* Use the SLF4J logging framework, as it is the standard logging for sling; there’s no need to introduce another logging framework.
* Do not use the log levels INFO,WARN and ERROR for debugging your code during development. You won’t have the time to fix them, and then you will pollute the production log with your debugging statements. Your operation people will hate for that, because the CQ5 standard loglevel is “INFO”.
* Instead use the log level “DEBUG” and a special logging configuration for your development environment, which logs your debug statements.
* Cleanup your debug statements when you are done with development. In production environment I recommend to setup logging on level DEBUG if there are problems with some parts of the application. The last thing I want to see are statements like “loop1”, “Inside for-loop” and “1234” (yes, I have seen them.)
* Ask yourself: Does a piece of code, which does not have logging statements, really perform any function? Even it is perfect written code with no bugs it is sometimes interesting, what information are processed. So every code should contain debug statements.
* Stacktraces are sometimes a valuable asset. Which also means, that not always a full stacktrace is required.
* And the most important: Do not use “System.out.println()”!

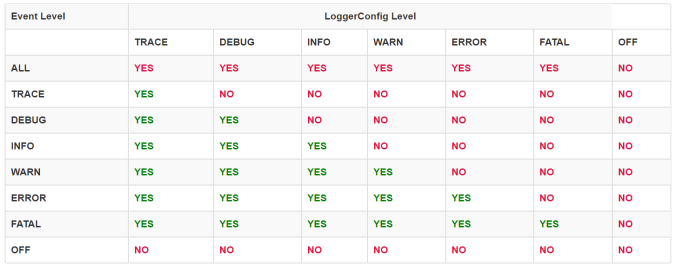
[Marketing as a Growth Engine](https://rmengji.wordpress.com/)

Loggers are the saviors for any project. Without loggers it would become impossible to identify RCA for any issue. But it is also important to understand when and how to write loggers. In this article I will go through few of the basics of the loggers and its best practices. Logging is a huge topic but i will just cover the most basic things that a developer should know.

We will use SL4J or log4j for the examples:

First let’s start with basics of when to use which level of logging:

The below table will give you an overview of the different levels in logging and the hierarchy.



It is also important to understand of what level to use in different scenarios:

* **Trace** – Only when I would be “tracing” the code and trying to find one **part** of a function specifically.  (Should be enabled only in Dev environment)
* **Debug** – Information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.)

(should be enabled only while debugging some scenario in pre-prod environments and in rare situations in production)

* **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.

(can be enabled in Production, hence [log.info](http://log.info/) statements should be with caution)

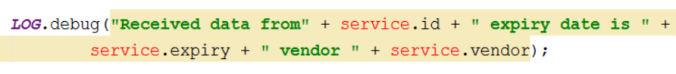
* **Warn** – Anything that can potentially cause application oddities, but for which I am automatically recovering. (Such as switching from a primary to backup server, retrying an operation, missing secondary data, etc.)

(All the service related exception or connection failures should go into warn)

* **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.

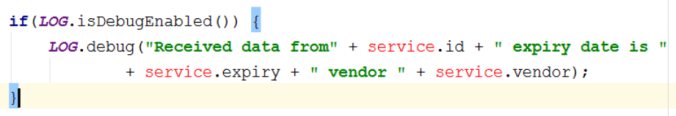
Here are few examples of the mistakes we make while logging and the better way to write it.

1. **Normal way**:



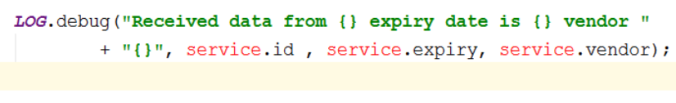
In this scenario, it’s not so bad to use logging without the check, as all that would happen is that the three strings’ arguments would be concatenated. But consider that the input string is of infinite length. This concatenation of three strings, as silly as this may sound, can take a lot of time. Here we just have a very simple, small statement with just one debug line. Imagine a thousand users concurrently calling this method, and it taking 200 milliseconds per method call. 200 milliseconds might not seem like a lot of time, but if the method is called thousands of times, this could use a lot of processing power on a server or a PC, leading to a drop in performance.

**2. Better way:**



We only want to concatenate the strings and prepare them for logging if logging is enabled. If we call a logging method and logging is not enabled, without checking first, the strings will be concatenated, but not written to the log. As a result, the concatenation would have been done unnecessarily. This is something we would like to prevent, and is the reason why we use isDebugEnabled, isInfoEnabled, or isErrorEnabled, and so on.

**3. Best Way:**



Even though this String concatenation logging is already fairly simple with Log4j and SLF4J, there is a smarter, simpler way, as shown above. We can put in curly braces in place of the variables, replace the “+” sign with a comma and then write the input String.

The framework will check if the user wants to log on debug level. If so, it will take this input, convert it to a string, if it’s not a string already, and concatenate all strings. Like in example 1, the strings won’t be concatenated unless the system is in debug mode, which helps to keep your program as efficient as possible. Here, however, we don’t have to use a conditional to check this since the debug function internally concatenates the String. This is stylistically better because it takes up fewer lines, while still providing the same functionality and protection from inefficiencies.

The same will work for all the levels of logging (Trace, Debug, Info, Warn, Error, Fatal).

I want to also talk about logging the exceptions. Please note that the below code snippets just to show the logging best practices during the exceptions.

Here are the examples and better way to write it.

Example 1: Incorrect way

Example 2:  Incorrect way : This will just print the error message, doesn’t print the stack trace.

Example 3: Best way to handle