message can be generated by some debugger and your code has not call to `log` at all. So messages that are not from your application are marked like `TRACE` level. For example your run your application by with `strace`

So generally in your program you do `DEBUG`, `INFO` and `WARN` logging. And only if you are writing some web service/framework you will use `FATAL`. And when you are debugging application you will get `TRACE` logging from this type of software.

Share Edit Follow

answered Feb 22 '20 at 14:12



Eugen Konkov

15.9k 7 70 108

Is this answer outdated? [Yes](#) | [No](#)



I've built systems before that use the following:

1. `ERROR` - means something is seriously wrong and that particular thread/process/sequence can't carry on. Some user/admin intervention is required
2. `WARNING` - something is not right, but the process can carry on as before (e.g. one job in a set of 100 has failed, but the remainder can be processed)



In the systems I've built admins were under instruction to react to `ERRORs`. On the other hand we would watch for `WARNINGS` and determine for each case whether any system changes, reconfigurations etc. were required.

Share Edit Follow

answered Jan 8 '10 at 22:23



Brian Agnew

255k 36 317 423

Is this answer outdated? [Yes](#) | [No](#)



Btw, I am a great fan of capturing everything and filtering the information later.

1. What would happen if you were capturing at Warning level and want some Debug info related to the warning, but were unable to recreate the warning?

to the warning, but were unable to recreate the warning:

Capture *everything* and filter later!

This holds true even for embedded software unless you find that your processor can't keep up, in which case you might want to re-design your tracing to make it more efficient, or the tracing is interfering with timing (you *might* consider debugging on a more powerful processor, but that opens up a whole nother can of worms).

Capture *everything* and filter later!!

(btw, capture everything is also good because it lets you develop tools to do more than just show debug trace (I draw Message Sequence Charts from mine, and histograms of memory usage. It also gives you a basis for comparison if something goes wrong in future (keep all logs, whether pass or fail, and be sure to include build number in the log file)).

Share Edit Follow

answered Jan 10 '10 at 0:48



[Mawg says reinstate Monica](#)

35k 92 283 512

👍 Is this answer outdated? [Yes](#) | [No](#)

I suggest using only three levels

0

1. Fatal - Which would break the application.

2. Info - Info

3. Debug - Less important info

Share Edit Follow

answered Aug 28 '19 at 10:30



[Dinesh Kumar](#)

2,149 1 13 17

👍 Is this answer outdated? [Yes](#) | [No](#)



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.