Contents

<https://betterstack.com/community/guides/logging/how-to-start-logging-with-log4j/>

# LOG4J

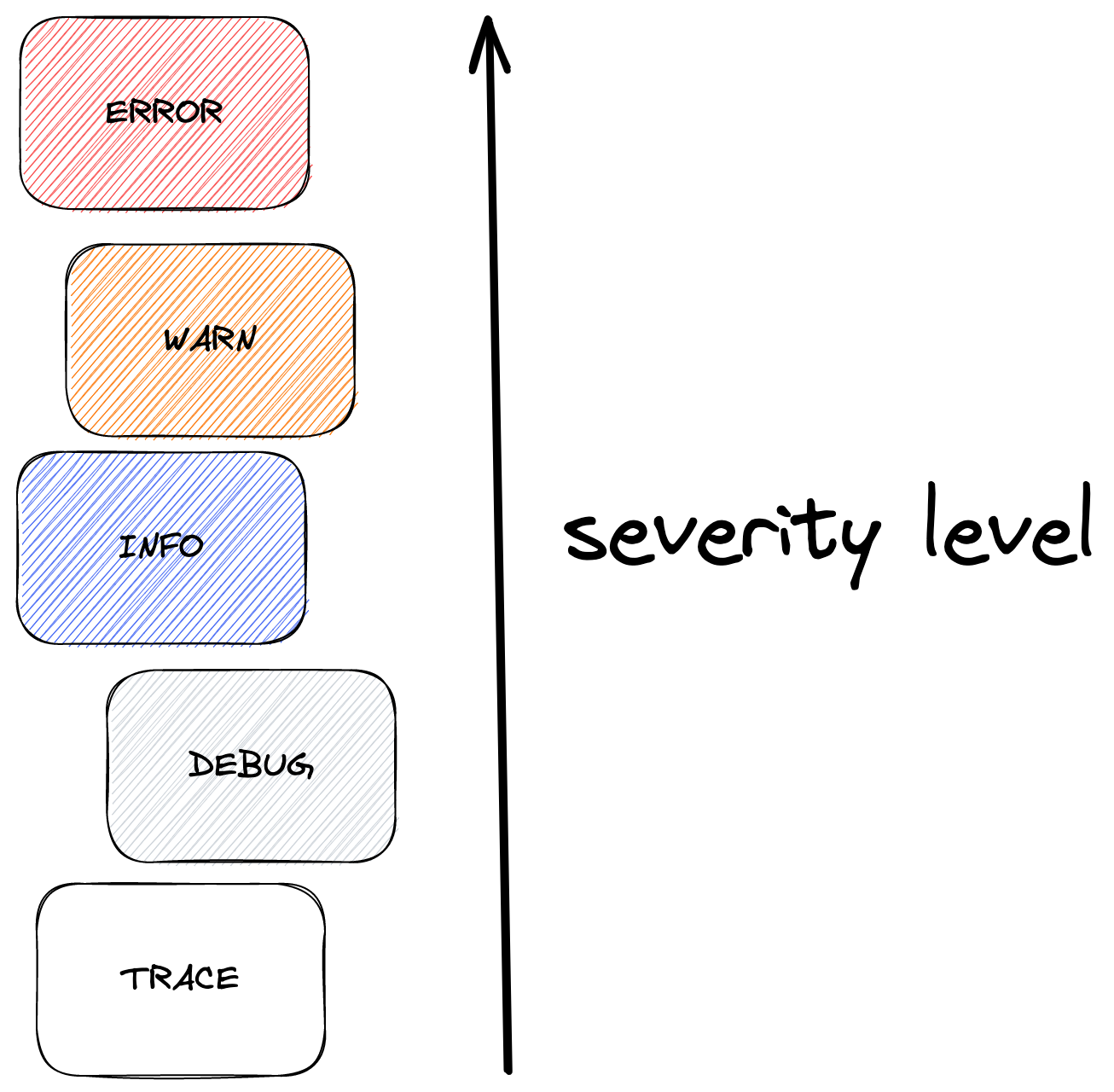
Log4j is a logging framework for Java applications created under the Apache Software Foundation. It offers features such as log levels, filters, appenders, etc. Log4j has been used extensively in the Java development community for many years and has become the de-facto option for logging in Java applications.

Log4j 2 is the latest version of the Log4j framework, released in 2014. It is a complete rewrite of the original Log4j library and introduces many new features and improvements over its predecessor. The update provides improved performance, better support for asynchronous logging, enhanced configuration options, better support for web applications, and so on.

# Log levels

The info() method here is used to log an event at the INFO level. In software development, [log levels](https://betterstack.com/community/guides/logging/log-levels-explained/) serve as a way to categorize log messages based on their severity or importance. Log4j offers six log levels by default, and each level is associated with an integer value:

* TRACE (600): this is the least severe log level, typically used to log fine-grained information about a program's execution such as entering or exiting functions, and variable values, and other low-level details that can help in understanding the internal workings of your code.
* DEBUG (500): it is used for logging messages intended to be helpful during the development and testing process, which is usually program state information that can be helpful when ascertaining whether an operation is being performed correctly.
* INFO (400): it is used for informational messages that record events that occur during the normal operation of your application, such as user authentication, API calls, or database access. These messages help you understand what's happening within your application.
* WARN (300): events logged at this level indicate potential issues that might require your attention before they become significant problems.
* ERROR (200): it is used to record unexpected errors that occur during the course of program execution.
* FATAL (100): this is the most severe log level, and it indicates an urgent situation affecting your application's core component that should be addressed immediately.



logger.trace("Entering method processOrder().");

logger.debug("Received order with ID 12345.");

logger.info("Order shipped successfully.");

logger.warn("Potential security vulnerability detected in user input: '...'");

logger.error("Failed to process order. Error: {. . .}");

logger.fatal("System crashed. Shutting down...");

In addition to these predefined log levels, Log4j also supports custom log levels. For example, if your project requires a log level VERBOSE with integer value 550, which is between levels DEBUG and TRACE, you can use the forName() method to create it.

Alternatively, you can define custom log levels directly in the configuration

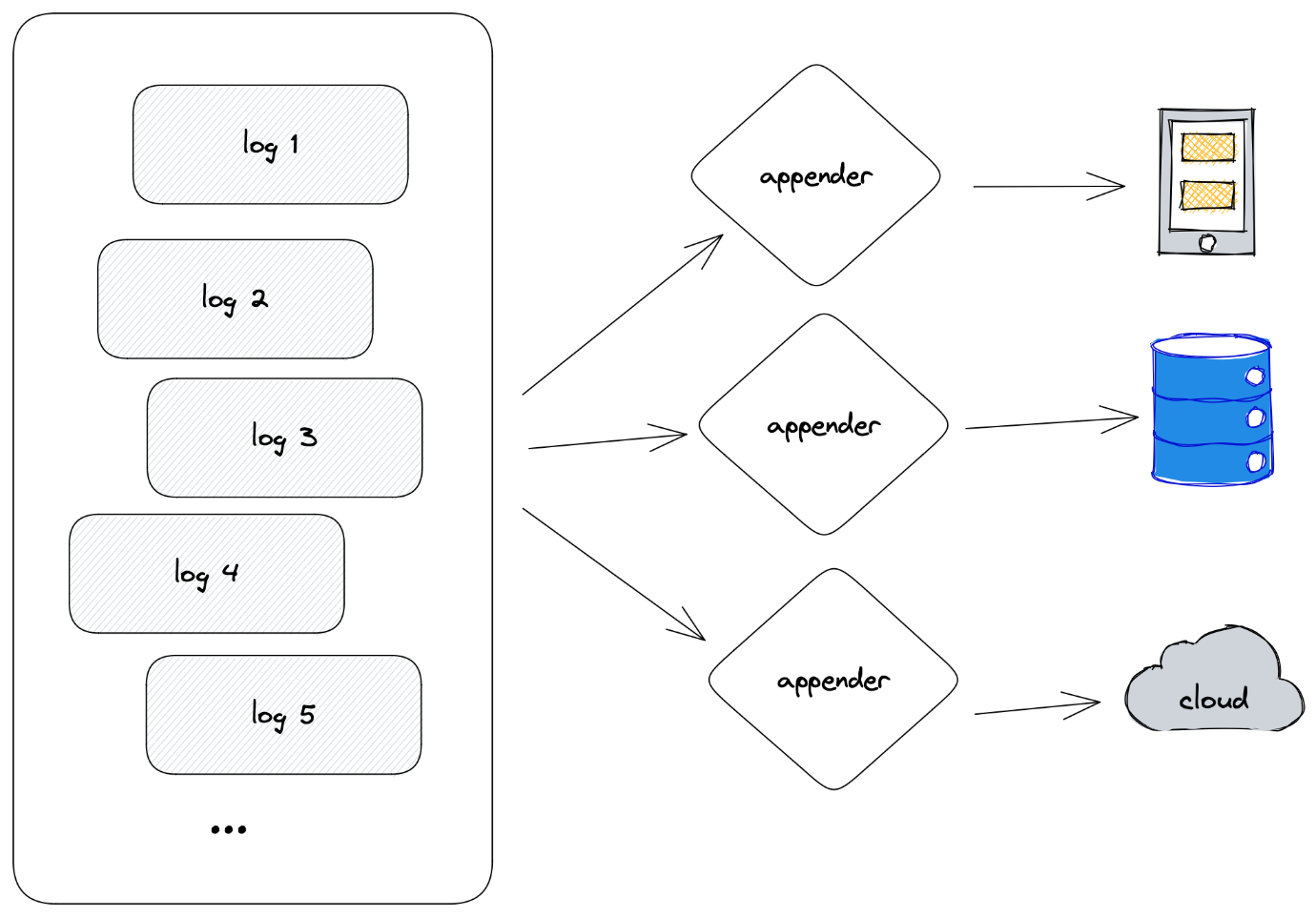


Log levels also play a crucial role in controlling the volume of logs generated by an application. By setting the appropriate log level, you can filter out less critical log messages, reducing the overall volume.

The level attribute in the above screen shot defines the minimum log level a record must have to be logged. So, for example, if you set level="info", then the trace and debug level messages will be exempted from the output. Of course, this will work for custom log levels as well.

# Appenders

In Log4j, appenders are used to forward log messages to different destinations. Log4j comes with multiple appenders out of the box, allowing you to push log messages to the console, files, databases, and so on.



Notice the second highlighted area, indicating that the Root logger is using the appender named console, and the ref should match the name parameter in Console.

Also, notice that Console has a target parameter. This parameter takes two possible values, SYSTEM\_OUT will push the logs to the standard output, and SYSTEM\_ERR will push the logs to the standard error.

Besides name and target, the Console appender also takes the following parameters:

* filter: determines whether a log message should be accepted by this appender.
* layout: defines the format and pattern the log message should follow.
* follow: determines whether the appender honors reassignments of System.out or System.err using System.setOut or System.setErr.
* direct: if set to true, this appender will write directly to java.io.FileDescriptor and bypass System.out. It can give up to 10x performance boost when the output is redirected to file or other process.
* ignoreExceptions: determines whether or not this appender will ignore exceptions. The default is true, and if set to false, the exceptions will be propagated.

# Logging to files

Logging to files is a common practice for capturing and storing log messages generated by an application. With Log4j, you can forward the log messages to files using the File appender.



The fileName parameter defines the name and path of the file this appender will write to. If the file or any of its parent directories do not exist, they will be automatically created. Rerun your application, and a logs directory and app.log file will be created

# Rotating Log files

When logging to files, a common practice to manage the logs and prevent them from growing indefinitely is [log rotation](https://betterstack.com/community/guides/logging/how-to-manage-log-files-with-logrotate-on-ubuntu-20-04/). Log4j provides a RollingFile appender for this purpose:



The filePattern parameter specifies the naming pattern for the old log files. If the file pattern ends with .gz, .zip, .bz2, .deflate, .pack200, or .xz, the resulting archive will be compressed using the compression scheme that matches the suffix.

The Policies parameter defines the condition that will trigger file rotation. There are four different triggering policies available:

* CronTriggeringPolicy: file rotation is triggered based on a [Cron expression](https://logging.apache.org/log4j/2.x/log4j-core/apidocs/org/apache/logging/log4j/core/util/CronExpression.html).

*<!-- Triggers rotation at 04:05 of every day -->*

<CronTriggeringPolicy schedule="5 4 \* \* \*" />

* OnStartupTriggeringPolicy: rotation is triggered if the log file is older than the current JVM's start time and the minimum file size is met or exceeded.

<OnStartupTriggeringPolicy minSize="20 MB" />

* SizeBasedTriggeringPolicy: rotation is triggered is file size exceeds the specified limit.

<SizeBasedTriggeringPolicy size="20 MB" />

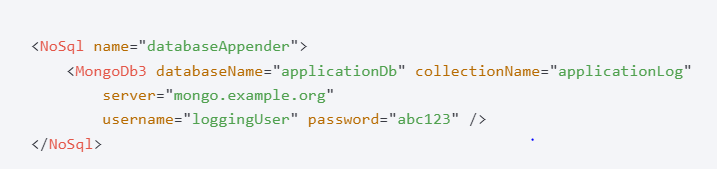
* TimeBasedTriggeringPolicy: rotation is triggered periodically based on the specified time interval.

*<!-- Rotation every 4 hours -->*

<TimeBasedTriggeringPolicy interval="4" />

After file rotation occurs, you may also specify how many archived files you wish to retain:

# Logging to other destinations





* The Syslog appender send logs to syslog.



# Formatting logs

The pattern parameter takes an expression consisting of one or more conversion specifiers, which specifies how each log message should be formatted. Each conversion specifier starts with a percentage sign (%), followed by an optional format modifier and a conversion character. The conversion character specifies the data type, such as category, priority, date, and thread name. The format modifiers control things like field width, padding, left and right justification, etc.

In the above example, the following conversion specifiers are defined:

* %d{yyyy-MM-dd HH:mm:ss.SSS}: the date and time of the logged event.
* %t: the name of the thread that processed the logged event.
* %-5level: level is the conversion character, outputting the log level. -5 is the format modifier, making sure that the log level is left justified and restricted to five characters.
* %logger{36}: the name of the logger, followed by a precision specifier, which controls the length of the logger name.
* %msg: the actual log message.
* %n: a new line character.

For different patterns

<https://logging.apache.org/log4j/2.x/manual/layouts.html#patterns>

# Adding information

In a production environment, you should include detailed information about the logged event so that the logs can help you and your team understand and troubleshoot possible issues. For example, instead of a simple message, Order shipped successfully., you could include the order number, buyer's name, destination, and so on.



The put() method will add new items to the context map, and the clearAll() method will clear the entire map. You must ensure that the logging call is placed in between. Next we will discuss how to log in a structured format so that the contextual information is included in the log message.

# Logging in structured format

So far, we've only been working with the PatternLayout, but Log4j also allows you reformat the log messages in other formats such as JSON, XML, and so on. These formats make it easier for the log records to be automatically parsed, analyzed and monitored by log management systems.

The de facto format for structured logging is JSON, which can be configured using JsonLayout. Before you proceed, ensure that you have jackson-core and jackson-databind dependencies in your pom.xml:



The JsonLayout also takes a set of [optional parameters](https://logging.apache.org/log4j/2.x/manual/layouts.html#json-layout), allowing you to customize the output. For example, by setting properties="true", you can include contextual information in the output.

# Log4j Performance

Log4j is a production-ready logging framework with many performance features, so that it can handle massive log ingestion without slowing down your application. Some of these features require special setup.

For example, there is an asynchronous logging feature, which allows the framework to execute I/O operations in a separate thread. This reduces the impact on the application's response time, unlike synchronous logging, where the application thread generates the log message and waits until it reaches its destination before continuing execution.

However, async logging does come with some downsides, such as increased memory usage, the risk of data loss in case of a system crash, and potential incorrect log message ordering, which could complicate debugging and troubleshooting. Nevertheless, Async logging can be beneficial in high-traffic environments with numerous generated log messages.

To enable asynchronous logging, you can set either one of the following system properties:

log4j2.contextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector

log4j2.contextSelector=org.apache.logging.log4j.core.async.BasicAsyncLoggerContextSelector

Or if you only want selected loggers to be asynchronous, use the <asyncRoot> or <asyncLogger> for your loggers: